RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Anonymous Bitcoin Transactions

Felix Maduakor

Bachelor's Thesis – December 4, 2017.
Chair for Network and Data Security.

Supervisor: Prof. Dr. Jörg Schwenk
Advisor: M. Sc. Martin Grothe

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

Bitcoin was created to be the electronic version of cash.[37] However, it lacks of privacy, since every transaction which ever has been processed by the bitcoin network is publicly accessible through the blockchain. While much scientific effort has been made to create P2P algorithms, which could enhance the privacy of the bitcoin network, none of those approaches has been widely adopted yet. To enhance privacy it is often recommended to use commercially driven *mixing* services. By swapping customers bitcoin, these centralized mixing services allow customer to anonymize transactions.
In this thesis, we are going to discuss (dis-)advantages of these services and provide an overview of attacking possibilities.
Furthermore, we are going to implement an attack on coinmixer.se, a frequently used centralized mixing service. Our implementation is going to deanonymize transactions which priorly have been anonymized by coinmixer.se. We are going to analyze different attacking scenarios and create an attack which could be able to attack most of the known centralized mixing services.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of High School.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Date	author

Contents

Glossary ix

Acronyms 1

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Contribution . . . 4
- 1.3 Organization of this Thesis . . . 4

2 Background 5
- 2.1 Bitcoin . . . 5
  - 2.1.1 Blockchain . . . 5
  - 2.1.2 Transactions . . . 6
    - 2.1.2.1 P2PKH and P2SH transactions . . . 7
    - 2.1.2.2 Multisignature transactions . . . 8
    - 2.1.2.3 Replace-By-Fee . . . 8
    - 2.1.2.4 Locktime, sequence numbers and version . . . 9
    - 2.1.2.5 Transaction fee . . . 10
    - 2.1.2.6 Transaction time and IP addresses . . . 10
    - 2.1.2.7 Example transaction . . . 11
- 2.2 Fungibility . . . 12
- 2.3 Privacy in Bitcoin . . . 13
- 2.4 Mixing techniques . . . 18
  - 2.4.1 Decentralized mixing (P2P mixing) . . . 18
  - 2.4.2 Centralized Mixing Services (CMS) . . . 19
  - 2.4.3 Off chain mixing . . . 19

3 Centralized Mixing Services 21
- 3.1 Advantages . . . 21
- 3.2 Disadvantages . . . 21
- 3.3 Attacker models . . . 22
- 3.4 Possible attacks . . . 22
  - 3.4.1 Blockchain analysis . . . 22
    - 3.4.1.1 Taint Analysis . . . 23
  - 3.4.2 Sybil attack . . . 24
  - 3.4.3 Web security bugs . . . 24
  - 3.4.4 DDoS . . . 25

3.4.5 Attacks on the Bitcoin protocol . . . 26
3.4.5.1 Double spending . . . 26
3.4.5.2 Stale blocks . . . 26
3.4.5.3 Replay attack on forks . . . 27
3.4.5.4 Transaction malleability . . . 28
3.4.6 Conclusion . . . 28

4 Attack on coinmixer.se 29
4.1 Functionality of coinmixer.se . . . 29
4.1.1 Optional setting: Multiple addresses . . . 29
4.1.2 Optional setting: Time delay . . . 30
4.1.3 Mixing fee . . . 30
4.2 Attacker Model . . . 31
4.3 Attacking Method . . . 31
4.3.1 Steps to break coinmixer.se . . . 31
4.4 Identifying coinmixer.se's network . . . 32
4.4.1 Characteristics of customer's input transactions . . . 33
4.4.2 Characteristics of coinmixer's output transactions . . . 33
4.4.3 Identifying customer's and coinmixer's transactions . . . 37
4.5 Crawler . . . 40
4.5.1 Gathering blockchain data . . . 42
4.5.2 Data structure . . . 42
4.5.3 Forward crawling . . . 43
4.5.4 Backward crawling . . . 47
4.5.5 Incorrect transaction distinguishing . . . 49
4.6 Deanonymization . . . 50
4.7 Results . . . 51

5 Conclusion 59
5.1 Related Work . . . 59
5.2 Future Work . . . 60

List of Figures 63

List of Tables 65

List of Listings 66

Bibliography 67

A Database structure 71

B Python Code 77

Glossary

Block Generation Time Average time required till a new block is found.

Blockchain Difficulty The required time to mine a Bitcoin block is based on a dynamically calculated blockchain difficulty.

Chain-Split A split of the current Bitcoin blockchain.

Change/Refund Address An address which receives the refund of a sent transactions.

Fee Bitcoin miners receive a fee for confirming transactions..

Fork An intentional chain-split.

Untainted Bitcoins Anonymized Bitcoins.

input Address An Address which spends unspent outputs in transaction.

Input Transaction A customer's transaction which starts a mixing process. Customer sends tainted Bitcoins.

Letter Of Guarantee A letter signed by the coinmixer, which provides the input and output addresses of the mixing process.

Mapping A mapping of possible input and output transactions, which deanonymizes a transaction.

Mixing The process of anonymizing Bitcoins.

Off-Chain Transaction A Bitcoin transaction, which is processed by the Bitcoin network but not stored into the blockchain.

On-Chain Transaction A transaction which is processed and stored in the Bitcoin blockchain.

Online Wallet A website through which Bitcoin wallets can be created.

Output Address An Address which receives transaction's outputs.

Output Transaction A transaction sent by Coinmixer's transaction to customers. Coinmixer sends untainted Bitcoins.

Replay-Protection A protection to separate two blockchains.

Unspent Transaction Output Bitcoins which still can be spent in a transaction.

Wallet/Client Software A software which can be used to create, receive and send Bitcoin transactions.

Acronyms

BIP Bitcoin Improvement Proposals.

BTC Bitcoin.

CMS Centralized Bitcoin Mixing Service.

Coins Bitcoins.

CPFP Child Pays For Parent.

fss RBF First-Seen-Save Replace-By-Fee.

P2PKH Pay To PubKey Hash.

P2SH Pay to script hash.

RBF Replace-By-Fee.

RPC Remote Procedure Call.

Sat Satoshi.

Seq Sequence Number.

TX Transaction.

UTXO Unconfirmed Transaction Output.

1 Introduction

At the time of writing, Bitcoin is the leading P2P cryptocurrency [4]. It's based on a decentralized P2P structure and was introduced through Satoshi Nakamoto in 2008. It was build to be the digital form of cash. Through Bitcoin it is possible to transfer assets without an intermediary. [37]
However, through the implementation of the Bitcoin blockchain it is possible to trace every transaction. Bitcoin does not offer *strong privacy guarantees* [42]. If Bitcoin gets widely adopted as a payment method, there may be the need to enhance the privacy of the network.
Recently multiple P2P algorithms have been published, which aim to enhance privacy in the Bitcoin network [44, 40, 42, 30]. Tim Ruffing, Pedro Moreno-Sanchez and Aniket Kate introduced *coinshuffle ++* as a decentralized *Bitcoin mixing protocol* which is fully compatible with the current Bitcoin system [40]. However, none of the mentioned privacy enhancing algorithmns seems to be widely adopted.
Bitcoin Core, as the reference client of Bitcoin, does not implement any mixing protocol [3].
Another way to enhance privacy of Bitcoin transactions is based on commercially driven centralized mixing services. In this theses we are going to focus on centralized mixing services. We are going to discuss their (dis-)adventages and present possible attacking approaches. Furthermore, we are going to implement an attack on coinmixer.se, a frequently used centralized mixing service. Our aim is to implement an attack which is able to deanonymize transactions, which priorly have been anonymized by coinmixer.se. We are going to discuss a general attacking approach which could be able to successfully attack most of the commonly used centralized Bitcoin mixing services.

1.1 Motivation

Privacy is an important aspect of cryptocurrencies. There have been created several cryptocurrencies which aim to enhance the privacy of transactions like *Monero* or *Zerocash*. Based on a zero-knowledge proof, the cryptocurrency *Zerocash* is able to offer *strong privacy guarantees* [42].
While implementations of P2P mixing algorithmns in Bitcoin clients could enhance privacy in Bitcoin, they have not been adopted widely yet. Centralized Bitcoin mixing service are often recommended to be used to enhance privacy [9].

However, we want to show that there may be a general attacking approach which could lead to the deanonymization of most transactions, which were processed by centralized mixing services.
We are going to show, that even though the internals of a mixing service could be based on a secure mixing algorithm, the implementation of this algorithm as a centralized service could easily lead to vulnerabilities. Even a small information leak could lead to the deanonymization of every transaction which the service ever processed. Since all Bitcoin transactions are based on the blockchain, an identified information leak could deanonymize transaction which were processed years ago.

1.2 Contribution

While we attack coinmixer.se as a centralized mixing service, we were able to show the general problems of centralized mixing services. We implemented a tool which is able to deanonymize transactions which priorly have been anonymized by coinmixer.se. While our implementation is based on coimixer.se, it easily easily can be adopted to work with other centralized mixing services. We were able to create and implement an attacking approach which is able to break nearly all known centralized mixing services. Furthermore, it may be even applied to cryptocurrency networks.

1.3 Organization of this Thesis

In chapter two we are going to address the basic concepts of the Bitcoin protocol and transactions. We are going to discuss how Bitcoin transactions are built and which features have been recently introduced. Furthermore, we are going to discuss why privacy is important in the Bitcoin ecosystem and how the standard implementation of the Bitcoin protocol tries to enhance privacy. We also address different techniques to enhance privacy in the Bitcoin network.
In chapter three we are going to discuss centralized mixing services. We are going to address possible advantages and disadvantages of centralized mixing services. Furthermore, we discuss the most common attacking possibilities on mixing services.
In chapter four we are going to attack a centralized mixing service. Coinmixer.se anonymizes Bitcoin transactions. Based on a simple blockchain analysis we will deanonymize transactions which priorly have been anonymized by the service. Our attack is going to be implemented as a python script.
Chapter five gives an overview on our results and address show future work.

2 Background

In this chapter we will give some technical background information about the basic concepts behind Bitcoin. We will discuss the importance of privacy in Bitcoin and different approaches which have been developed to enhance privacy in the Bitcoin ecosystem.

2.1 Bitcoin

Bitcoin is peer-to-peer electronic cash through which digital transactions can be sent without the need to use an intermediary. While cash payments are typically made directly between individuals, there haven't been a digital way to make this kind of transactions until Bitcoin solved this problem. [37]
The basic design idea of the Bitcoin network has been created by a developer with the pseudonym *Satoshi Nakamoto*. He published the basic concepts through his whitepaper *Bitcoin: A Peer-to-Peer Electronic Cash System* in 2008. citenakamoto2008bitcoin No specific thrid party of the Bitcoin network needs to be trusted. However, big parts of the Bitcoin network and the client software should be trusted, otherwise some specific attacking approaches may be possible [11]. This property has been achieved through the sophisticated use of cryptographic models.
The Bitcoin protocol is entirely open source. There are several groups of developers around Bitcoin. Unlike government issued currencies, changes in the Bitcoin protocol require consensus of broad parts of the Bitcoin ecosystem. The realization of these changes is time consuming, since multiple parties have to be actively involved in a successful protocol update. A dynamic monetary policy, which is performed on government issued currencies, cannot be applied to Bitcoin [13]. As the first cryptocurrency, Bitcoin created a new type of digital assets. In recent years, hundreds of cryptocurrencies, which differ in specific properties to Bitcoin, have been developed [4].

2.1.1 Blockchain

The Bitcoin blockchain is an implementation of a public ledger. Every transaction that ever have been confirmed by the network is stored in this blockchain. Through cryptographic primitives the blockchain is protected against modifications. All transactions are publicly availbile, because they are permanantly stored in the

blockchain. Every transaction stored in the blockchain is publicly accessible and cannot be changed after insertion. [9]

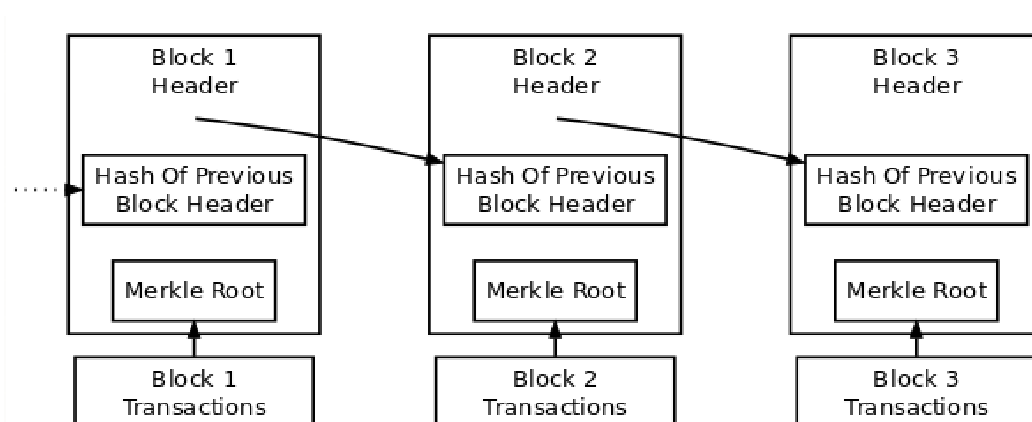

Figure 2.1: Simplified Bitcoin blockchain [9]

The Bitcoin blockchain consists of several blocks that have been cryptographically connected to each other. The header of a previous block has to be hashed and stored into the next block header. A block consists of multiple transactions which are hashed into a merkle tree. [9]
Bitcoin uses the SHA-256 hash function to cryptographically connect blocks in the blockchain. Changes in the blockchain should not be able to be done without changing at least one hash value that results in corrupting this version of the blockchain. A corrupted version of the blockchain will be rejected by the network. It should be cryptographically hard to create two versions of the same block with different data but the same hash. While this strict assumption holds in practice, it brings some security issues.
Especially the *transaction malleability* bug made it possible to generate multiple valid transactions with different transaction hashes, since even changing a single bit of the hash functions' input results in a new SHA-256 hash. It is likely that this bug led to the loss of more than 302.000 Bitcoins. [19]
The transaction malleability bug has recently been fixed through the implementation of segwit. However, this bug can still be used in some specific circumstances. [16]
The smallest unit of Bitcoin is named *satoshi*. One satoshi equals 0.00000001 BTC (8 decimals).

2.1.2 Transactions

There are two different types of Bitcoin transactions. A Transaction, which is sent to a miner who produced a new Bitcoin block, so called *coinbase transaction*, and

typical Bitcoin transactions that are sent through a Bitcoin address. In the following sections we are not going to discuss coinbase transactions, since they only can be generated by miners. Some of the specific properties we are going to describe may not apply to coinbase transactions.

As we have already seen in the previous section, transactions are a part of a Bitcoin block. Every transaction consists of inputs and outputs. For every input of a transaction, there has to be a previous output. An output can only be spent once. Outputs that haven't yet been used as an input are called *unspent transaction output* (UTXO). In a general view, the transaction of Bitcoins equals the transfer of UTXO. The amount of UTXO, which a Bitcoin address is able to spend, is the amount of Bitcoins which is often called as the *balance* of a Bitcoin address. It is important to know, that every transaction has to spend the full UTXO. So if a user is able to spend 0.5 BTC though an UTXO, he has to spend the full 0.5 BTC. If he only wants to send 0.30 BTC, he has to add another address which is controlled by himself to be able to receive the change of 0.20 BTC.

2.1.2.1 P2PKH and P2SH transactions

Bitcoin transactions are evaluated through a forth-like, stack-based script language [9].
There are two different types of non-coinbase transactions which are commonly used. Pay-To-Public-Key-Hash (P2PKH) transactions and Pay-To-Script-Hash (P2SH) transactions. P2PKH is the most commonly used type of transaction [9]. In P2PKH transactions the receivers' hashed public-key is added to the transaction. If the receiver wants to spend received Bitcoins, he has to prove that he is the owner of the public-key. He does this by signing the transaction he wants to send with his private key.

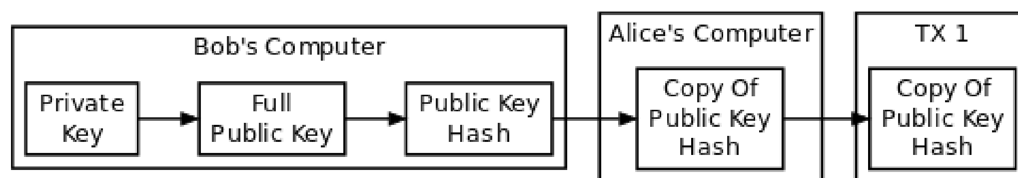

Figure 2.2: P2PKH transaction [9]

In this figure 2.2, Alice sends Bitcoins to Bob. She adds his hashed public-key to the transaction. If Bob wants to make use of the received Bitcoins, he has to prove that he is the owner of the public-key.

In P2SH transactions the sender sends his coins to a script called *redeem script*. This redeem script is provided through the receiver. Whenever the receiver tries to spend UTXO which were sent to the P2SH address generated by him, the redeem script

will get evaluated. Only if the redeem script evaluates to *True* the transactions' UTXO can be spent by the receiver. While P2SH could be seen as general smart contracts, most of the Bitcoin nodes will only accept standardized redeem scripts [9].
Today P2SH transactions are mainly used to provide multisignature transactions. In figure 2.3 the evaluation of the redeem script is based on multiple signatures.

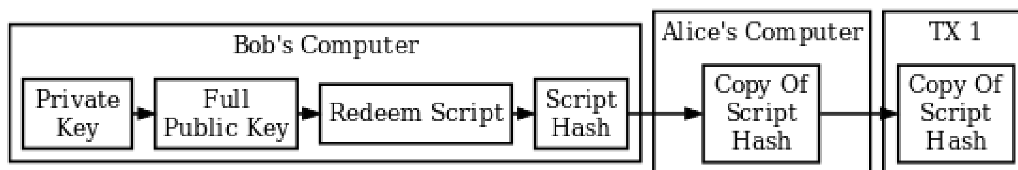

Figure 2.3: P2SH transaction [9]

In figure 2.3, Bob created a redeem script. Bob can choose on which conditions the redeem script evaluates to *True*. Bob sends Alice the hash of the redeem script. Alice sends the coins to the received hash. If Bob wants to spend the received coins, the redeem script has to evaluate to *True*.

Based on the first letter of an address it can be distinguished if it's a P2PKH address or a P2SH address.

Table 2.4: Example of P2PKH and P2SH transaction hashes

First letter	**Type of address**	**Example**
1	P2PKH	1E9bQsqFtf7SwTpZaiNvjq2d3BLNWo82ko
3	P2SH	3H8tiRY6GfcTgjn2bDRjs9AwAgaT7KVE2P

It's worth mentioning that there are two more types of Bitcoin transactions which were introduced through the segwit soft-fork (P2WPKH and P2WSH) [29]. However, they are very similar to P2PKH and P2SH, since they are mainly focused on moving the signature script to another location [9].

2.1.2.2 Multisignature transactions

Multisignature transactions are m-of-n transactions, which require at least m signatures of n public-key hashes that were provided in the transaction. The redeem script evaluates to *True* if at least m correct signatures are provided. [9]

2.1.2.3 Replace-By-Fee

Opt-in Replace-By-Fee (RBF) is an optional feature which was introduced through BIP0125 and had been implemented in *Bitcoin Core* nodes since version 0.12 [18].Through the opt-in RBF feature a transaction can be resent with a higher fee as

long as it stays unconfirmed. The previously sent transaction will be ignored. This feature can be useful if a transaction has been sent with a too low transaction fee and might not get confirmed in the near future.
Opt-in RBF is activated whenever the sequence numbers of a transactions' inputs are set to a value lower than 0xfffffffe (4294967294) [18].
It's worth mentioning that there are also further methods to replace unconfirmed transactions like first-seen-safe Replace-By-Fee (fss RBF) and full Replace-By-Fee (full RBF). Since all RBF methods are enforced through Bitcoin nodes, it does not necessarily mean that miners have to realize them.

2.1.2.4 Locktime, sequence numbers and version

The relative locktime is a new feature in the Bitcoin protocol, which has been introduced with BIP0068 [23]. Through the locktime a sender of a transaction is able to define the earliest time at which a transaction can be added to the blockchain. The locktime can be specified through a unix timestamp or a specific block height. Before the locktime is reached, the sender is able to cancel and renew the transaction. To achieve this, he has to create a transaction which spends the same UTXO but has no locktime or a lower locktime than the original transaction specified. If one of these transactions has been added to the blockchain, the other transaction will be ignored by the network. Typically the transaction with the lowest locktime will be inserted first in the blockchain. However, changing the transaction fee could influence the order of transaction processing done by the miner. Sequence numbers were introduced as a feature which should make it possible to update unconfirmed transactions. Whenever the sequence number of a UTXO would be set with a higher sequence number, transactions which use the same UTXO but a lower sequence number should be ignored. However, this feature has not been implemented properly and is not enforced by the network [23].
Yet still features like (opt-in) RBF and locktime make use of sequence numbers. Whenever the sequence numbers of UTXO are set to the maximum-value (0xffffffff), the transactions is processed as a finalized transaction and will be added to the blockchain. In this case a specified locktime is going to be ignored. No further changes can be applied to this transaction. So whenever a sender wants to make use of the locktime feature, he has to set the sequence numbers to values lower than 0xffffffff (4294967294).
Bitcoin Core, the refence client of Bitcoin, sets per default the sequence numbers of transactions to the maximum-value 0xffffffff.

Table 2.5: Transaction's settings based on sequence number

sequence number	**features**
0 - 0xfffffffd	(opt-in) RBF activated | locktime can be specified
0xfffffffe	(opt-in) RBF deactivated | locktime can be specified
0xffffffff	(opt-in) RBF deactivated | locktime deactivated

It's important to state, that if the locktime features should be activated, the transaction's version has to be set to at least *2* [23].

2.1.2.5 Transaction fee

Every transaction can specify a fee, which the miner receives who adds the transaction to the blockchain. Miners can choose transactions which they want to confirm and add to the blockchain. Typically miners try to make the most profit from confirming transactions and automatically choose transactions which generate most fees. [5]
Since a block has a specific block size the number of transactions which can be added to a block is restricted [15]. While miners typically choose transactions with the highest fee, transactions sent with a low fee have to wait till the high feed transactions have been processed. If the sender chose a too low fee, there are several methods of still getting the transaction confirmed like opt-in Replace-By-Fee (opt-in RBF), first-seen-safe Replace-By-Fee (fss RBF), full Replace-By-Fee (full RBF) or Child Pays For Parent (CPFP). However, miners do not have to accept transactions which made use of these options.
Since we focus on privacy aspects, we are not going into further details of the mentioned methods.
It is important to know, that there is no field in the Bitcoin protocol which specifies the miner fee. The fee of a transaction is calculated by the result of the inputs subtracted from the outputs:
$Fee = value\,of\,inputs - value\,of\,outputs$

2.1.2.6 Transaction time and IP addresses

While many blockchain explorers show transaction times and IP addresses, this information is not stored in a transaction [9]. This information is gathered through the logs of a node and only describes the timing and origin of incoming connections of this specific node. However, the transaction time will be in most cases very accurate since the Bitcoin network is well connected. Still, it should be known that if a transaction is not spread through the whole network, it still can be inserted in a block. By most of the nodes, the transaction time of this transaction would be set equal to the block time.
While this circumstance normally should not be an issue, it shows clearly the problem of correct time measurements in decentralized networks, which can even be used for attacking attempts. The same origin problem applies to shown IP addresses. Blockchain explorer typically show logged IP addresses of relaying nodes. In most cases this won't be the IP address of the signer of the transaction.

2.1.2.7 Example transaction

All blockchain data are saved binary. However, there is a remote procedure call (RPC) interface implemented in Bitcoin Core which allows to submit customized JSON formatted transactions. This kind of transaction is called *simple raw transaction*. [20] We are going to use the *blockchain.info* API and *Bitcoin-cli* RPC to gather necessary blockchain and transaction data JSON formatted.
As an example we will analyze the following transaction:
81b46084e181eea9d846b2400e91a545178e61ca4a7730e9c0e3c15f7322a778
We are using *Bitcoin-cli* to receive the transaction data JSON formatted, online tools tools to achieve this can be used likewise. We are only going to focus on the most interesting parts of the transaction.


```
{ "result": {
    "txid": "81b46084e181eea9d846b2400e91a545178e61ca4a7730e9c0e3c15f7322a778",
    "hash": "81b46084e181eea9d846b2400e91a545178e61ca4a7730e9c0e3c15f7322a778",
    "size": 517,
    "vsize": 517,
    "version": 2,
    "locktime": 489733,
    "vin": [
      {
        "txid": "b5a4d2d97d5b8dcfbccdb366dad557c507bdf6219bbd83eb7f9ce4f719995b20",
        "vout": 0,
        "scriptSig": {"hex": "4730..."},
        "sequence": 4294967294
      },
      {
        "txid": "44b88be87299336ec02f94677ebaac6f63e37dbf21e38ba0829196937dc84a33",
        "vout": 0,
        "scriptSig": {"hex": "473044022...},
        "sequence": 4294967294
      },
      {
        "txid": "e0c68b0b557700f26f65e27aea5ab4003e6a2a28af140db7c4508429d6b23873",
        "vout": 1,
        "scriptSig": {"hex": "4730440..."},
        "sequence": 4294967294
      }
    ],
    "vout": [
      {
        "value": 0.20000000,
        "n": 0,
        "scriptPubKey": {
          "reqSigs": 1,
          "type": "scripthash",
          "addresses": [
            "3H8tiRY6GfcTgjn2bDRjs9AwAgaT7KVE2P"
          ]
        }
      },
      {
        "value": 0.01359680,
        "n": 1,
        "scriptPubKey": {
          "type": "pubkeyhash",
          "addresses": [
            "1E9bQsqFtf7SwTpZaiNvjq2d3BLNWo82ko"
          ]
        }
      }
    ],
    "blockhash": "000000000000000000ec8d98e4ccdf87b35b21a980098d316f166442fe47c81c",
    "confirmations": 3973,
    "time": 1508008653,
    "blocktime": 1508008653
  },}
```


Listing 2.6: JSON-formatted Bitcoin transaction

Some parameters have been truncated or removed. As we can see in listing 2.7, the transaction has three inputs and two outputs specified. One of the outputs is a P2PKH address and the other is a P2SH address.
Furthermore a locktime has been set. As already discussed, *BIP0068* requires the version to be set to *2* or greater whenever the locktime feature is being used. As we can see, this requirement is met in this transaction. Since the locktime is set to *489733*, the first block which this transaction could be added to is *489734*. In fact, this transaction was included in block *0000000000000000000ec8d98e4ccdf87b35b21a980098d316f166442fe47c81c* at the block height *489831*, which is nearly 100 blocks later than the specified locktime. Furthermore the sequence numbers are set to *4294967294* (0xfffffffe), which indicates that (opt-in) RBF is deactivated and a locktime could be set.
Interestingly, the transaction time and the block time are the same. However, if we check the block time and timestamp in a blockchain explorer like *blockchain.info* we see that they actually differ.
This is the case, because the timestamp of a transaction is not saved in the blockchain data. The timestamps are logged by the node. However, our node was not connected to the network when the transaction has been published, so the first time our node notices this transaction is when parsing this block.
We are mainly focused on privacy aspects in Bitcoin, so we are not going to discuss this simple raw transaction in further detail. However, it is important to see, that there are many different features which can be used in Bitcoin transactions. Some of them are used very rarely. Moreover, there are several ways to correctly sign a Bitcoin transaction on a low-level view.
In the following chapters we will show, that the analysis of Bitcoin transactions can lead us to identify implementations of generic transaction generation. Through this we will be able to identify and break networks that are used to enhance privacy in Bitcoin.

2.2 Fungibility

In an economical sense, a good is fungible if it's interchangeable with other individual goods of the same asset. Typically, government issued currencies and assets like gold are fungible. [28]
For example the same amount of gold with the same weight and purity has normally the same value. However, this does not necessarily apply to Bitcoin.
Through the usage of Bitcoin in criminal activity a *blacklist* of Bitcoin addresses is heavily discussed and has already been implemented by multiple service providers [36]. Since every Bitcoin transaction is publicly accessible, a blacklist can be easily enforced. There are multiple suggestions and services that believe to enhance privacy in Bitcoin and transform it into a fungible asset. We are going to discuss them in the following sections.

2.3 Privacy in Bitcoin

While Bitcoin is today broadly seen as an asset like gold [22], it was created to be the digital version of cash. It should allow people to create digital payments without an intermediary. [37]
Bitcoin is a *untrusted system*, with a public ledger. Every transaction that has been processed by the network is permanently stored in the publicly accessible blockchain. If Bitcoin evolves to a widely adopted digital currency it should also provide privacy to the users. While there is no privacy in Bitcoin transactions, yet still Bitcoin is often described as an anonymous currency:
"***Bitcoin** anonymous digital currency that is not tied to any government [...]*" [24]
Contrariwise to this definition, Bitcoin is not anonymous. However, it it also not fully transparent. Bitcoin is a pseudonymous network [5]. Since public-keys are unique, they are the pseudonyms of their users. But since thousands of public-keys can be generated by a smartphone in seconds, it's typically not possible to identify the owner of a public-key without further information. However, if a public-key can be linked to a specific individual, it is possible to track every transaction this individual received and spent through this public-key. If Bitcoin gets widely adopted as a currency, this obviously would lead to serious privacy issues.
For example in the business environment. Without any enhancement of privacy, it e. g. could be possible to determine the salary of the company's employee. Furthermore, it could also lead to identify when and in which size investments are made by a company. Specific suppliers could be identified as partners and the revenue of a specific product could be publicly tracked.
Generally speaking, companies and individuals could get harmed if all of their financial data are publicly accessible. Since this was already known at the time Bitcoin was created, the Bitcoin whitepaper recommends to use a new address for every transaction [37]. The wallet client should automatically generate a new addresses for every transaction that is going to be received or sent. The amount of a Bitcoin wallet would be the sum of all amounts of every address the wallet manages. This is the standard implementation of Bitcoin wallet software. [9]
In figure 2.7 we can see the simplified transaction flow. For every ouput transaction an unique Bitcoin address is used.

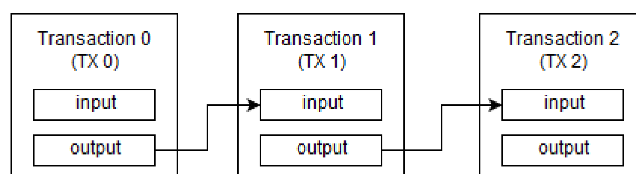

Figure 2.7: Simplified Bitcoin transaction flow

This method was implemented to provide better privacy. In contrast to the model describe above, where all transactions are sent and received by a single address, it definitely enhances the privacy of the Bitcoin ecosystem. However, it's still easy

to spot which public-keys belong to a unique user which we will show through the next examples. In figure 2.8 we are going to show a simplified Bitcoin transaction. None of the addresses used as input will never be used again to receive any UTXO.

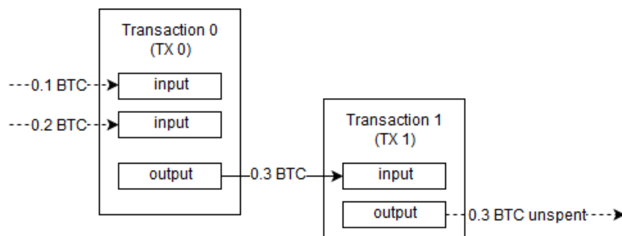

Figure 2.8: Bitcoin transaction without fee

Figure 2.8 is simplified, since the sum of both inputs exactly match the output amount ($0.1+0.2 = 0.3$) and no fee has been applied. It is important to know, that always the whole amount of an input is spent. The sender would always have to pay the full 0.30 BTC, even if he only wants to send 0.25 BTC.

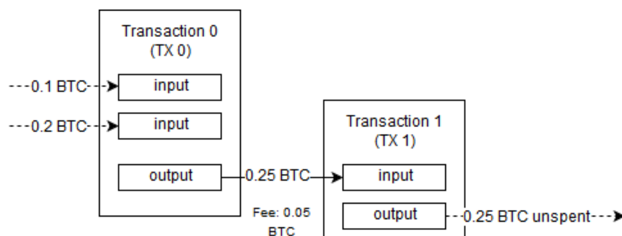

Figure 2.9: Bitcoin tranction with fee

Since we already know that the transaction fee equals the sum of the inputs subtracted from the sum of outputs, the sender would have paid 0.05 BTC fee in this transaction. We can cleary see this in figure 2.09. A more dramatic example would be if the sender has received 1 BTC at his address in a single transaction and wants to send 0.001 BTC. We are going to show this through figure 2.10.

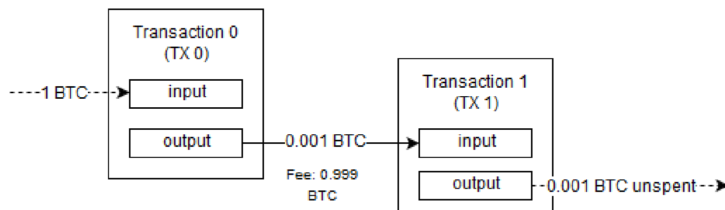

Figure 2.10: Bitcoin tranction with fee

As we can see in figure 2.10, the sender would have paid a fee of 0.999 BTC for

transferring 0.001 BTC. This all happens because in it in Bitcoin always the full UTXO has to be spent.
Typically the fee should not be determined by the value of UTXO, it rather should be dynamically adjusted in consideration of the network traffic.
To solve this problem the Bitcoin protocol uses *change addresses*, often also mentioned as *refund addresses*. Change or refund addresses are Bitcoin addresses which are automatically generated by a wallet software [31]. They receive the change of a sent transaction. Whenever the sent outputs would not exactly match the amount the sender wants to send, the change will be returned to the change address.
For privacy reasons the change address is typically after every transaction newly generated. Since fees of transactions are usually calculated dynamically, most of the transactions make use of change addresses.
The whole process of generating and managing change addresses is handled by the wallet software.

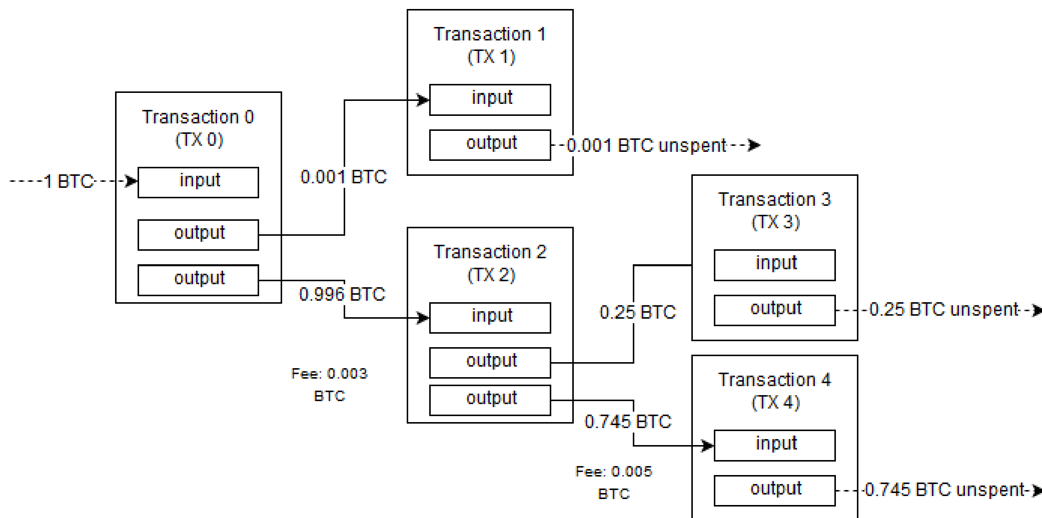

Figure 2.11: Bitcoin transaction with change address

As we can see in figure 2.11, the sender sent 0.001 BTC through *Transaction 0*. He paid a miner fee of 0.003 BTC and received a change of 0.996 BTC at a newly generated change address. Through this address he sent another payment (*TX 2*) of 0.25 BTC, for which he paid a fee of 0.005 BTC. The change of this transaction (0.745 BTC) has been saved in another newly generated address as UTXO and can be spent in further transactions. However, this example is still not quite realistic, since the fees are typically dynamically generated and specified up to 8 decimals.
We saw through figure 2.10 that the use of change addresses is essential to adjust the miner fee in a precise way. However, we will show through our next example how this system corrupts the idea of enhancing the privacy of using newly generated

addresses for every transaction.

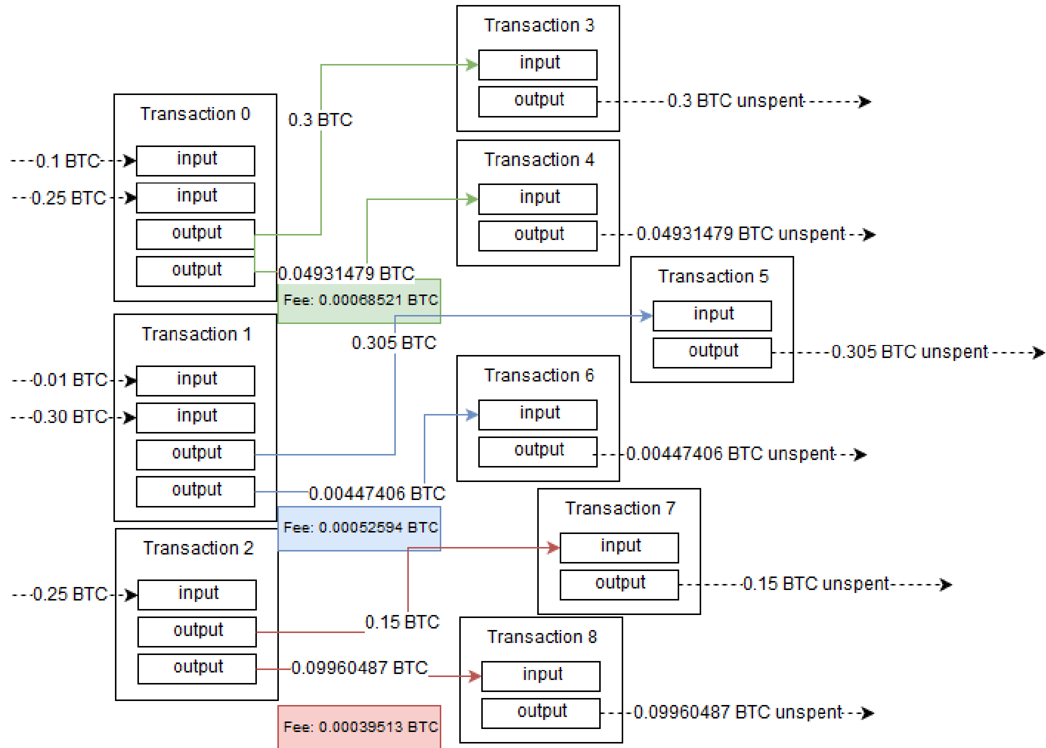

Figure 2.12: Bitcoin transaction with dynamic fee

In figure 2.12, we analyze three transactions send by the same person with a realistic choice of dynamically generated fees. All of the three transactions use unique addresses which were never combined in any transaction before. An attacker could not distinguish whether these addresses are controlled by one person or by multiple people. However, the attacker tries to identify the change addresses of these transactions. At least in cases of *Transaction 0* and *Transaction 1* he could argue that *0.04921479* and *0.00447406* are the outputs send to the change addresses, since the change is often lower than the sent amount whenever multiple inputs of similar size are combined in a transaction. Even in *Transaction 2*, where only one input is given, he could argue that *0.09960487* is sent to the change address, since 0.15 BTC seems to be a more reasonable amount to be sent as a manual payment. It is important to state, that these arguments can only be indicators to distinguish between change and receiver addresses. Yet there is way more information that can be taken in consideration. In most cases transactions are automatically generated. A further analysis of transaction flows, the way of signing a transaction, the way of adding a change address in an implementation and many more factors can be used to distinguish between change and receiver addresses. For simplification we will stick to our basic indicators mentioned above and assume that we are able to distinguish between

change and receiver address. In our practical verification, we are going to show, that our basic indicators are met in most real world cases.

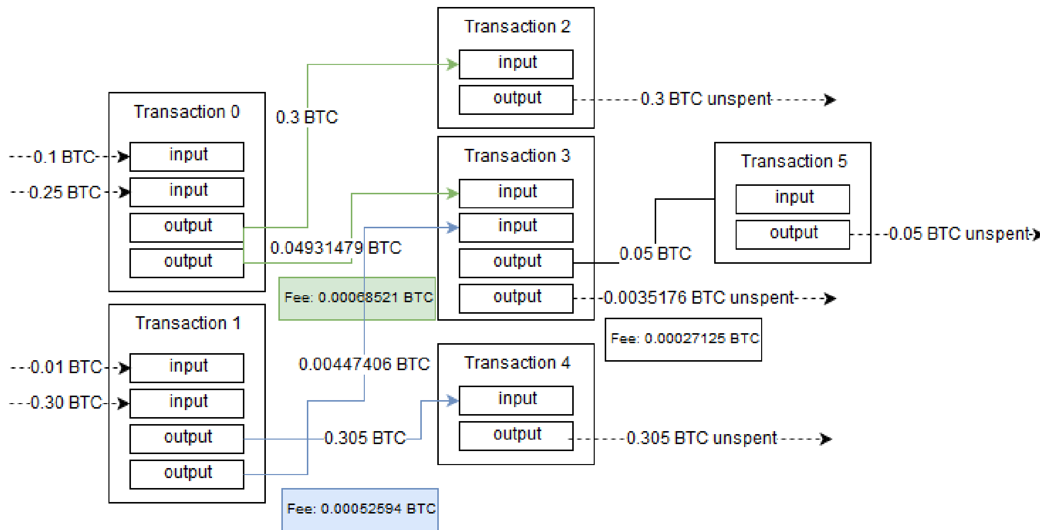

Figure 2.13: Bitcoin transaction with connected change addresses

We can clearly see in figure 2.13, that *Transaction 3* uses two inputs which seem to be changes of previous transactions. Since the sender of this transaction has to have access to the private keys of both change addresses, he is the sender of *Transaction 0*, *Transaction 1* and *Transaction 3*. While this case looks simplified, it is the actual implementation of the standard Bitcoin protocol which is implemented in most Bitcoin wallets [9]. The change addresses are typically automatically combined in the next transaction in a way that the following transaction is cheapest.
While the attacker now knows that each output address of *Transaction 0*, *Transaction 1* and *Transaction 3* is owned by the same person, he does not know if there may be more addresses connected to the owner. However, since the next transaction, which uses UTXO *0.0035176* as input, will probably be connected again with a change addresses, the attacker will gather more possible addresses which are owned by the victim. With the elapse of time the attacker is able to gather more information, which he can use to identify all addresses that are managed through the wallet used by the individual.
The privacy could be enhanced by either not using change addresses or not combining any UTXO. We already discussed that both approaches would be very expensive. A possible solution could be to use different Bitcoin wallets. Each wallet would control multiple addresses. In this case the amount of each wallet could be determined by an attacker, but, as long as the wallets won't get combined, the full amount of all wallets cannot be determined. While this approach could enhance the privacy in some situations, it still got huge drawbacks in practical use, which we are not going to discuss in detail.
We can finally say that the approach Satoshi Nakamoto described in his whitepa-

per enhances the privacy of the Bitcoin network in a coarse way, but still an attacker is able to identify and track a specific user by simple blockchain analysis.

2.4 Mixing techniques

It is broadly known, that the standard implementation, we described in the last sections, only marginally enhances privacy [41]. New methods have been created which should lead to better privacy in Bitcoin. Typically, these methods are summarized as *mixing*. These methods should evolve Bitcoin to a fungible currency.

Figure 2.14: Bitcoin mixing

In figure 2.14 we can see a typicall mixing process, which is normally initiated by the sender. He sends Bitcoins which can be traced back to him (*tainted* coins) into the mixing network and receives anonymized (*untainted*) Bitcoins by the network. An attacker neither should be able to trace the origin of untainted Bitcoins nor should he be able to follow the tainted coins to identify the untainted coins. The tainted and untainted coins should not rely in any connection which could be analyzed by an attacker. In recent years there have been developed multiple mixing methods. We will divide them into three categories.

2.4.1 Decentralized mixing (P2P mixing)

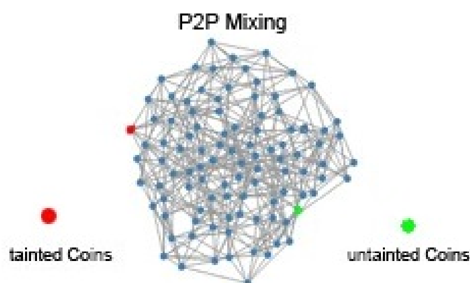

Figure 2.15: P2P mixing

Figure 2.15 shows the structure of a P2P mixing service. Multiple scientific papers like [33] [44] or [26] about algorithms, which could make it possible to anonymously

transfer Bitcoins, have been published. Some of those algorithms have been implemented in different cryptocurrencies by default (e. g. Zerocash [42]), but at the time of writing, none of those approaches have been widely adopted in the Bitcoin network. Unlike Centralized Mixing Services (CMS), P2P mixing has to be implemented in Bitcoin wallet software to be accessible by users.

2.4.2 Centralized Mixing Services (CMS)

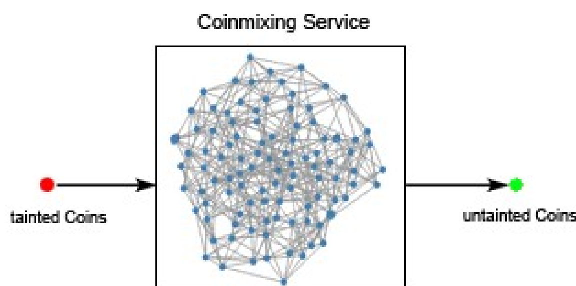

Figure 2.16: Centralized Mixing Service

In figure 2.16 we can see the structure of a centralized mixing service. Centralized Mixing services (CMS) are typically provided through a website.
While P2P mixing is based on no central instance, centralized mixers are typically run by a commercial website provider who advertises that his service is able to anonymize Bitcoin transactions.
The customer specifies addresses where he wants to receive the anonymized Bitcoins. After that, the customer sends Bitcoins to an address which the CMS individually generated for him. When the transaction has been confirmed by the network and an optional specified delay has been waited, the anonymized coins are sent to the customers' addresses. For providing this service, CMS typically charging a fee up to 3% of the initial amount of untainted coins. Often P2P mixing algorithms are internally used by CMS.

2.4.3 Off chain mixing

Every transaction sent over the Bitcoin network is publicly accessible through the Bitcoin blockchain. However, recently there have been made much scientific and economic effort to find solutions of sending and receiving Bitcoin transaction in an untrusted manner without the need to publish them in the blockchain. [38]
The implementation of a layered solution to this problem is called *Lightning Network*. The Lightning Network should lead to nearly instantaneous transactions and could possibly eliminate transaction fees. While the main goal of this approach is to provide a better scalability of Bitcoin, it could also enhance the users' privacy. Since the transaction data are not saved in a publicly accessible blockchain, attackers

cannot perform blockchain analysis. However, since the Lightning Network is still under development and even it's layered structure is not finalized yet we won't discuss it in great detail. [38]
Besides the Lightning Network there are more approaches which should enhance privacy and do not fully rely on the original Bitcoin blockchain. Especially *Tumblebit* [26] and the use of sidechains (*Drivechains*) [17] should be mentioned here.

3 Centralized Mixing Services

In this chapter, we are going to provide general information and attacking possibilities against CMS. Most of CMS advertise, that they are able to anonymize their customer's Bitcoin payments.

3.1 Advantages

While no decentralized mixing technique is widely adopted yet, there exist several CMS which are frequently used. The user does not have to install any software or execute any script to use CMS. Typically no registration is required and the mixing process can be easily started through a publicly accessible website. Furthermore, the customer is often able to change optional settings (e. g. time delay), which should enhance the privacy. Since no specific software is required to use CMS, the services can also be used by customers which use online Bitcoin wallets.

3.2 Disadvantages

The main disadvantage of CMS are, that they are centralized commercially driven services.
Typically, these mixing services can be accessed through a publicly accessible website. None of the service providers is personally known. Since the customer has to send assets in form of Bitcoins to the service, he can easily be defrauded. Furthermore, logs of the mixing process could be stored, which could lead to an exposure of personal data which are not even stored in the blockchain (e. g. IP address, user agent). Some CMS publicly state which internal mixing method they are using, while others don't mention how they are mixing the customer's coins. A fully open source script of a Bitcoin mixing service could not be found.
Unlike P2P mixing protocols, which are typically published through scientific papers, CMS can only be attacked as a black-box. Since all transactions are permanently stored in the Bitcoin blockchain, an implementation bug could easily lead to the deanonymization of every transaction ever processed by the service. Even if a transaction is anonymized through a mixing service, this transaction may be deanonymized in future.
While some of these drawbacks also apply for on-chain P2P mixing algorithms, they are more harmful to CMS, since their implementation cannot be publicly reviewed. Some CMS make use of P2P mixing techniques, however, the imple-

mentation may lead to multiple side-channel vulnerabilities or other bugs, since P2P mixing algorithms are typically not developed to be used as a centralized service.

3.3 Attacker models

We are going to define several attackers, which require different types of resources.

> **A** - An attacker who has access to the Bitcoin network.

> **B** - An attacker who has access to the Bitcoin network and is able to start mixing procedures through the centralized mixing service. He is able to send tainted Bitcoin to the mixing service and receive untainted coins by the mixing service.

> **C** - An Attacker who has access to the Bitcoin network and is able to forge multiple nodes. Based on the P2P network, this usually requires multiple IP addresses (IPv4 and IPv6) and a powerful server.

> **D** - Attacker C, with enough computational power to mine new blocks in reasonable time.

In all mentioned attacking scenarios, we assume that the attacker is able to retrieve publicly accessible statistics which may be published through the centralized mixing service.

3.4 Possible attacks

There are multiple attacks that can be carried out against CMS. Since CMS are typically based on publicly accessible websites and rely on internal P2P mixing algorithms, there are multiple possible layers to attack. The attacker could use web security bugs, internal bugs of the P2P mixing algorithm or general attacks on the Bitcoin network to successfully break the centralized mixing service.

3.4.1 Blockchain analysis

Requires attacker: **A - B**

A blockchain analysis is based on information which can be gathered through the publicly available blockchain [27].

There are different approaches to analyze these data. While blockchain analysis is a powerful attack which can be used for various purposes, it is often mistaken with a taint analysis. Through a blockchain analysis every accessible blockchain data can be used to gather the wanted information, while a taint analysis is only focused on the transaction flow.
In case of identifying specific implementations (e. g. mixing services) in the Bitcoin network, side-channels are important to mention. Through side-channels specific information of transactions (e. g. time, size, way of signing) can be used to identify a specific implementation. When attacking CMS it's often crucial to identify the centralized mixer as a subnetwork in the Bitcoin network.
As we already have mentioned, there are several ways to construct correct Bitcoin transactions. Through the blockchain analysis an attacker is able to analyze the specifics of black-boxed mixing services. His aim is to distinguish whether a transaction is part of the service's subnetwork.
After he achieves this, he tries to deanonymize the transaction. Through a single side-channel, which is able to identify the service's subnetwork, the whole centralized mixing service could be broken. This side-channel could lead to the deanonymization of all transactions the service ever processed.

3.4.1.1 Taint Analysis

Till early 2017 blockchain.info provided a service which visualized the *taint* of an address. The so called *Taint Analysis* evaluates the associations between multiple addresses and shows how strong the links between them are [35]. The analysis was introduced to evaluate how much anonymity a specific mixing service provides. The mixing service should provide *untainted* coins. Which in fact means, that the customer's input addresses should not be connected to his output addresses.
This analysis has often been used to determine how good a mixing service anonymizes transactions [34].
While a taint analysis only takes direct connections between addresses into account, other approaches like sophisticated blockchain analysis or statistical tests are able to use side-channels to find associated addresses which are not directly connected. In this way a blockchain analysis could lead to a deanymization of transaction's addresses which are not tainted.
It is controversial discussed if taint analysis is a suitable tool to determine a transaction's origin since exchanges and other services in the Bitcoin ecosystem create new links between addresses.
The *Taint Analysis* function, introduced by blockchain.info, has recently been removed. At the moment there is no known publicly available tool which is able to perform a taint analysis.

3.4.2 Sybil attack

Requires attacker: **C - D**

Sybil attacks can be used in decentralized networks by forging identities. In case of Bitcoin a sybil attack can be carried out by forging multiple nodes through a single server [35].
On 1st August 2017 Bitcoin cash (BCH) came alive as a fork of the Bitcoin blockchain. On that day the nodes of the Bitcoin network increased from 11.000 to 16.000 in only 12 hours and later dropped back to 11.500. It is believed that attackers tried to execute a sybil attack on the Bitcoin network. However, the motivations of this possible attack are still unclear. [43]
Small sized sybil attacks, where the attacker cannot provide any mining power, do not harm the stability of a stabilized network [21]. An attacker who successfully carried out a sybil attack can be seen as a man-in-the-middle between the network and a client. He could hold back incoming or outgoing transactions. However, whether a sybil attack may have an effect on the network or specific network members is primarily based on the network structure and the implementation of the client software.
Since the attacker is able to control the data which are sent from the client to the network, he is able to log data which could interfere user's privacy. If an attacker is able to spot and control the incoming and outgoing connections of a centralized mixing service, this could compromise the privacy of every customer. Since CMS automatically send and receive transactions, the attacker is able to carry out sophisticated timing attacks. If an attacker is able to mine valid blocks, he could steal the mixer's Bitcoins by creating stale blocks. However, this attack may not be a suitable attack carried out against Bitcoin, since the Bitcoin network provides a comparatively high difficulty and long block generation time. It would be a very costly attacking attempt.
There are two well known countermeasures against sybil attacks. The client could connect to as many nodes as possible or he could add a static connection to at least one well connected node he trusts [21].
A sybil attack needs resources, much time in planning and a precise knowledge of the victim's system and subsystems. While this attack is difficult to carry out and can be very expansive, it is very difficult to be spotted, if it only attempts to attack specific services or network members.

3.4.3 Web security bugs

Requires attacker: **A - B**

CMS are typically accessible though a website. Web security bugs and vulnerabilities could compromise the service. We are going to discuss some vulnerabilities which could lead to deanonymization of customers.

Session hijacking/fixation - Through session hijacking an attacker might be able to gain access to the customer's *Letter of Guarantee*. The letter of guarantee typically provides all necessary information to deanonymize a mixing process.

XSS - Cross-site scripting attacks could be used in various ways to deanonymize customer's transactions. Through website overlays or logging of user information (IP addresses, user agents) and a sybil attack customers could be deanonymized. Furthermore, it may even be possible to steal customer's Bitcoins through changing the coinmixer's Bitcoin address.

SQL and other code injections - May lead to information disclosure e. g. through stored letter of guarantees or could lead to actively storing customer's information. Furthermore, customer's Bitcoins could be stolen by changing the coinmixer's Bitcoin address.

Broken authentication - May lead to information disclosure.

It should be mentioned, that this list is not complete. It is important to know, that based on the attacker model, even bugs which does not seem to be vulnerabilities could be used in combination with other attacking approaches like blockchain analysis or sybil attacks to deanonymize customers.
While in common attacking scenarios it might not be the main goal to gain access to access logs, in case of attacking a centralized mixing service, this could easily lead to the deanonymization of every customer.

3.4.4 DDoS

Requires attacker: **A** and resources to successfully execute a DDoS attack.

DDoS attacks can harm CMS and even be used to compromise the customer's privacy. The simplest case is, when DDoS is being used to block customers of using CMS.
More interesting are cases where through a DDoS attack an attacker is able to gather information about the black-boxed system. Since the security of CMS is often based on time delays, a DDoS attack could influence the mixing process which leaks information to identify customers.
If a node did not specify static connections, a DDoS attack could also be used to successfully carry out a sybil attack. The victim could reconnect to the forged attacker's nodes.

3.4.5 Attacks on the Bitcoin protocol

We described already attacks which primary are based on the Bitcoin network (e. g. sybil attack), however, there are also vulnerabilities and known Bitcoin protocol bugs which can be exploited to attack a mixing service. If the developer of a mixing service is not aware of these bugs, they could lead to deanonymization of the customer's transactions or financial loss.

3.4.5.1 Double spending

Requires attacker: **B**

We discussed already the RBF and other features which allow a sender to update and replace a Bitcoin transaction. These features may lead to vulnerabilities, if CMS sent their untainted coins to the customer without waiting till the customer's transaction have been included in the blockchain. Since the user is able to change the transaction, the mixing service could be sending untainted coins without receiving the user's taint coins.
In general, it should be waited at least for three confirmations, before an incoming transaction is evaluated as confirmed [5].

3.4.5.2 Stale blocks

Requires attacker: **D**

Stale blocks are Bitcoin blocks which fulfill all requirements to be evaluated as a valid Bitcoin block, but they are not part of the main Bitcoin blockchain [9].

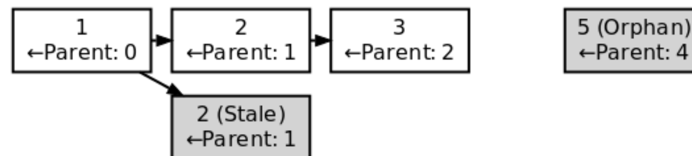

Figure 3.1: Stale and orphan block [9]

In figure 3.1. we can see a stale and an orphan block. Normally stale blocks are created whenever at least two miners were able to mine a block at the same time [25]. While both blocks are valid, only one of them can be added to the blockchain, since the main blockchain does not allow to have multiple blocks at the same block

height. Whenever multiple blocks were mined at the same block height, miners can choose on which block they want to mine their next block. Typically miners choose the block they received first [5]. The block which is not mined on, is the *stale block* which won't be part of the main blockchain.
Since the stale block is not part of the main blockchain, none of the transactions included in this block are confirmed. However, the transactions can still be a part of other blocks in the main blockchain.
A miner could exploit this behavior by willingly creating blocks which are going to be stale blocks later on. The miner would be able to include several transactions in his stale block which will be confirmed through his block, knowing they will be reversed later on. In case of the main blockchain, the transactions have never been made. If a mixing service does not wait for multiple confirmations, it may send the untainted coins to the attacker. While the attacker's input transaction will be ignored, since it is only available in a stale block, the output transaction will be included in the main blockchain.
However, it is hard to willingly create a stale block, which will be accepted by the network but later on ignored. Furthermore, the mining reward will be lost. For successfully carrying out this attack there may be a sybil attack prior to the mining necessary. In this case the attacker controls which blocks the centralized mixing service is able to receive and would be able to disconnect the service from the main Bitcoin network till he mined his own stale block. The same drawbacks we described in the sybil attack apply for this attacking approach.

3.4.5.3 Replay attack on forks

Requires attacker: **B**

When a stale block was created there is typically a wipe out of the short blockchain. However, this is not always true. In case of a planned fork, miners could still be working on the shorter chain. This happened on 01.08.2017 through the Bitcoin cash (BCH) fork. Since both chains are valid, the UTXO which were sent before the chain-split are valid on both chains. If the forked chain does not have any replay protection, it is possible to spent UTXO on the old chain and replay this transaction to the forked chain. And vice versa.
However, CMS are typically only connected to one of these chains. If no replay protection is applied, an attacker is able to replay the transaction, which the CMS sent on one chain, to the other chain. The attacker will receive coins on both chains, since he owns the private key on both chains, but in fact he only sends his coins on one chain to the centralized mixer.
Typically the forked chain should implement a replay-protection.

3.4.5.4 Transaction malleability

Requires attacker: **B**

Through transaction malleability an attacker is able to change a transaction's hash as long as it stays unconfirmed [19]. In implementations where transactions are tracked by their hashes, this could lead to financial loss, since the transaction with the changed hash cannot be found anymore by the implementation. Most probably the transaction will still be confirmed by the Bitcoin network. In the case of CMS, the attacker could change the outgoing transaction and the mixer might not find it anymore and automatically resend it. The mixer would spend two output transactions, while only receiving a single input transaction.
While this attack requires a rare setting, it may have led to a loss of more than 302.000 Bitcoins [19].
Recently this bug has been fixed through the *Segregated Witness* (SegWit) soft fork [16]. Since the fix has been activated as a soft fork, the sender of a transaction has to specify if he wants to enable this feature in his transactions. At the moment only every 10th transaction uses SegWit [10]. Every transaction, which does not make use of the SegWit feature is still malleable.

3.4.6 Conclusion

As we can see, there are multiple attacking possibilities against CMS. The implementation has to be secured against Bitcoin network weaknesses and web security vulnerabilities. Furthermore, a secure mixing algorithm has to be implemented. Even if all of these layers are implemented in a secure fashion, they could lead to side-channels when they are combined. A centralized mixing service should be secured against all of these vulnerabilities. It should be able to automatically process transactions, without leaking information about the mixing process. An attacker should not be able to differentiate between transactions, which are connected to the mixing service and other transactions found in the blockchain.
A general drawback of CMS is, that their service is commercially driven and not open source. The customer has to trust the service provider.

4 Attack on coinmixer.se

With a mixing volume of around 120 Bitcoins per week, coinmixer.se is probably one of the most frequently used centralized Bitcoin mixing services available [6].
In this chapter we are going to implement an attack on this service. Our aim is to create a tool, which allows us to deanonymize transactions which priorly were anonymized by coinmixer.se.
Coinmixer.se publishes every week the amount of mixed Bitcoins and the number of performed anonymizations on their website.

4.1 Functionality of coinmixer.se

Figure 4.1: Funcionality of coinmixer.se [6]

Coinmixer.se works in a similar way as typical CMS we described in Centralized Mixing Services (CMS). The customer visits coinmixer.se and specifies the amount of untainted coins he wants to receive. He also specifies the forward address where the untainted Bitcoins should be sent to. He is able to specify multiple addresses (see Optional setting: Multiple addresses) and a time delay (see Optional setting: Time delay) as optional settings. After that, coinmixer.se generates an unique input address and calculates the amount of tainted coins the customer has to pay. When the customer has sent the required amount to the coinmixer's input address, three confirmations are awaited.
After three confirmations are received through the Bitcoin network, and a possible optional delay was waited, the untainted coins are sent to the customer's forward address(es) which previously has/have been specified.

4.1.1 Optional setting: Multiple addresses

Coinmixer.se allows the customer to specify up to five forward addresses.
The amount of untainted Bitcoins which should be received through a specified address can be chosen independently, but has to be at least 0.001 BTC and cannot be greater than 5 BTC.
Every forward address has to be unique.

Even though the customer is able to specify multiple forward addresses, he will always have to pay the untainted Bitcoins to a single address controlled by coinmixer.se.

4.1.2 Optional setting: Time delay

The customer is able to specify a time delay (one hour intervals), which coinmixer.se will wait before it sends the untainted Bitcoins to the specified forward address(es). The time delay can be set to a maximum of 120 hours.
For every forward address a time delay can be specified separately. Per default the time delay is set to one hour. However, the customer is also able to set the delay to zero hours. In this case the output transaction will be sent as soon as the input transaction received three confirmations.

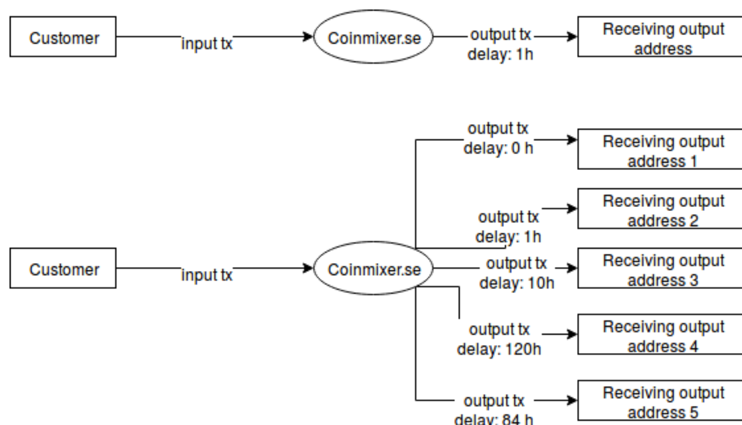

Figure 4.2: The default case of a mixing process and a case which makes use of the optional settings

4.1.3 Mixing fee

Coinmixer.se charges a fee from 1% to 3% of the initial Bitcoin amount the customer wants to anonymize. For every forward address additional 0.0007 BTC are charged. [6]
Based on the fee, the amount a customer has to send should be calculated as follows:

$$BTC_{tainted} = (BTC_{untainted} * fee) + forwards * 0.0007\ BTC,$$

where $BTC_{tainted}$ is the amount the customer has to pay, $BTC_{untainted}$ is the amount the customer receives, fee is between 1.001 and 1.003, $forwards$ is the number of specified output addresses.
However, the last three to four decimals of this amount are changed to not predictable numbers. They may have been randomized. The actual amount a customer has to pay can be illustrated through the following example:
The customer wants to anonymize 1 BTC and did not change any optional settings

(1h delay, 1 forward address).
If coinmixer.se automatically chooses a fee of 1.62 % he would need to pay

$$(1\ BTC * 1.0162) + 1 * 0.0007\ BTC = 1.01690000\ BTC.$$

However, since the last four decimals are not based on the calculation shown above, the customer has to pay a value between 1.01690001 BTC and 1.01699999 BTC. The exact amount which the customer has to pay is most likely randomly chosen in this range.
It is important to state, that the mentioned fees could change. In the last few months, the address fee has changed multiple times in a range from 0.0005 to up to 0.0007. It changed in 0.0001 steps. Most likely a manual fee adjustment is applied whenever the average Bitcoin network fees heavily change.

4.2 Attacker Model

Our attacker is able to access the Bitcoin blockchain. He knows the input transaction which the customer has sent to coinmixer.se and he knows, which of the transaction's output addresses is controlled by the coinmixer. The attacker's aim is to identify coinmixer.se's output transactions.
Furthermore, we are going to describe different attacking scenarios. In some scenarios the attacker knows information about the set optional settings (maximum time delay, maximum number of forwards) while in others these information are unknown to him. We are going to compare the results of these different scenarios.
Mixing statistics, published on coinmixer.se, can be used by the attacker.

4.3 Attacking Method

As we already described in Possible attacks there are many different attacking possibilities which could lead to breaking a centralized mixing service. In our attack we are going to focus on simple blockchain analysis. While our aim is to break a single instance of a centralized Bitcoin mixer, we want to show that there are general problems in using centralized mixing services. Our aim is to successfully implement an attack, which could also be modified to break other CMS. We are going to focus on implementing a Proof of Concept, which may not break the whole implementation in all of it's optional settings, but we are going to show how it could be done in further steps.

4.3.1 Steps to break coinmixer.se

We are not interested in the internal mixing algorithm of coinmixer. While we are going to address it in the crawling process, we try not to make any use of it in the process of transaction deanonymization. Coinmixer.se weekly publishes the amount

of mixing processes it has performed in the last week. As the time of writing the service processed only around 1300 mixings in the last week (2017-10-20 00:00:00 UTC - 2017-10-26 23:59:59 UTC) [6].
Since the number of processed mixings is small, our main attack will focus on mapping customer's input transactions to coinmixer's output transactions based on the amount of sent untainted Bitcoins.
To achieve this, we need to accomplish three steps:

1. We need to identify the coinmixer.se network in the blockchain. We need to be able to distinguish whether a transaction is part of the coinmixer.se network. Furthermore, we have to be able to identify which of these transactions were sent by customers as pay-in transactions and which are pay-out transactions sent by the coinmixer.

2. We need to be able to implement a crawler which stores the coinmixer's network in a database.

3. We need to identify which input transaction can be mapped to which output address. Vice versa.

It is important to state, that the deanonymization step is not based on a taint analysis (see Taint Analysis). Since we do not analyze the internal mixing structure in our deanonymization step, the results are highly correlated to to the mixing behavior of other customers.

4.4 Identifying coinmixer.se's network

The first important step is to identify the mixing network within the blockchain data. As we are not interested in the internal mixing process, we only need to identify the customer's input transactions and the coinmixer's output transactions. However, it might be hard to filter these transactions. For the following calculations we are going to ignore possible internal mixing transactions.
Based on the published data, we assume that coinmixer.se processes around 1300 mixing processes per week [6]. There should be around 1300 input transactions sent by customers. Since every customer is able to specify up to five forward addresses, there will be a maximum of $1300 * 5 = 6500$ weekly output transactions sent by the coinmixer. If we want to identify the coinmixer network, we need to identify a maximum of $6500 + 1300 = 7800$ transactions in the weekly produced blockchain data. To get a feeling how hard it is to identify these 7800 transactions without any further information, it is necessary to know how many transactions the Bitcoin network is able to process per week.
In general the Bitcoin network is able to process 7 TPS (transactions per second)

[38]. When the SegWit feature is widely adopted the maximum number of TPS should increase [38].
With a theoretically maximum of 7 TPS we achieve a weekly maximum of 4.233.600 transactions. This number of transaction would be sent in 1008 blocks with a block generation time of 10 minutes.
However, 4.233.600 transactions per week is the theoretically maximum. In last week (2017-10-20 00:00:00 UTC - 2017-10-26 23:59:59 UTC) only around 2.182.236 transactions have been processed through the Bitcoin network, while the average blocksize was even bigger than 1 MB [1]. Through the SegWit feature miners are able to create blocks which are bigger than 1 MB. [38]
Either way, our implementation would need to identify a maximum of 7800 transactions in more than 2 million transactions.

4.4.1 Characteristics of customer's input transactions

We need to identify the customer's input transaction and the coinmixer's output transaction(s). The identification of the customer's input transaction seem to be not trivial, since input transactions are manually sent by customers. The customer is able to choose the timing, the fee and the client through which he sends his transaction. He is also able to make use of various features like SegWit or RBF in his transaction.
However, the amount, which the costumer has to send to coinmixer.se, is predefined by coinmixer.se. As we already described in Mixing fee, the last three to four decimals of this amount were not predictable. We noticed this abnormality, since the predefined amount the customer has to send to coinmixer.se does not add up with our manual calculation. In more than 30 test cases which we created, the input transaction's amount specified by the coinmixer never ended with a zero.
In all cases the customer would have to send a transaction which is specified to exactly eight decimals. We assume that coinmixer.se behaves like this for every customer. While the precision of a transaction's amount may be an indicator for customer's input transactions, this indicator may not be good enough to filter 7800 transactions out of more than 2 million Bitcoin transactions. However, this indicator and the knowledge that customer's transactions are typically sent through standard Bitcoin clients might help us later on. While services, which perform Bitcoin transactions, often send generic Bitcoin transactions, manual transactions sent by individuals are typically more varying.

4.4.2 Characteristics of coinmixer's output transactions

While input transactions are manually sent through the customer, the coinmixer's output transactions are automatically sent [6].
Since the coinmixer's output transactions are automatically generated, their internal structure most probably will be similar. To be able to differentiate coinmixer's output transactions from other Bitcoin transactions, we need to identify these char-

acteristics. To receive transactions sent by coinmixer.se, we used the service several times with different settings set.
In total we received and analyzed more than 20 coinmixer.se transactions. All of the received transactions had in common, that a locktime was specified and transaction version *2* was used. When analyzing we also noticed that the sequence number of every transactions sent by the coinmixer was set to *4294967294*. We also realized that every transaction sent by the coinmixer used a static transaction's *fee per byte* with a variance of ± 1.
It could be argued that the provider is able to set a specific transaction's *fee per byte*, which will be used until it is manually changed. Based on this assumption, all coinmixer transactions sent at the same time should have the same fee set. The fee can be divided into fee partitions. Since the decimals did not seem to be predictable, we will only focus on the predecimals.

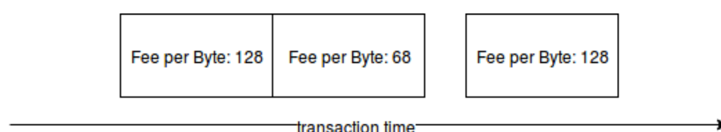

Figure 4.3: Fee partition

For easier understanding we are using the term "fee" when we refer to "fee per byte", knowing that a transaction's fee is in fact a different value.

While there are probably way more characteristics of coinmixer's transactions, we will focus on the indicators we mentioned above.

Table 4.4: *Strong* and *good* indicators to spot coinmixer's transactions

Indicator	**Credibility**
version $= 2$	strong indicator
sequence number $= 4294967294$	strong indicator
locktime > 0	strong indicator
transaction in fee partition	good indicator

We now need to determine if these indicators can be used to filter coinmixer's transactions from random Bitcoin transactions. To analyze how good these indicators are, we analyzed 1036 Bitcoin blocks (around 1 week), starting from block height 494120 (2017-11-13 00:03:34 UTC) and ending at block height 495154 (2017-11-19 23:47:43 UTC). Based on the published mixing static of coinmixer.se [8], the service anonymized 1365 transactions in that same week, which would result to be at least 1365 and at most $1365 * 5 = 6825$ coinmixer's output transactions.
We ignored possible internal mixing transactions.
Through an implemented script (see Python Code), we were able to identify 2.150.927 confirmed transactions between block heights 494120 and 495154. Erroneous transactions (e. g. double spends) and coinbase transactions have been

removed.

Based on our test set we obtained the following results:

463.090 of 2.150.927 transactions used version *2* (21,53 %).

376.567 of 2.150.927 transactions used version *2* and specified a locktime (17,51 %).

450.810 of 2.150.927 transactions set the sequence numbers of every inputs to *4294967294* (20,96 %).

362.709 of 2.150.927 transactions used version *2*, specified a locktime and set the sequence numbers of every input to *4294967294* (16,86 %)

As we can see, we were able to filter 362.709 (16,86 %) out of 2.150.927 transactions, which fulfill the mentioned indicators to be coinmixer.se transactions.
Still, the indicators do not seem to filter enough transactions, since, based on our assumption, a maximum of 6825 transactions could be sent by the service. Since every transaction we received from coinmixer.se met the indicators (version, locktime, sequence number), we will refer to them as *strong indicators*.

However, the fee indicator might be an even better way to determine coinmixer's output transactions, since Bitcoin clients typically use a dynamically fee adjustment. [5]
Through dynamical fee adjustment, the client calculates the fee based on the Bitcoin network traffic, while coinmixer's fees seem to be fixed and only adjusted in a big scale. While the analysis of version, sequence and locktime is pretty easy to accomplish, for analyzing the fee it must be taken into account that the fee might change in future.
We have to reduce the block range to be able to obtain reliable results, since we have to be sure that the fee does not change within our testing time frame.
We received 14 transactions in the time range between 2017-09-26 00:58:49 UTC and 2017-09-28 01:37:46.

Table 4.5: Transactions received from coinmixer.se between 2017-09-26 00:58:49 UTC and 2017-09-28 01:37:46

Transaction hash	**Time (UTC)**	**Fee (sat/Byte)**
8b6211ac88f1e149189c5c4ddddd0190f8...	2017-09-26 00:58:49	123.119
378ecac1ca091dbbd22d577b61feea2f15...	2017-09-26 02:21:44	123.78
a14ede087bc7284d4258a6022f141d87fa...	2017-09-26 03:19:00	123.445
534b293cae650114e8ab9e0aa6a5385df7...	2017-09-26 08:17:34	123.649
85320bcd73368a0cfa11ea8c7c05fad624...	2017-09-26 10:18:19	123.088
0e43ab54b65ed2bce19c0dd4a90ee7b6fc...	2017-09-26 18:20:28	123.649
ae429009d9dfb44c26568f066c85378997...	2017-09-26 18:43:41	123.08
f9335402ff634502e843ef5645eee5f190...	2017-09-26 18:55:18	123.08
c807cb9ecbbd757d13d84a0e7c8f33a9b3...	2017-09-26 20:12:31	123.075
0411d53d67a76defefac7f7f44eca365e2...	2017-09-26 20:13:19	123.075
69b194882d74a1091e23ad0cbf253c6e39...	2017-09-26 20:59:10	123.618
59810e31b7c73a69b385cf533be3166336...	2017-09-26 21:12:42	123.071
9a602394119532c0f019b092913fa83bad...	2017-09-27 01:34:13	123.044
dfb90f81135a363897126893b34cb8d736...	2017-09-27 03:32:39	123.587
0616d01f2f84d43cee37a986d3da6ed3f4...	2017-09-28 01:37:46	123.449

As we can see in table 4.5, all transactions we received in that time frame have been sent with the same fee. The first transaction we received at 2017-09-26 00:58:49 UTC was included in the Bitcoin block with height 486977. The last one we received at 2017-09-28 01:37:46 UTC was included in the block with block height 487265. We assume that in this period of time every transaction sent by coinmixer.se to customers has a fixed set fee of 123 sat/Byte (±1).
531.558 transactions have been sent between block height 486977 and 487265. Similar to our previous analysis, 17,19 % (91.382) of 531.558 transactions fulfilled the version, sequence and locktime indicators. However, when we applied the fee indicator and filtered the transactions to show every transaction which has a fee set between 122 sat/Byte and 124 sat/Byte, we received a result set of 2.839 (0,53 %) from 531.558 transactions. When we tightened up the fee indicator to only show results where the fee is set to 123 sat/Byte, it results in a set of 1057 (0,19 %) transactions.
Even though all of the transactions we received by coinmixer.se had the fee set to 123 sat/Byte, we realized in further analysis that in some settings the fee was off by one sat/Byte. So we decided to stick with a variance of ±1 for our implementation.
When we apply all four indicators (version, locktime, sequence number, fee) to the gathered transactions between block heights 486977 and 487265, we received a filtered output of 2.839 transactions. Only 0.53 % of all transactions send at the time fulfilled the mentioned indicators.
While the fee indicator seems to be good for filtering purpose, it should not be used as a *strong indicator*, since the specified fee is able to change.
It should be stated, that we only used simple characteristics as indicators, which can easily be spotted through a high-level comparison of standard Bitcoin transactions sent through a Bitcoin client and coinmixer's output transactions. On a low-level

view of Bitcoin transactions probably more characteristics could be spotted. This could lead to even more accurate results.

4.4.3 Identifying customer's and coinmixer's transactions

In the last two subsections we analyzed additional features of transactions which could lead us to identify transactions which are connected to the coinmixer's network. Based on these indicators full Bitcoin blocks could easily be analyzed, which we will refer to as *blockwise crawling*. However, these results could be improved, since we did not take into account any transaction's characteristics regarding the number of input/output addresses, the sent Bitcoin amount or general transaction flow. Many of these features have to be analyzed dynamically, since they need to be analyzed in context with other transactions and addresses. Through our static filtering process even transactions which cannot be coinmixer's output transactions would not be filtered as long as the described indicators are met.
An example would be a transaction with all outputs under 0.001 BTC or over 5 BTC. Since it is not possible to specify an output amount less than 0.001 BTC or greater than 5 BTC at coinmixer.se, this transaction could not be a possible coinmixer's output transaction. An improved way of filtering would be to check whether the transaction's output, which is sent to the customer, is located within this range. But to analyze how many Bitcoins are sent to the customer, we first have to distinguish which of the output's addresses is the change and which the customer address. This typically requires the analysis of multiple addresses and transaction flows.
For a further analysis, we are now going to take the transaction flow into account. The indicators mentioned above have still to be met.
As we notice, every transaction, which we received by the coinmixer, has exactly two outputs.
When we follow the output address which is not controlled by us, we see that it is always used in exactly two transactions. Firstly in the transaction, where also our address is used as an output address, and afterwards in a transaction which uses the change received through the first transaction. No further transaction is sent before or after these two transactions.
When we analyzed the second transaction, we recognized that all indicators mentioned above (version, sequence number, locktime, fee) are met. The second transaction also uses exactly two outputs. Since all indicators are met for the second transaction, we assume, that this is also a coinmixer transaction. It follows that the second address in the first transaction is a coinmixer's change address. The analyzed transaction flow of coinmixer's transactions can be seen in figure 4.6

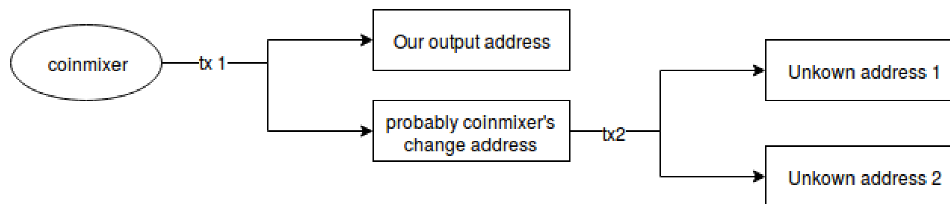

Figure 4.6: Analyzed coinmixer's output flow

When we analyzed the sent coinmixer transactions, we were able to distinguish customer's and coinmixer's transactions based on the characteristics we described in the last subsections. Furthermore, the customer often only specified up to four decimals and uses common values. We define a value to be *uncommon*, if it is specified to more than four decimals (e. g. 0.9286472 BTC).
It seems like the coinmixer's network sends output transactions to a customer, and receives the change on a change address. After that, the change, sometimes combined with other changes, is sent to a next costumer. The change is again saved on a new change address and reused for a next customer. And so on.
Also we noticed that the costumer's outputs are sometime unspent, while the change addresses in all of our analyzed transactions were spent. Customer's addresses could also be differentiated from coinmixer's by the previous and the following transaction's flow.
Some customers addresses received and sent multiple transactions, while coinmixer addresses only received a single transaction and sent a single transaction.
Till now we spotted several indicators to differentiate between customer and change addresses. They can be found in table 4.7.

Table 4.7: Indicators to distinguish between customer's and coinmixer's address

Indicator	**Indicates**	**Credibility**
Sent multiple transactions	customer address	strong indicator
Following transaction does not fulfill all strong indicators	customer address	strong indicator
Following transaction fulfills fee indicator	coinmixer address	good indicator
Received input less than 0.001 BTC or greater than 5 BTC	customer address	strong indicator
Transaction output is unspent	customer address	medium indicator
Received a common value	customer address	low indicator

Based on these indicators we were able to follow the chain of change addresses, which can be seen in figure 4.8.

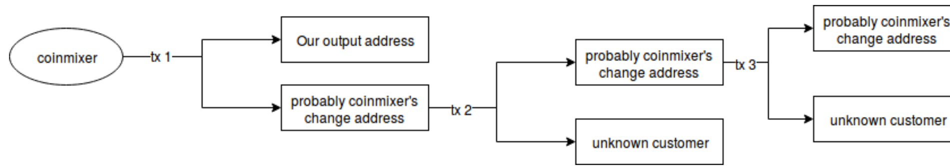

Figure 4.8: Analyzed chain of change addresses

Through the analysis of the transaction flow we were able to identify the internal mixing process. While we were able to identify possible customer's and internal addresses in a forward manner, it may be also interesting to take a look back. As we already know, sometimes change addresses are combined. This could provide us with more information about the network, since till now we only analyzed a single chain of change addresses.

When we analyzed the input addresses, we recognized some addresses which are probably used as cash-in addresses for customers and others that are used for output transactions. The cash-in addresses could be identified, since they are sent through the customer and typically did not fulfill the mentioned indicators, while the input transactions which are used for output transactions do meet them.

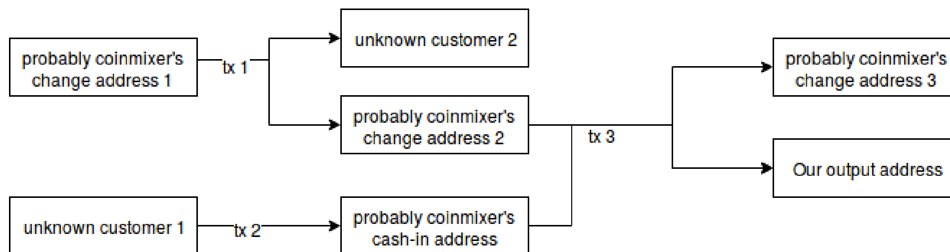

Figure 4.9: Analyzed full coinmixer network

As we can see in figure 4.9, we supposedly identified a method to spot the coinmixer.se network. However, we didn't confirm it yet. A way to verify our assumptions would be to create the network based on a given input transaction and check if another unique input transaction we made after that can be found in the created network.

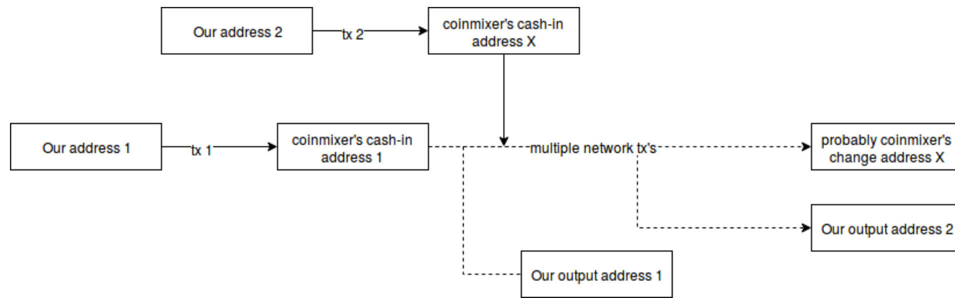

Figure 4.10: Confirm coinmixer network

In figure 4.10 we perform two unique mixing processes to confirm the constructed coinmixer network. We receive through *Our output address 1* the untainted coins for the tainted coins of *Our address 1*. To confirm the network, we did start a second mixing process. Based on the second output address and the known indicators, we reconstructed the coinmixer network.
If our first transaction can be found in this network, we assume that we were able to reconstruct the correct coinmixer network.

While we were able to describe the method to reconstruct the coinmixer.se's network based on the spotted indicators, we want this automatically to be done. To achieve this, we are going to implement a crawler which is able to reconstruct the network based on a given coinmixer's output transaction.
Based on the described method above, we are going to verify in chapter Results that our crawler is able to create the correct coinmixer.se network.

4.5 Crawler

As we already described in the previous section, the crawler should be able to create the coinmixer network based on a given coinmixer's output transaction. To implement this we are going to implement two different ways of crawling. The *forward crawling*, which takes a coinmixer's output transaction and follows the change addresses till it reaches the end of the coinmixer network and the *backward crawling*, which also takes a coinmixer's output transaction but analyzes the input addresses to find previous transactions and addresses which are part of the coinmixer's network.
The process of creating the whole network should be done as we can see in figure 4.11.

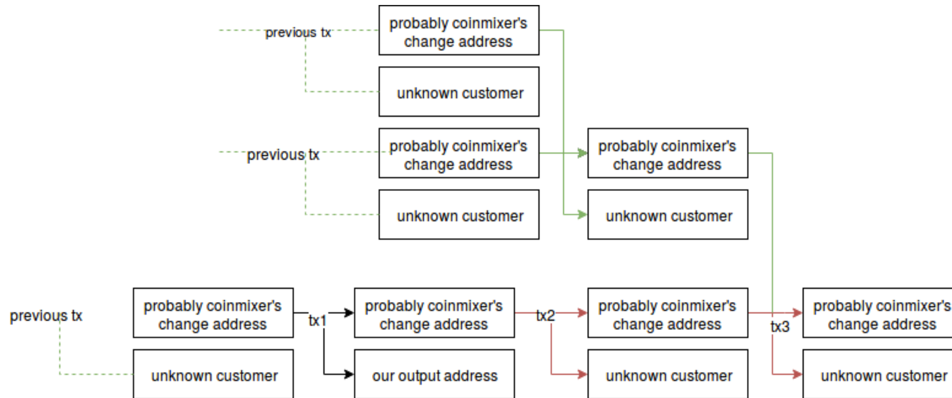

Figure 4.11: Forward (red) and backward (green) crawling process

Through the forward crawling only a single chain (red) of transactions will be found. When the endpoint of this chain is reached the crawler is going to stop. The user should specify the last transaction (tx 3) as the starting point for the backward crawling process.
The green colored transactions can be found through this approach.
Through this approach nearly all transactions which belong to the mixer should be found.
But transactions, which were entirely spent before and don't have any connection with an address, which can be found in the crawled network, cannot be found. However, this should be a rare situation, since the output has to exactly match the customers specified output amount.
Also transactions which have not been connected to a change address chain yet, cannot be found through this approach. But the chains are most probably getting connected through further mixing processes.

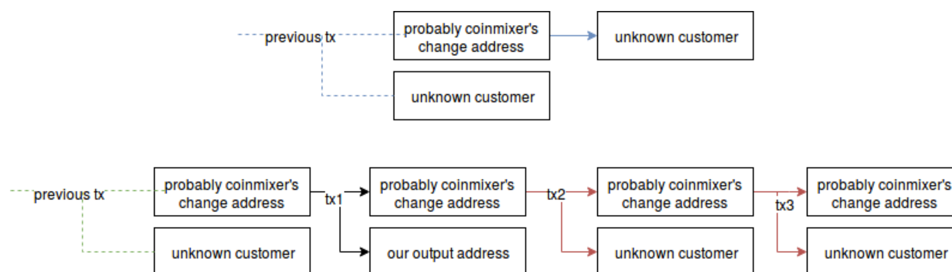

Figure 4.12: Transaction chain which can't be found by forward/backward crawling (blue)

As we can see in figure 4.12, some transactions (blue) might not be found through a forward/backward crawling process. However, they can be found through a *blockwise crawling* process.

It should be noted, that we did not specify any assumptions yet, how the service is receiving the mixing fee. Based on the described network it could either be that some of the output addresses are controlled by the coinmixer or a change address chain leads to an address which is manually controlled by the coinmixing service. We did not further examined these assumptions.

4.5.1 Gathering blockchain data

To analyze Bitcoin transactions it is necessary to parse the blockchain data. There are different approaches to retrieve blockchain data. A complex way would be to connect to the Bitcoin network and download the whole binary blockchain data (143 GB [2]). After that the whole blockchain data can be parsed. A running node would be necessary to receive new Bitcoin blocks. While this is a complex method and may take some time to be implemented, it is entirely based on the Bitcoin network and no external API is required. If an ongoing network analysis should be achieved, this might be the most efficient and reliable approach.
However, if only a specific time frame of blockchain data should be analyzed, external APIs are an easier way to go. It should also be known, that there are also solutions of receiving specific blockchain data through the Bitcoin network, but since we won't use them, we are not going to discuss them in further detail.
We are going to focus on the blockchain.info API. Through the blockchain.info API it is possible to receive blockchain data JSON-formatted. Through this API it is even possible to gather specific transaction and address data without parsing a whole block. However, there is a request limit and there might be a trust issue, since the received data could be flawed. To simplify our implementation, we will use the blockchain.info API to receive the necessary blockchain data. We trust, that the received parsed blockchain data are correct. We are not going to confirm signatures or other cryptographic details of the received data.

4.5.2 Data structure

To decide which data structure should be used by our implementation to store the blockchain data, several options should be discussed. The Bitcoin network is able to process around 2 million transactions per week and might even process way more transactions in future. [38]
We should implement a database which is able to store and access the data in an easy and fast way.
Since our purpose is not to provide a schema-free database we will stick to a MySQL database.
Our database structure can be found in Database structure.
For performance reasons we used in most cases fixed column lengths. We moved most columns which are varying in size (e. g. list of input/output addresses) to separated tables. We can clearly see this at the *address_and_value_mapping*

table. While most of the transaction data are stored in the *transaction_data* table, we stored transaction data that belong to addresses in separate tables (*transaction_addresses*, *transaction_values*). We chose this way to store address data, since transactions vary in the number of input and output addresses and for most of our queries, the specific address data are not important.
This way of storing the transaction data should enhance the accessing performance. Besides the mapping, we created two different tables to store transaction data. Every transaction that's processed through our script will either be stored in *transactions_size_normal* or in *transaction_size_big*. Transactions which exceed a length of 40.000 characters (JSON-formatted) will be stored in *transactions_size_big*. Based on the average Bitcoin transaction size and the overhead produced through the blockchain.info API we have chosen this size. Most transactions should be stored to *transactions_size_normal*, since *transactions_size_big* uses the *mediumtext*-type to store data.
We chose to store every processed transaction, otherwise the crawling processes would need to request transactions which were already processed before. Since the blockchain.info API blocks IP addresses after too many requests, our aim is to send as less API requests as possible.
For performance reasons we divided received Bitcoin transactions in two size categories Database structure.
The column indexing has been chosen based on the implemented MySQL statements.

4.5.3 Forward crawling

As we already described, the starting point of the crawling processes has to be a coinmixer's output transaction provided by the user. Based on the known indicators, the forward crawling process should be able to distinguish which of the transaction's outputs is the change address and which belongs to the customer. The change address will be used as an input for further coinmixer transactions. The crawling process should be executed till the endpoint of the change address chain is reached or the user interrupts the script.

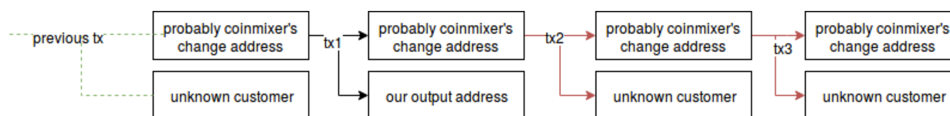

Figure 4.13: Forward crawling

We created a *network* class which manages the crawling process. If a forward crawling process is started, the transaction's hash of the coinmixer's transaction has to be specified. The crawling process will start by recursively searching whether the transaction has already been analyzed before. If so, the endpoint of the last crawling process is loaded through the *cm_log_check* function and replaces the

input transaction specified by the user. The crawler won't analyze a transaction twice.
After the starting point has been identified, the crawler checks whether all *strong indicators* are met. Since the forward crawling process follows the change address, all strong indicators have to be met. If this check fails, most probably the user's specified transaction has not been sent by the coinmixer. The erroneous transaction will be saved. The crawling process is aborted.
If the input transaction/last endpoint is a valid coinmixer transaction, it is stored in the database and the main forward crawling process is able to begin.
Now the output addresses of the transaction are going to be checked. Based on the indicators, it should be checked which of the addresses is the customer address and which is the coinmixer's change address.
The first indicator, which the crawler checks, is the number of sent transactions. As we already described, a coinmixer's change address should only send one transaction.
While it is a strong indicator for being a customer's address, if through an address multiple transactions have been sent, it can also be an indicator for being a customer's address if the transaction's output, which is being checked, is still unspent. This is the case, because most of the coinmixer's addresses will be used in further mixing processes. However, this is only a low indicator, since the last address of the coinmixer network will definitely be able to spend UTXO.
While strong indicators have to be fulfilled for being a coinmixer transaction, the indicators we mainly introduced through the Identifying customer's and coinmixer's transactions subsection should not only be relied on.
We implemented a counting system which helps us to distinguish between a customer's address and a coinmixer's change address. We specified a value for each indicator which will be added to a counter if it's met. After both addresses have been checked, the crawler will evaluate which address has the highest counter. The address with the highest counter fulfills more indicators and is most probably the change address.

Table 4.14: Forward crawling counter

Indicator	Add to counter	Description
More than one tx sent	-1	No further check applied
Version of next tx =! 2	-1	No further check applied
Sequence of next tx != 4294967294	-1	No further check applied
Locktime < 1	-1	No further check applied
UTXO available	0	Probably a customer address
No UTXO available	1	Probably a coinmixer address
Received an uncommon value	2	Probably a coinmixer address
Tx fee based on partitions correct	3	Probably a coinmixer address
Tx fee is located within a partition	1	Probably a coinmixer address

As we can see in table 4.14, there might be a situation where both addresses result in the same counter. However, this situation should occur very rarely and we did not experienced it in any testing scenario. More about this situation we described in Future Work. An *uncommon value* is a Bitcoin amount which is specified to at least five decimals (e.g. 0.98263890 BTC).
We are going to describe the fee indicator in a little more detail. As we mentioned, the fee of the coinmixer's transactions are most probably static. But, they are able to change manually. Since they are able to change, it forms the fee indicator only in some cases to a good indicator for identify a coinmixer's transaction.
As we can see in figure 4.15, the set fee of coinmixer transactions can most probably be divided into static partitions.

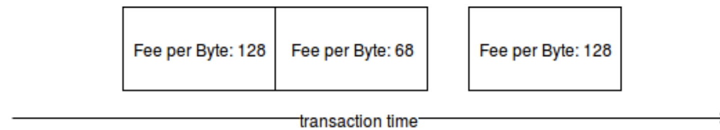

Figure 4.15: Fee partition

Since we are able to start the crawling process with any coinmixer transaction, the transaction is able to lay in different spots of these fee partitions.

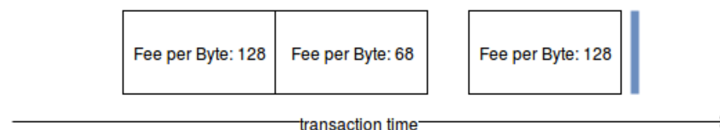

Figure 4.16: Transaction newer than every fee partition

In figure 4.16 we can see a transaction which is located after the newest fee partition. This is the case if the processed transaction is the newest coinmixer transaction the crawler has ever seen. While the fee indicator is still a reliable indicator, it might be wrong, since a new fee partition could be created. The counter is increased by 3 as long as the fee of the transaction is set correct.

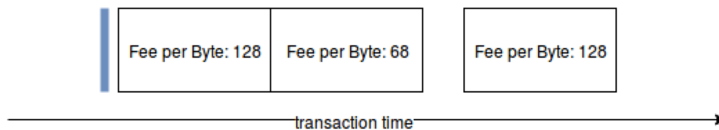

Figure 4.17: Transaction older than every fee partition

As we can see in figure 4,17, it is also possible that the processed transaction is older than every transaction the crawler ever has seen. This situation is handled similar as the situation described above. The counter is increased by 3.

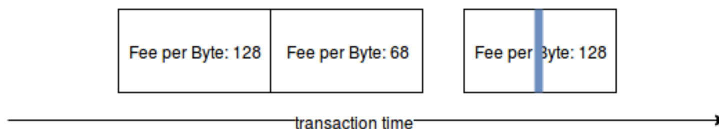

Figure 4.18: Transaction located within a fee partition

In figure 4.18 we are seeing a transaction which is located inside of a partition. This case is a good indicator for being a coinmixer's transaction, since the fee should not be able to change within a partition. The counter is increased by 4 (3+1), if the transaction's fee meets the partition's fee.

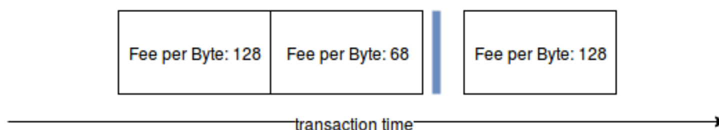

Figure 4.19: Transaction located in a gap

As we can see in figure 4.19, the transaction may be located within a gap of partitions. When the transaction's fee is located between partitions it is also a good indicator, since we assume that there are no huge gaps between the partitions. The counter is increased by 4 (3+1), if the transaction's fee meets one of the surrounding partition's fees.

While analyzing existing partitions seems to be easy, it is also important to update them. In the forward crawling process the partitions are going to get updated, whenever the transaction is not located within a partition.

In figure 4.20 we can see a partition which got extended, after a processed transaction was located within a gap of partitions.

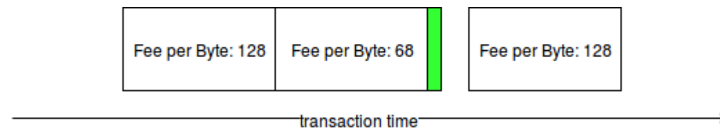

Figure 4.20: Updating a fee partition

We won't get into more details of other indicators, since we already discussed them before.
The results of the counting process will be stored in the database in *coinmixer_analysis_log* and *coinmixer_analysis*. While in *coinmixer_analysis_log* the results of each address analysis are stored, *coinmixer_analysis_log* primary logs which transaction's hash already has been processed and should not be processed in further crawling processes.
After the crawler determined which of the output addresses is the change address, he follows the change address's sent transaction and starts the analysis of it. In this way the crawler is able to follow the change addresses and create the coinmixer's network based on them.

4.5.4 Backward crawling

The backward crawling process also starts with a user specified coinmixer transaction. However, now the inputs of these transaction are processed. It is checked whether an input address is a *cash-in* address.

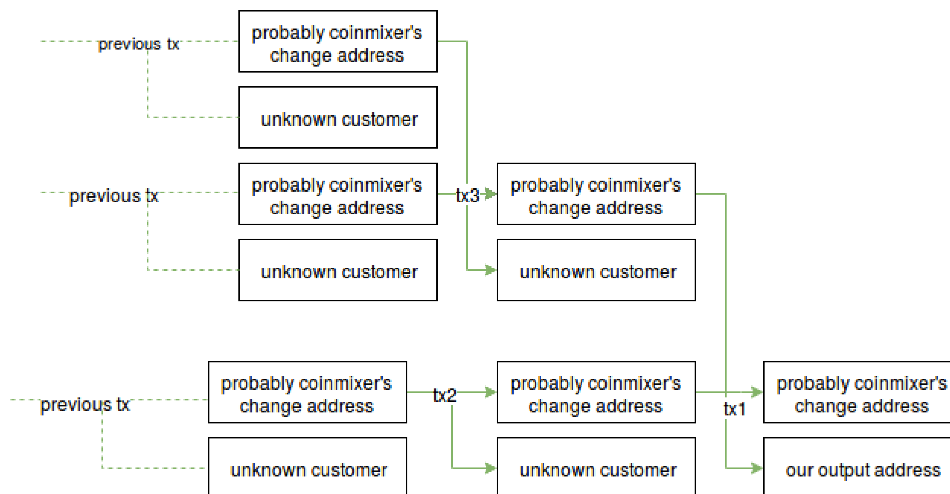

Figure 4.21: Backward crawling process

In figure 4.21 the user provides tx1 as the coinmixer's output transaction. Starting

from there, the crawler should be able to identify tx2, tx3 and other connected previous transactions.
The backward crawling process is primary based on the forward crawling process. However, some specific adjustments were needed to be implemented. While the forward crawling process checks whether an output address is controlled by the coinmixer, all input addresses of the transaction have to be controlled by coinmixer. If the inputs of the transaction would not be controlled by the coinmixer, it would not be a coinmixer transaction. The main task in the backward crawling process is not to identify who controls the inputs, but rather to identify who controls the inputs of the inputs.

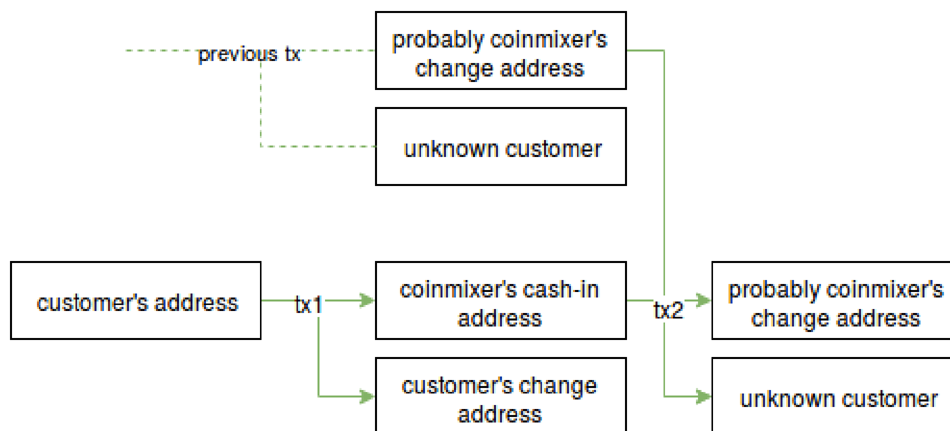

Figure 4.22: Identifying cash-in and change addresses

In figure 4.22 the crawler checks whether an input address is a cash-in address or a change address from a previous mixing process. If it's a change address, this path will be followed.
The user is able to specify a maximum crawling depth. This is the depth a single chain of change addresses will be followed. It is recommended to choose a moderate maximum depth (< 50), since one of the change address chains most probably will lead to the first coinmixer transaction the crawler is able to find. After a backward crawling process is interrupted or the maximum depth is reached, the crawler will find the last transactions processed in each chain and continues the crawling process from there.
While the backward crawling works similar to the forward crawling, there is one essential difference between both processes. At the forward crawling process only two outputs have to be checked. Based on our assumptions, one of the outputs has to be a coinmixer's change address, while the other address is a customer's address. However, there is no restriction how many input addresses are cash-in addresses and how many are change addresses. We are not able to distinguish between these types of transactions based on a counter.
The crawler categorizes transaction's input addresses as coinmixer address only if

each of the following requirements are met:

- Only one transaction have been sent through this address
- Only one transaction have been received through this address
- Following requirements are met for the received transaction
  - The sequence number of all inputs is set to 4294967294
  - The transaction's version is set to 2
  - The transaction's locktime is set to at least 1
- One of the following requirements have to be met
  - The received value has to be specified to at least five decimals
  - The fee of the transactions is located within a fee partition or a gap

If an input address is not categorized as a coinmixer address, it is automatically categorized as a cash-in address.
All of the requirements are based on the known forward crawling indicators. Most of the requirements are strong indicators for being a change address. We discussed them already. If any of these strong indicators is not met, we identify this transaction not as a coinmixer transaction.
The first of the last two requirements is based on the assumption that a change address typically does not receive a common value and is typically specified to eight decimals.
The last assumption checks whether the fee of the transaction matches existing fee partitions. In case of backward crawling, the fee indicator is not reliable, since the transactions, which are going to be analyzed, are sent prior to the provided output transaction and are typically before any existing fee partition.
As we can see, the backward crawling process is way more error-sensitive than the forward crawling process.
Updating fee partitions based on the results of the backward crawling process is critical, since it could influence further crawling processes.
In our implementation of the backward crawling process, the fee partitions will be updated in the same manner as it is done at the forward crawling. If no fee update would be provided, it could lead to a situation where the crawler might not be able to find transactions which are sent prior to the input coinmixer transaction. No new fee partitions would be created.
However, the partition updating rules can be manually changed through the *force* parameter in function *cm_check_transaction_fee_correct_partition_and_update*. If the *force* parameter is not set to *True*, no new partitions are going to be created. Nevertheless, old partitions will still be extended.

4.5.5 Incorrect transaction distinguishing

The crawling processes are based on indicators. However, it is still possible that the crawler did choose a wrong differentiation between customer/cash-in and change

addresses. As we already described, the backward crawling process is way more error-sensitive. Some specific errors might occur, when a customer's input transaction is located in the same fee partition as the coinmixer's transactions and further transaction characteristics (version, sequence number, locktime) are met. Furthermore, errors could occur, if a customer uses a coinmixer's output transaction to spend it to a coinmixer input address or multiple input transactions are sent. However, these should be rare situations.
We implemented an *error* class which is able to catch these exceptions. Since the mentioned strong indicators are only met for around 17 % of Bitcoin transactions (see Characteristics of coinmixer's output transactions), an error should be raised very soon after an erroneous differentiation. These exceptions are primarily captured whenever the crawler is stuck in a situation which contradicts with the known indicators. For example, the crawler will log an error, if an identified coinmixer's change address sent multiple transactions.
An analysis of erroneous behavior might lead to the identification of other network specifics. It even could lead to the identification of addresses which are manually coordinated through the service provider.
Erroneous transactions are logged and will be skipped on the next crawling process. It's important to know that the crawler is only going to stop crawling the identified erroneous change address chain.
The crawling process is not going to be interrupted, every other change address chain will still be crawled. However, erroneous identified coinmixer transactions should be analyzed manually, since it is possible that they flawed fee partitions.
For the sake of completeness, it should be noted, that in rare situations, where the crawler did a wrong classification and a wrongly identified coinmixer transaction is part of another generic Bitcoin subnetwork, which behaves similar to coinmixer.se and is based on the same strong indicators and is not in conflict with the already known fee partitions, the crawler might not be able to identify the erroneous classification.

4.6 Deanonymization

As we described in Attacking Method, the deanonymization process is not going to take the internal structure of the coinmixer into account. Our implementation will deanonymize transactions based on the sent and received Bitcoin amount.
In subsection Mixing fee we described how the input of the coinmixing service is calculated. Since we identified the coinmixer.se network through the crawling process, we are now able to map every possible coinmixer output to a given input transaction. Vice versa.
A *mapping* provides the deanonymization of a given anonymized transaction. The mapping is based on the equation mentioned in Mixing fee.
We implemented a deanonymization process as a Proof of Concept. Based on a given

customer's input transaction, a maximum number of possible forward addresses and an optional maximum time delay, our implementation calculates and prints out every possible deanonymization.
It should be noted that our implementation might print out duplicate mappings (see Future Work).
The implementation of this Proof of Concept is able to calculate possible mappings for up to three forward addresses.

4.7 Results

In the last subsections we described how we implemented the crawling and deanonymization processes. Now we examine how good our implementation works in practice. Our practical verification is based on the same transactions mentioned in Characteristics of coinmixer's output transactions. The first transaction which we are going to analyze is *1e11f70d0db8c177a19ebdbc782e0b7bfddaef3e314f7b339f702bd76d76b76f*. We have sent this transaction to coinmixer.se on 2017-09-26 18:36:02 UTC as an input transaction for a mixing process. This transaction has been confirmed by the Bitcoin network through the main chain block at height 487072. We will later describe more specifics of the settings, which we have chosen for this mixing process.
We chose the mentioned transaction as the starting point for the forward crawling process.
It should be stated, that typically the forward crawling process starts with a given output transaction, however, since our input transactions has been used without any big time delay in an output transaction sent by the coinmixer, we are also able to use our input transaction as starting point.
While we have sent the tainted coins through our input transaction at 2017-09-26 18:36:02 UTC, the next transaction, which uses our tainted coins, was sent on 2017-09-26 19:40:38 UTC. Since the transaction at 2017-09-26 19:40:38 UTC is the first coinmixer transaction which uses our tainted coin, this transaction should be seen as the initial starting point of the crawled network.
To be able to crawl as much transactions as possible, we start with the forward crawling process and after that apply the backward crawling process.
The last transaction, which we received on an address controlled by us from coinmixer.se, has been sent on 2017-09-28 01:37:46 UTC. An analyzes of the crawled network until that point of time would be enough to deanonymize all of our input transactions, however, we let the crawling process proceed to identify more of the whole coinmixer.se network. We stopped the crawling process at a transaction which was sent on 2017-10-11 18:08:41 UTC.
The forward crawling process was able to identify 486 transactions which were sent in the time frame from 2017-09-26 19:40:38 UTC to 2017-10-11 18:08:41 UTC. Based on our assumptions, all of these transactions belong to the coinmixer.se network.

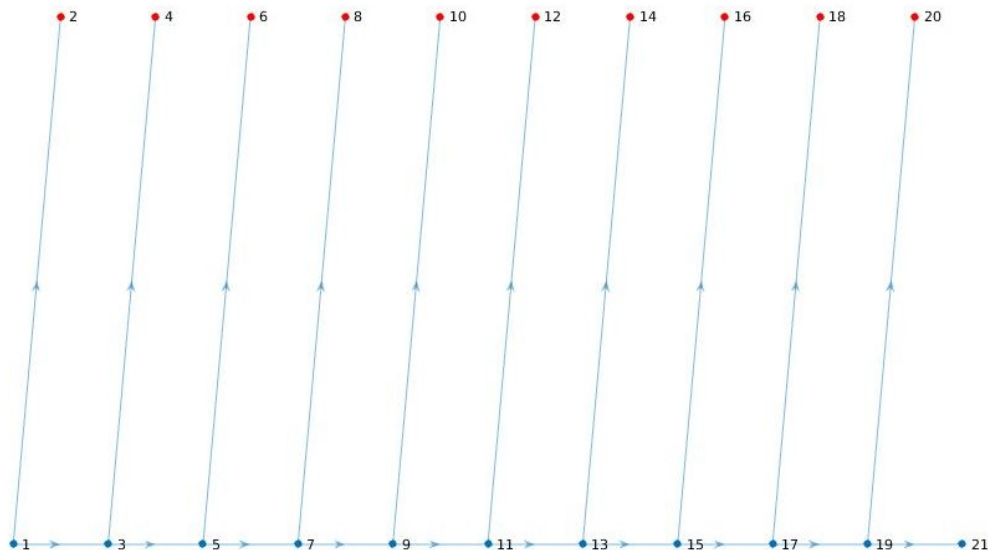

Figure 4.23: Part of coinmixer.se network based on forward crawling. Customer transactions: Red (dummy), coinmixer transactions: Blue

Figure 4.23 shows 20 of 486 crawled transactions. Since the rest of the graph looks the same, we only provide this excerpt. Every node should correspond to a valid Bitcoin transaction. The blue nodes should show change address transactions sent by coinmixer.se and the red nodes should show transactions sent by customers.
In case of coinmixer transactions this is true, however, in case of customer's transactions there also could be cases where a customer received untainted Bitcoins but did not spent them yet. Yet still, we created also for these cases (red) nodes, since otherwise the network infrastructure might not be easily understandable.
Generally speaking, blue nodes show transactions which have been made by the coinmixer, while the red nodes show transactions which were spent by customers or can be spent by them in future.

We were able to identify 486 transactions, however, based on the published mixing statistics of coinmixer.se more than 1000 coinmixer transactions have been sent in this time frame. To identify more transactions sent by the coinmixer, we need to run a backward crawling process. As starting point of the backward crawling process we choose the endpoint of the forward crawling process and a crawling depth of 20. The endpoint transaction of the forward crawling process was *2430b04b47520764d82422e696c1f39c85ca568f381729e427eea2b9c12c6190* (2017-10-11 18:08:41 UTC).
The earliest transaction which our backward crawling process was able to find was

sent on 2017-08-28 19:23:22 UTC, which is way before the time frame we want to analyze. However, since the timestamp of every transaction is stored by the crawling process, filtering can easily be accomplished through a MySQL query.
Our backward crawling process was able to identify 3609 transactions. We cannot specify any specific time frame for these transactions, since they depend on the depth of recursion.

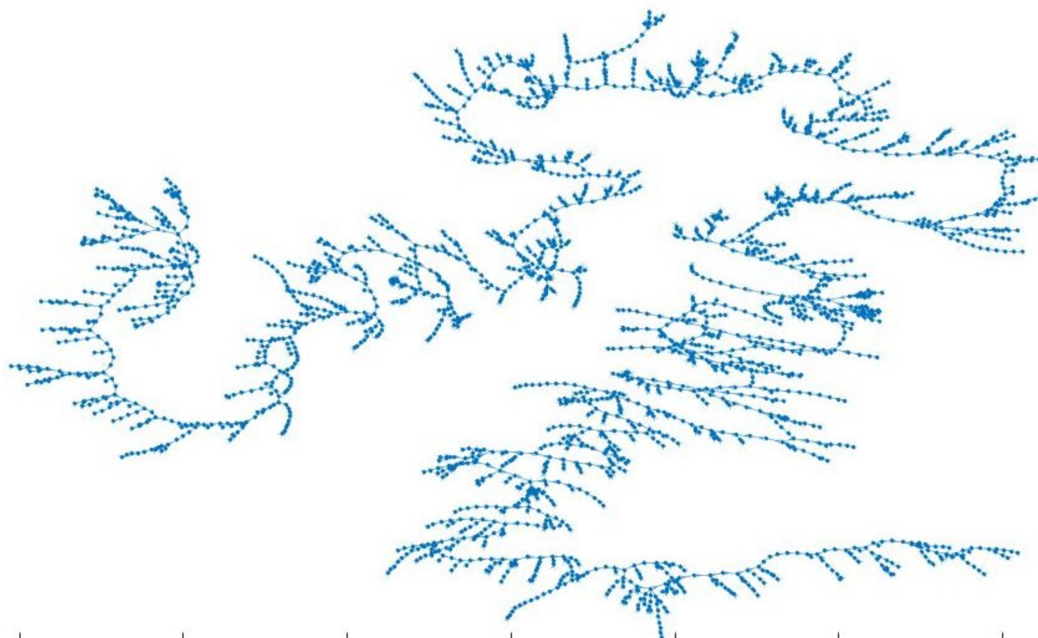

Figure 4.24: Crawled coinmixer.se network without customer transactions

To simplify our graph we removed all customer transactions. Figure 4.20 only shows coinmixer's change address transactions.
Based on forward and backward crawling we were able to create coinmixer.se's network. However, we still need to verify that the found network is the actual coinmixer.se network. We do this through the approach we described in Identifying customer's and coinmixer's transactions.
We received 14 transaction's from coinmixer.se. We were able to identify all of them in the crawled network.

Figure 4.25: Part of crawled network. Received coinmixer transactions highlighted.

In figure 4.25 we highlighted every transaction node which coinmixer.se sent to an address controlled by us.
Our crawling process covers a whole week (2017-10-02 00:00:00 UTC - 2017-10-08 23:59:59 UTC). Coinmixer.se's published statistic data stated, that their service anonymized 1040 input transactions in the specified week [7].
As we know, based on 1040 coinmixer.se participations at least 1040 transactions and at most $1040 * 5 = 5200$ transactions should have been sent through the coinmixer in the particular week. Our crawler was able to identify 1069 transactions sent by coinmixer.se in the mentioned week. If our crawler did not miss any coinmixer transaction, this result would imply that hardly any customer uses the optional setting of specifying multiple forward addresses. This makes for most use cases sense, since customer's typically do not want to spend coins through different addresses. However, we are not going in further analysis of the reliability of the crawled network, since all of our transactions have been found in it.
In Future Work, we will describe possibly ways to identify how accurate our crawled network is. Furthermore it should be noted, that the statistics provided by coinmixer.se may not be reliable. While coinmixer.se publishes every week the statistic of last week's participations, it is not known how they choose this time frame. Furthermore the published statistics my not be trustworthy at all.
Since we were able to create the crawled network, we now need to deanonymize transactions. For a given customer's input transaction (tainted coins), our deanonymization process tries to find any coinmixer's output transactions which might receive the untainted coins. Our deanonymization tool is fully based on the crawled network. There is no further blockchain analysis done in the deanonymization process.

We are going to analyze how reliable the results of our implementation are by trying to deanonymize transactions which we priorly anonymized with the coinmixer. At first we will focus on an anonymization process where we chose one forward address and a time delay of 0 hours.

We mixed 0.001 BTC on 2017-09-27 23:23:32 UTC and sent 0.00162812 BTC through the input transaction *74fb84d805fe35f00141fdca4f07a5a36e64b67fc1cab895033bb338c78d1d26"* to coinmixer.se. We did not set any time delay. The forwarding fee at the time of sending was 0.0006 BTC per forward address. At 2017-09-28 01:37:46 UTC we received the anonymized Bitcoins at address *13g89ys797GE9QtP1kj8Nv1gDfYFU1nps4*.
Our aim was to deanonymize this transaction. Through the given input transaction, the output address should be found by our implementation. A maximum time of delay might be specified through the attacker. Based on the information the attacker might know about the mixing process we specified multiple attacking scenarios.

Table 4.26: Results for first testing case (0h delay, 1 forward)

Max. forwards	**Max. time delay**	**Possible output addresses**	**False positives**
1	<=12 h	1	0
1	<=24 h	3	2
1	<=48 h	4	3
1	<=72 h	5	4
1	<=120 h	11	10

Since we sent the minimum possible amount of 0.001 BTC, this input could not be divided in multiple output transactions.
Our tool was able to identify *13g89ys797GE9QtP1kj8Nv1gDfYFU1nps4* as our output address. If the attacker knows, that the mixing process took place within 12 hours after sending the input transaction, he would be able to deanonymize the transaction without any false positives. The deanonymization of the transaction would be successfully accomplished.
If the attacker does not have any information about the chosen time delay, our tool would return 11 results with 10 false positives.
It should be noted, that even if we would have specified a time delay to up to 12 hours at the mixing process, the deanonymization results would still be the same.
As we can see, the deanonymization results are varying with the amount of information known by the attacker. Even for the worst case of 120h delay, 10 false positives seem to be a good result for beeing filtered out of 2 million Bitcoin transactions.

For most use cases the use of multiple forward addresses does not seem to be useful, since typically the customer only wants to anonymize the coins and do not want to split them across different addresses. Furthermore, the use of multiple addresses, which are managed through the same Bitcoin wallet, do not enhance the privacy (see Privacy in Bitcoin). The forward addresses should never be combined again.
Nevertheless, we are trying to deanonymize a coinmixer.se participation where we specified three forward addresses. Our implementation has to check every possible combination of transactions which might be the possible deanonymization for the specified input transaction.
We tried to deanonymize a transaction with three forwards. We specified 0.001 BTC, 0.001 BTC and 0.00100001 BTC as the amounts of untainted coins we want to receive. As time delays we chose values between 0 and 7 hours.
First we wanted to check if there may be a single transaction which could be identified as a false positive for our input transaction. None false positive has been found. In the next step we tried to identify possible false positives for two forwards. Every combination of two forwards would be a false positive, since we chose three forwards in the mixing process. Our implementation was able to identify 33 different Bitcoin addresses as false positives for two forwards and a possible delay of 120 hours. If the attacker would know that the mixing is done within 8 hours after the initial input transaction, no false positives would be found.
When our implementation checked all possible combinations of three forwards with no knowledge about the time delay, it could identify 24 different Bitcoin addresses. This test was done with a maximum delay of 120 hours. If the attacker would know that the mixing has been done within 8 hours after the input transaction, our implementation would be able to identify 9 possible output transactions.
It is important to state, that in the last testing scenario, six addresses which were identified as false positives, were addresses under our control. These addresses were output addresses of other testing cases we have done. These addresses have been identified as false positives, since we used the same output amount of 0.001 BTC for multiple testing cases. Since these six addresses are artificially produced by our testing procedures, we need to ignore them in our results.
When we ignore them, we receive the results shown in table 4.27 for the deanonymization process of a mxing process using three forwards:

Table 4.27: Results for second testing case (up to 7h delay, 3 forwards)

Max. forwards	**Max. time delay**	**Possible output addresses**	**False positives**
1	120 h	0	0
2	8 h	0	0
2	120 h	27	27
3	8 h	3	0
3	120 h	18	15

If the attacker knows that the mixing process has taken place within 8 hours, he would be able to deanonymize the transaction by finding all three output addresses without any false positive.
If he does not have any information about the set mixing options, the result set of two forwards would contain 24 false positives and the result set of thee forwards 15 false positives.
It should be mentioned, that our results only show found unique addresses, however, there are multiple combinations of these addresses possible.
Generally speaking, our implementation is able to identify coinmixer outputs which are based on a similar input. If and how many false positives are going to be found through our implementation is primarily based on the mixing behavior of other customers. If we are the only customer who tries to anonymize a specific amount of Bitcoins, our tool will be able to identify the exact output transaction even with no knowledge about the specified time delay. However, if multiple customers are mixing the same amount of Bitcoins at the same time, the results are going to contain multiple false positives. In our testing scenario we were able to deanonymize our input transaction without any false positives when the time frame of the mixing procedure could be limited.

5 Conclusion

Through our implemented crawling methods we were able to filter 3609 out of more than 2 million Bitcoin transactions. Most probably all of these 3609 transactions define the coinmixer.se network in a specific time frame. Our crawler accomplished this through a simple blockchain analysis. The identified network could be verified through multiple transactions we received by coinmixer.se. Based on our crawled network it can be assumed, that in our testing time frame most of the coinmixer.se's customers did not use the optional setting of specifying multiple forward addresses. Our implementation was able to deanonymize two transactions which we priorly anonymized through coinmixer.se. When we set the maximum time delay to the time delay specified in the mixing process, the deanonymization results were correct and did not contain any false positives.
Based on our analyzes we can conclude, that the anonymization process of coinmixer.se is heavily based on the mixing behavior of other customers. While our implementation was able to deanonymize our testing cases, it does not necessarily mean that every transaction of coinmixer.se could be deanonymized through our implementation. However, a customer cannot be sure if the coins he received could easily be deanonymized. Practical use cases where time delay to up to 120h and multiple forward addresses should be used are limited. Based on our analyzes the mixing process of coinmixer.se is not able to provide reasonable privacy for practical use.

5.1 Related Work

Satoshi Nakamoto introduced Bitcoin through *Bitcoin: A Peer-to-Peer Electronic Cash System* in 2008 [37]. Privacy in cryptocurrencies have been addresses in multiple scientific publications. Dorit Ron and Adi Shamir analyzed the the transaction flow of the Bitcoin network based on a transaction graph [39]. Multiple algorithms have been published which are able to enhance privacy in the Bitcoin network. Ian Miers, Christina Garman, Matthew Green and Aviel D. Rubin published Zerocoin, an extension which could be introduced to the Bitcoin network to enhance it's privacy [33]. While this protocol extension was not implemented in the Bitcoin protocol, the cryptocurrency *Zerocash* is based on it's main protocol idea [42]. However, Zerocash is able to provide *strong privacy* guarantees and is based on a zero-knowledge proof.
Tim Ruffing, Pedro Moreno-Sanchez and Aniket Kate introduced *CoinShuffle* and the improved *CoinShuffle++* protocol. CoinShuffle++ is based on the current Bitcoin system. Through this mixing protocol transaction could be made unlinkable

[41].
Furthermore, *CoinJoin* [30], *Mixcoin* [14], *Coinparty* [44], *Xim* [12] and *Tumblebit* [26] have been introduced as mixing protocols.
The implementation of the *Lightning Network* may enhance privacy of the Bitcoin network. [38] Analyzes of centralized Bitcoin mixing services and privacy enhancing overlays also have been published. Malte Möser, Rainer Böhme and Dominic Breuker analyzed Bitcoin Fog, BitLaundry, and the departed mixing function of Blockchain.info as centralized mixing services through a taint analysis [35]. Sarah Meiklejohn and Claudio Orlandi analyzed multiple mixing algorithms and services like coinjoin [32].

5.2 Future Work

While our implementation seems to be able to deanonymize transactions which priorly have been anonymized by coinmixer.se, improvements could lead to less false positives. The deanonymization process of our implementation calculates duplicates. Some of the found mappings are provided multiple times in a different order. Through further development, this bug could be fixed.
In case of the forward crawling process, the counter of both addresses may result in the same amount. This is the case, if one of the output addresses received an uncommon value, which it spent, and is newer than every fee partition, while the other transaction is located in a fee partition but no other low/medium indicators are met. All other strong indicators have to be met for both addresses. This is a very special case and should occur rarely, since only 0.52% of blockchain transactions fulfill the strong indicators and meet the fee indicator. In none forward crawling processes we experienced this situation, however, this bug should be fixed through further development.
Furthermore, the deanonymization process should be implemented recursively and should be able to deanonymize transaction which have been anonymized with up to five forward addresses.
A blockward crawling process might be able to identify coinmixer transactions which can't be found through the other crawling processes. While our implementation is able to deanonymize specified input transactions, it should also be implemented to deanonymize the input transaction based on given output transactions.
We only applied simple blockchain analysis, a byte-level analyzes of transactions sent by coinmixer.se might lead to better transaction indicators.
Furthermore, a dynamically mapping of input transactions and possible output transactions would be able to create multiple network possibilities based on the found network. Our deanonymization process is static, however, a dynamic deanonymization, which analyzes the behavior of output addresses could lead to less false positives.
In Identifying customer's and coinmixer's transactions we described a way to verify the identified coinmixer network. Through a mapping of every input transaction to

possible output transactions the verification process may be improved. Furthermore, coinmixer.se published how many Bitcoins it anonymized within the last week. This information could also be used to verify the coinmixer.se network.
While these are aspects of the current implementation, which can be improved, we were also able to identify multiple attacking possibilities on other mixing services.
We were able to identify that bitmixer.io, a discontinued mixing service, set the transaction fee of every output transaction in a range of 0.0002 BTC to 0.0008 BTC. It also sent the output transaction automatically seconds after the input transaction reached the necessary number of confirmations. While this coinmixing service is not available anymore, the anonymized transactions are still able to be deanonymized. A deanonymization of this service could be implemented in our existing deanonymization tool.
Furthermore, we found a web security bug in cryptomixer.io, another centralized mixing service. Cryptomixer.io provides every customer with a *Letter of Guarantee* prior to a mixing process. Through this PDF document, the transaction flow of the tainted and untainted coins can easily be traced. While this document should only be accessible for the individual customer, no session data are checked by cryptomixer.io when accessing it. An attacker is able to gain access to the letter of guarantee of other customers by changing a POST-Parameter. Through a simple script every mixing process can be observed by the attacker.
This attack should also be implemented in further development. In general, we were not able to identify a secure implementation of a mixing service. Through further development of our implementation, our implementation could be able to attack most of the common mixing services.

List of Figures

2.1 Simplified Bitcoin blockchain [9] 6
2.2 P2PKH transaction [9] 7
2.3 P2SH transaction [9] 8
2.7 Simplified Bitcoin transaction flow 13
2.8 Bitcoin transaction without fee 14
2.9 Bitcoin tranction with fee 14
2.10 Bitcoin tranction with fee 14
2.11 Bitcoin transaction with change address 15
2.12 Bitcoin transaction with dynamic fee 16
2.13 Bitcoin transaction with connected change addresses 17
2.14 Bitcoin mixing 18
2.15 P2P mixing 18
2.16 Centralized Mixing Service 19

3.1 Stale and orphan block [9] 26

4.1 Funcionality of coinmixer.se [6] 29
4.2 The default case of a mixing process and a case which makes use of the optional settings 30
4.3 Fee partition 34
4.6 Analyzed coinmixer's output flow 38
4.8 Analyzed chain of change addresses 39
4.9 Analyzed full coinmixer network 39
4.10 Confirm coinmixer network 40
4.11 Forward (red) and backward (green) crawling process 41
4.12 Transaction chain which can't be found by forward/backward crawling (blue) 41
4.13 Forward crawling 43
4.15 Fee partition 45
4.16 Transaction newer than every fee partition 45
4.17 Transaction older than every fee partition 46
4.18 Transaction located within a fee partition 46
4.19 Transaction located in a gap 46
4.20 Updating a fee partition 47
4.21 Backward crawling process 47
4.22 Identifying cash-in and change addresses 48

4.23 Part of coinmixer.se network based on forward crawling. Customer transactions: Red (dummy), coinmixer transactions: Blue 52
4.24 Crawled coinmixer.se network without customer transactions 53
4.25 Part of crawled network. Received coinmixer transactions highlighted. 54

List of Tables

2.4 Example of P2PKH and P2SH transaction hashes 8
2.5 Transaction's settings based on sequence number 9

4.4 *Strong* and *good* indicators to spot coinmixer's transactions 34
4.5 Transactions received from coinmixer.se between 2017-09-26 00:58:49 UTC and 2017-09-28 01:37:46 35
4.7 Indicators to distinguish between customer's and coinmixer's address 38
4.14 Forward crawling counter 44
4.26 Results for first testing case (0h delay, 1 forward) 55
4.27 Results for second testing case (up to 7h delay, 3 forwards) 56

List of Listings

2.6 JSON-formatted Bitcoin transaction 11

Bibliography

[1] Blockchain.info charts. `https://blockchain.info/charts`, . Accessed: 2017-12-02.

[2] Blockchain.info blockchain size. `https://blockchain.info/charts/blocks-size`, . Accessed: 2017-12-02.

[3] Bitcoin core privacy features. `https://Bitcoin.org/en/Bitcoin-core/features/privacy`. Accessed: 2017-12-02.

[4] Market cap of cryptocurrencies. `https://coinmarketcap.com/`. Accessed: 2017-12-02.

[5] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. "O'Reilly Media, Inc.", 2014.

[6] Unknown Author. Coinmixer.se website. `https://coinmixer.se`, 2017. Accessed: 2017-12-02.

[7] Unknown Author. Coinmixer.se anonymization statistics. `https://coinmixer.se`, 2017. Accessed: 2017-10-11.

[8] Unknown Author. Coinmixer.se anonymization statistics. `https://coinmixer.se`, 2017. Accessed: 2017-11-21.

[9] Unknown Author. Bitcoin developer guide. `https://Bitcoin.org/en/developer-guide`, 2017. Accessed: 2017-12-02.

[10] Unknown Author. Segwit charts. `http://segwit.party/charts/`, 2017. Accessed: 2017-12-02.

[11] Lear Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013*, 2013.

[12] George Bissias, A Pinar Ozisik, Brian N Levine, and Marc Liberatore. Sybil-resistant mixing for bitcoin. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 149–158. ACM, 2014.

[13] Rainer Böhme, Nicolas Christin, Benjamin Edelman, and Tyler Moore. Bitcoin: Economics, technology, and governance. *The Journal of Economic Perspectives*, 29(2):213–238, 2015.

[14] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A Kroll, and Edward W Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In *International Conference on Financial Cryptography and Data Security*, pages 486–504. Springer, 2014.

[15] Grace Caffyn. What is the bitcoin block size debate and why does it matter. *URL: http://www. coindesk. com/what-is-the-Bitcoin-block-size-debate-and-why-does-it-matter/(visited on 27/11/2015)*, 2015.

[16] Bitcoin Core. Segregated witness benefits. *URL https://Bitcoincore. org/en/2016/01/26/segwit-benefits/.[Online*, 2016.

[17] Who Has Custody. Optimizations, confirmation, contest and postlocking periods sidechain implementation using smartcontract in the secondary chain side sidechain implementation using specific opcodes in the bitcoin side sidechain implementation using turingcomplete scripting in the bitcoin side drivechain.

[18] Peter Todd David A. Harding. Bip 0125: Opt-in full replace-by-fee signaling, 2015.

[19] Christian Decker and Roger Wattenhofer. Bitcoin transaction malleability and mtgox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[20] Bitcoin Core Developers. Bitcoin core. *ht tps://Bitcoin. org.*

[21] Jochen Dinger and Hannes Hartenstein. Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration. In *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 8–pp. IEEE, 2006.

[22] Anne Haubo Dyhrberg. Hedging capabilities of bitcoin. is it the virtual gold? *Finance Research Letters*, 16:139–144, 2016.

[23] Mark Friedenbach. Bip 0068: Consensus-enforced transaction replacement signaled via sequence numbers (relative locktime), 2015.

[24] Steven H Gifis. *Dictionary of legal terms.* Barron's Educational Series, 2016.

[25] DA Harding. Bitcoin developer guide, 2015.

[26] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. *Cryptology ePrint Archive, Report 2016/575, Tech. Rep.*, 2016.

[27] Jordi Herrera-Joancomartí. Research and challenges on bitcoin anonymity. In *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, pages 3–16. Springer, 2015.

[28] Stanković B Ivica, Mihajlović R Aleksandar, and Mihajlović A Radomir. Crypto-currency and e-financials. *OF ECONOMICS AND LAW*, page 132, 2014.

[29] Johnson Lau. Bip 0142: Address format for segregated witness, 2015.

[30] Greg Maxwell. Coinjoin: Bitcoin privacy for the real world. In *Post on Bitcoin Forum*, 2013.

[31] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. Refund attacks on bitcoin's payment protocol. In *International Conference on Financial Cryptography and Data Security*, pages 581–599. Springer, 2016.

[32] Sarah Meiklejohn and Claudio Orlandi. Privacy-enhancing overlays in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 127–141. Springer, 2015.

[33] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

[34] Malte Möser. Anonymity of bitcoin transactions. In *Münster Bitcoin conference*, pages 17–18, 2013.

[35] Malte Moser, Rainer Bohme, and Dominic Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *eCrime Researchers Summit (eCRS), 2013*, pages 1–14. IEEE, 2013.

[36] Malte Möser, Rainer Böhme, and Dominic Breuker. Towards risk scoring of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pages 16–32. Springer, 2014.

[37] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[38] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. *cit. on*, page 89, 2015.

[39] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[40] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *European Symposium on Research in Computer Security*, pages 345–364. Springer, 2014.

[41] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2p mixing and unlinkable bitcoin transactions. *IACR Cryptology ePrint Archive*, 2016:824, 2016.

[42] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.

[43] Neudecker Till. Bitcoin cash (bch) sybil nodes on the bitcoin peer-to-peer network, 2017.

[44] Jan Henrik Ziegeldorf, Fred Grossmann, Martin Henze, Nicolas Inden, and Klaus Wehrle. Coinparty: Secure multi-party mixing of bitcoins. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 75–86. ACM, 2015.

A Database structure

```
SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Database: ‘Bachelorarbeit‘
--

-- --------------------------------------------------------

--
-- Table structure for table ‘address_and_value_mapping‘
--

CREATE TABLE ‘address_and_value_mapping‘ (
  ‘ID‘ int(11) UNSIGNED NOT NULL,
  ‘address_list‘ mediumtext COLLATE latin1_german1_ci NOT NULL,
  ‘value_list‘ mediumtext COLLATE latin1_german1_ci NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table ‘coinmixer_analysis‘
--

CREATE TABLE ‘coinmixer_analysis‘ (
  ‘analysis_id‘ int(11) NOT NULL,
  ‘address_hash‘ char(35) COLLATE latin1_german1_ci NOT NULL,
  ‘forward‘ tinyint(1) NOT NULL,
  ‘fee‘ tinyint(1) DEFAULT NULL,
  ‘common‘ tinyint(1) DEFAULT NULL,
  ‘version_sequence_locktime‘ tinyint(1) DEFAULT NULL,
  ‘connected‘ tinyint(1) NOT NULL,
  ‘spent‘ tinyint(1) DEFAULT NULL,
  ‘transaction_hash‘ char(64) COLLATE latin1_german1_ci NOT NULL,
  ‘next_transaction_hash‘ char(64) COLLATE latin1_german1_ci DEFAULT NULL,
  ‘is_cm‘ tinyint(1) NOT NULL,
  ‘is_cashin‘ int(11) DEFAULT NULL,
  ‘transaction_history‘ tinyint(1) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table ‘coinmixer_analysis_log‘
```

```
--

CREATE TABLE `coinmixer_analysis_log` (
  `ID` int(10) UNSIGNED NOT NULL,
  `transaction_hash` char(64) COLLATE latin1_german1_ci NOT NULL,
  `previous_hash` char(64) COLLATE latin1_german1_ci DEFAULT NULL,
  `next_hash` char(64) COLLATE latin1_german1_ci DEFAULT NULL,
  `is_first` tinyint(1) DEFAULT NULL,
  `depth` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table `coinmixer_graph`
--

CREATE TABLE `coinmixer_graph` (
  `AnalysisID` mediumint(9) NOT NULL,
  `isCM` tinyint(1) NOT NULL,
  `connectedTo` mediumint(9) UNSIGNED NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table `errorlog`
--

CREATE TABLE `errorlog` (
  `ID` int(10) UNSIGNED NOT NULL,
  `error_data` varchar(2000) COLLATE latin1_german1_ci NOT NULL,
  `transaction_hash` varchar(64) COLLATE latin1_german1_ci DEFAULT NULL,
  `address_hash` char(35) COLLATE latin1_german1_ci DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table `fee_partition`
--

CREATE TABLE `fee_partition` (
  `partition_id` tinyint(3) UNSIGNED NOT NULL,
  `fee` mediumint(8) UNSIGNED NOT NULL,
  `timestamp_start` int(10) UNSIGNED NOT NULL,
  `timestamp_end` int(11) UNSIGNED NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table `list_of_all_transaction_hashes`
--

CREATE TABLE `list_of_all_transaction_hashes` (
  `transaction_hash` char(64) COLLATE latin1_german1_ci NOT NULL,
  `ID` mediumint(8) UNSIGNED DEFAULT NULL,
  `in_size_big_table` tinyint(4) DEFAULT NULL,
  `in_transaction_data` tinyint(1) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;
```

```
-- --------------------------------------------------------

--
-- Table structure for table ‘multiple_sequences‘
--

CREATE TABLE ‘multiple_sequences‘ (
  ‘transaction_id‘ int(11) NOT NULL,
  ‘sequences‘ text COLLATE latin1_german1_ci NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table ‘transactions_size_big‘
--

CREATE TABLE ‘transactions_size_big‘ (
  ‘hash‘ char(64) COLLATE latin1_german1_ci NOT NULL,
  ‘blockheight‘ mediumint(8) UNSIGNED NOT NULL,
  ‘transaction‘ mediumtext COLLATE latin1_german1_ci NOT NULL,
  ‘fullblock‘ tinyint(1) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table ‘transactions_size_normal‘
--

CREATE TABLE ‘transactions_size_normal‘ (
  ‘hash‘ char(64) COLLATE latin1_german1_ci NOT NULL,
  ‘blockheight‘ mediumint(8) UNSIGNED NOT NULL,
  ‘transaction‘ text COLLATE latin1_german1_ci NOT NULL COMMENT ’Up to 12 byte transactions’,
  ‘fullblock‘ tinyint(1) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table ‘transaction_addresses‘
--

CREATE TABLE ‘transaction_addresses‘ (
  ‘ID‘ int(10) UNSIGNED NOT NULL,
  ‘transaction_address‘ char(35) COLLATE latin1_german1_ci NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table ‘transaction_data‘
--

CREATE TABLE ‘transaction_data‘ (
  ‘transaction_id‘ int(10) UNSIGNED NOT NULL,
  ‘transaction_hash‘ char(64) COLLATE latin1_german1_ci NOT NULL,
  ‘blockheight‘ int(10) UNSIGNED NOT NULL,
  ‘fee‘ int(10) UNSIGNED NOT NULL,
  ‘size‘ int(10) UNSIGNED NOT NULL,
  ‘time‘ int(10) UNSIGNED NOT NULL,
  ‘version‘ tinyint(3) UNSIGNED NOT NULL,
```

```
  `sequence` int(10) UNSIGNED DEFAULT NULL COMMENT 'wenn null, dann daten in
      multipleSequences',
  `locktime` int(10) UNSIGNED NOT NULL,
  `inputaddress_value_mapping_id` int(10) UNSIGNED NOT NULL,
  `outputaddress_value_mapping_id` int(10) UNSIGNED NOT NULL,
  `is_cm` tinyint(1) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

-- --------------------------------------------------------

--
-- Table structure for table `transaction_values`
--

CREATE TABLE `transaction_values` (
  `ID` int(10) UNSIGNED NOT NULL,
  `transaction_value` int(10) UNSIGNED NOT NULL,
  `spent` tinyint(4) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_german1_ci;

--
-- Indexes for dumped tables
--

--
-- Indexes for table `address_and_value_mapping`
--
ALTER TABLE `address_and_value_mapping`
  ADD PRIMARY KEY (`ID`);

--
-- Indexes for table `coinmixer_analysis`
--
ALTER TABLE `coinmixer_analysis`
  ADD PRIMARY KEY (`analysis_id`),
  ADD KEY `isBM` (`is_cm`);

--
-- Indexes for table `coinmixer_analysis_log`
--
ALTER TABLE `coinmixer_analysis_log`
  ADD PRIMARY KEY (`ID`),
  ADD KEY `transactionHash` (`transaction_hash`),
  ADD KEY `previousHash` (`previous_hash`);

--
-- Indexes for table `coinmixer_graph`
--
ALTER TABLE `coinmixer_graph`
  ADD PRIMARY KEY (`AnalysisID`);

--
-- Indexes for table `errorlog`
--
ALTER TABLE `errorlog`
  ADD PRIMARY KEY (`ID`);

--
-- Indexes for table `fee_partition`
--
ALTER TABLE `fee_partition`
  ADD PRIMARY KEY (`partition_id`);
```

```
--
-- Indexes for table ‘list_of_all_transaction_hashes‘
--
ALTER TABLE ‘list_of_all_transaction_hashes‘
  ADD PRIMARY KEY (‘transaction_hash‘);

--
-- Indexes for table ‘multiple_sequences‘
--
ALTER TABLE ‘multiple_sequences‘
  ADD PRIMARY KEY (‘transaction_id‘);

--
-- Indexes for table ‘transactions_size_big‘
--
ALTER TABLE ‘transactions_size_big‘
  ADD PRIMARY KEY (‘hash‘),
  ADD KEY ‘fullblock‘ (‘fullblock‘),
  ADD KEY ‘blockheight‘ (‘blockheight‘,‘fullblock‘);

--
-- Indexes for table ‘transactions_size_normal‘
--
ALTER TABLE ‘transactions_size_normal‘
  ADD PRIMARY KEY (‘hash‘),
  ADD KEY ‘fullblock‘ (‘fullblock‘),
  ADD KEY ‘blockheight‘ (‘blockheight‘,‘fullblock‘);

--
-- Indexes for table ‘transaction_addresses‘
--
ALTER TABLE ‘transaction_addresses‘
  ADD PRIMARY KEY (‘ID‘),
  ADD UNIQUE KEY ‘inputAdress‘ (‘transaction_address‘);

--
-- Indexes for table ‘transaction_data‘
--
ALTER TABLE ‘transaction_data‘
  ADD PRIMARY KEY (‘transaction_id‘),
  ADD UNIQUE KEY ‘transactionHash‘ (‘transaction_hash‘),
  ADD KEY ‘isCM‘ (‘is_cm‘),
  ADD KEY ‘blockheight‘ (‘blockheight‘) USING BTREE;

--
-- Indexes for table ‘transaction_values‘
--
ALTER TABLE ‘transaction_values‘
  ADD PRIMARY KEY (‘ID‘),
  ADD KEY ‘values‘ (‘transaction_value‘);

--
-- AUTO_INCREMENT for dumped tables
--

--
-- AUTO_INCREMENT for table ‘address_and_value_mapping‘
--
ALTER TABLE ‘address_and_value_mapping‘
  MODIFY ‘ID‘ int(11) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=8429;
--
```

```
-- AUTO_INCREMENT for table ‘coinmixer_analysis‘
--
ALTER TABLE ‘coinmixer_analysis‘
  MODIFY ‘analysis_id‘ int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=12501;
--
-- AUTO_INCREMENT for table ‘coinmixer_analysis_log‘
--
ALTER TABLE ‘coinmixer_analysis_log‘
  MODIFY ‘ID‘ int(10) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=4196;
--
-- AUTO_INCREMENT for table ‘errorlog‘
--
ALTER TABLE ‘errorlog‘
  MODIFY ‘ID‘ int(10) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=43;
--
-- AUTO_INCREMENT for table ‘fee_partition‘
--
ALTER TABLE ‘fee_partition‘
  MODIFY ‘partition_id‘ tinyint(3) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=97;
--
-- AUTO_INCREMENT for table ‘transaction_addresses‘
--
ALTER TABLE ‘transaction_addresses‘
  MODIFY ‘ID‘ int(10) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=7539;
--
-- AUTO_INCREMENT for table ‘transaction_data‘
--
ALTER TABLE ‘transaction_data‘
  MODIFY ‘transaction_id‘ int(10) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=4065;
--
-- AUTO_INCREMENT for table ‘transaction_values‘
--
ALTER TABLE ‘transaction_values‘
  MODIFY ‘ID‘ int(10) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=6339;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

B Python Code

```
import urllib
import json
import MySQLdb

DEBUG = False


# =====================
# Database
# =====================
class Database:
    """
        Creates Database-connections and forwards SQL-statements.

    """
    db = MySQLdb.connect(host="127.0.0.1", user="xxxx", passwd="xxxxx", db="xxxx")

    def __init__(self):
        raise Exception("Should not be initialized")

    @staticmethod
    def sql_execute(sql):
        """
        Executes SQL-Statement

        :param sql: The sql-statement to execute
        :return: Returns result provided by mysql-Database
        """
        if DEBUG:
            print ("Class: Database; Method: sql_execute; SQL-statement: " + sql)

        cur = Database.db.cursor()
        cur.execute(sql)
        Database.db.commit()
        row = cur.fetchall()
        cur.close()
        return row


# =====================
# ErrorLog
# =====================
class ErrorLog:

    """ Through this class tansactions and addresses which created
        errors that could negatively interfere the crawling proccess
        are logged. Mainly errors which would lead to infinite loop-crawling are logged.
        """

    def __init__(self):
        raise Exception("Should not be initialized")
```

```
    @staticmethod
    def log(data, transaction_hash=None, address_hash=None):
        """
        Logs an occured error in database.

        :param data: Individual Error-Message.
        :param transaction_hash: Specifies the Bitcoin-transaction which lead to the error.
        :param address_hash: Specifies the Bitcoin-address which lead to the error.
        :return:
        """
        if DEBUG:
            print ("Class: ErrorLog; Method: log; error: " + str(data) + str(transaction_hash)
                  + str(address_hash))

        if transaction_hash is None and address_hash is None:
            return False

        if transaction_hash is None:
            transaction_hash = ""
        if address_hash is None:
            address_hash = ""

        sql = "INSERT IGNORE INTO errorlog (error_data, transaction_hash, address_hash) Values
              " \
              "('"+MySQLdb.escape_string(data)+"','" + transaction_hash + "', '" + address_hash
                    + "')"
        Database.sql_execute(sql)


class ParticipationPossibility:
    """
        A possible Participation. Its created whenever a search-process found a possible
             participation-outcome.
    """
    def __init__(self, outputtransaction, outputaddress, output_range, forwards):
        self._outputtransaction = outputtransaction
        self._outputaddress = outputaddress
        self._output_range = output_range
        self._forwards_number = forwards

    def get_outputaddress(self):
        return self._outputaddress

    def get_forwards_number(self):
        return self._forwards_number


# ====================
# ParticipationPossibilityContainer
# ====================
class ParticipationPossibilityContainer:
    """
        This class holds a container with each possible participation-possibility.
        Furthermore it executes the searching-process

    """

    def __init__(self, inputtransaction, output_range):
        self.container = []
        self.inputtransaction_time = inputtransaction.get_time()
        self.output_range = output_range
```

```
        self.blockheight_minimum = inputtransaction.get_blockheight() + 2
        self.transaction_time_minimum = self.get_block_timestamp()
        self.transaction_time_maximum = 0

        if self.transaction_time_minimum is False:
            raise Exception("Block could not be found.")

    def get_participations(self):
        return self.container

    def find_possibilities(self, forwards_maximum=3, max_delay=120):
        """
            Coordinates and executes the searching-process for possible mixing-participations
        :param forwards_maximum: The maximum number of forwards that should be checked
        :param max_delay: The maxiumum delay that should be checked
        :return:
        """

        if DEBUG:
            print("trying to find all possible mixings. forward_maxiumum: " + str(
                forwards_maximum) +
                 " maximum delay: " + str(max_delay))

        self.transaction_time_maximum = self.transaction_time_minimum + max_delay*3600

        if DEBUG:
            print("transaction-time minumum: " + str(self.transaction_time_minimum) + "
                maximum: "
                 + str(self.transaction_time_minimum))

        sql = "SELECT transaction_hash, outputaddress_value_mapping_id " \
              "FROM transaction_data WHERE is_cm=1 and time between " + \
              str(self.transaction_time_minimum) + " AND " + str(self.transaction_time_maximum)
        result = Database.sql_execute(sql)
        transaction_list = []

        for transaction_data in result:
            transaction_list.append(Transaction(transaction_data[0]))

        if DEBUG:
            print ("found " + str(len(transaction_list)) + " transaction in timeframe")

        for forwards_counter in range(forwards_maximum):
            if forwards_counter == 0:
                if DEBUG:
                    print("checking possibilities for 1 forward:")
                self.container += self.find_possibilities_1_forward(transaction_list, self.
                    output_range[0])
                if DEBUG:
                    print (self.container)
            elif forwards_counter == 1:
                if DEBUG:
                    print("checking possibilities for 2 forwards:")
                self.container += self.find_possibilities_2_forwards(transaction_list, self.
                    output_range[1])
                if DEBUG:
                    print (self.container)
            elif forwards_counter == 2:
                if DEBUG:
                    print("checking possibilities for 3 forwards:")
                self.container += self.find_possibilities_3_forwards(transaction_list, self.
                    output_range[2])
```

```
            if DEBUG:
                print (self.container)

        return self.container

    def get_block_timestamp(self):
        """
        This function returns the "first-seen"-timestamp on a block, which
        is specified by its blockheight. The Blockchain.info-API
        does not provide the block-headers for a given blockheight.
        To find the specific block, this function uses a timestamp of
        a transaction which should be inserted in a block on the same day.

        :return:   Returns the timestamp of the block at the given blockheight.
                   "False" returned if block coulndt be found.
        """

        transaction_timestamp = (self.inputtransaction_time+14400)*1000
        # second-based timestamp to millisecond-based timestamp
        url = "https://blockchain.info/de/blocks/" + str(transaction_timestamp) + "?format=
             json"
        response = urllib.urlopen(url)
        if DEBUG:
            print("loading list of blocks")
        try:  # todo: implement a way which also loads the next day (if transaction is send
             right before 0:00)
            data = json.loads(response.read())
            data_json = data
            block_time = 0 # default-value
            if DEBUG:
                print("trying to find block with blockheight " + str(self.blockheight_minimum))
            for block in data_json["blocks"]:
                if block["main_chain"] is False: # we only search on blocks on the main-chain
                    continue
                if block["height"] == self.blockheight_minimum:
                    if DEBUG:
                        print("block " + str(self.blockheight_minimum) + " found. blocktime: " +
                              str(block["time"]))
                    block_time = block["time"]

            if block_time == 0: # specified block coulndt be found.
                return False
            return block_time
        except ValueError:
            print("JSON-object could not be decoded. Probably your IP got blocked. Try again
                 later.")
            exit(0)

    @staticmethod
    def find_possibilities_1_forward(inputtransaction_list, output_range):
        """

        :param inputtransaction_list:
        :param output_range:
        :return:
        """
        if DEBUG:
            print ("transaction-value-range: " + str(output_range))
        if output_range[0] == 0 or output_range[1] == 0:
            return False

        value_minimum = output_range[0]
```

```
    value_maximum = output_range[1]
    if DEBUG:
        print("checking if any transaction-value is in range: ")
    participation_list = [] # list with possible participation
    for transaction in inputtransaction_list: # iterate through every transaction in list
        outputaddresses = transaction.get_outputaddress_list() # get transaction-list
        for outputaddress in outputaddresses:
            outputaddress.mixer_results_load(transaction.get_hash()) # load data for every
                outputaddress
            if not outputaddress.get_is_cm(): # only check addreses that are controlled by
                customer
                if value_minimum <= outputaddress.get_value() <= value_maximum: # check if
                    value in range
                    participation_list.append(ParticipationPossibility(
                        [transaction], [outputaddress], output_range, 1)
                    )
    return participation_list

@staticmethod
def find_possibilities_2_forwards(inputtransaction_list, output_range):
    value_minimum = output_range[0]
    value_maximum = output_range[1]

    participation_list = [] # list with possible participation
    for transaction in inputtransaction_list:
        outputaddresses = transaction.get_outputaddress_list()
        for outputaddress in outputaddresses:
            outputaddress.mixer_results_load(transaction.get_hash())
            if not outputaddress.get_is_cm():
                value_first = outputaddress.get_value()
                if value_first < value_maximum: # since there are two forwards,
                    #  the second forward has to have a value > 0

                    # second iteration:
                    for transaction_2 in inputtransaction_list:
                        outputaddresses_2 = transaction_2.get_outputaddress_list()

                        for outputaddress_2 in outputaddresses_2:
                            outputaddress_2.mixer_results_load(transaction_2.get_hash())
                            if not outputaddress_2.get_is_cm():
                                value_second = outputaddress_2.get_value()

                                value_full = value_first + value_second

                                if value_minimum <= value_full <= value_maximum\
                                        and outputaddress_2.get_addresshash() != outputaddress
                                            .get_addresshash():
                                    participation_list.append(
                                        ParticipationPossibility([transaction, transaction_2],
                                                                 [outputaddress, outputaddress_2
                                                                     ],
                                                                 output_range,
                                                                 2)
                                    )
    return participation_list

@staticmethod
def find_possibilities_3_forwards(inputtransaction_list, output_range):

    value_minimum = output_range[0]
    value_maximum = output_range[1]
    participation_list = [] # list with possible participation
```

```
    for transaction in inputtransaction_list:
        outputaddresses = transaction.get_outputaddress_list()
        for outputaddress in outputaddresses:
            outputaddress.mixer_results_load(transaction.get_hash())
            if not outputaddress.get_is_cm():
                value_first = outputaddress.get_value()
                if value_first < value_maximum:

                    # second iteration:
                    for transaction_2 in inputtransaction_list:
                        outputaddresses_2 = transaction_2.get_outputaddress_list()
                        for outputaddress_2 in outputaddresses_2:
                            outputaddress_2.mixer_results_load(transaction_2.get_hash())
                            if not outputaddress_2.get_is_cm():
                                value_second = outputaddress_2.get_value()
                                if (value_first+value_second) < value_maximum:

                                    # third iteration
                                    for transaction_3 in inputtransaction_list:
                                        output_addresses_3 = transaction_3.
                                             get_outputaddress_list()
                                        for outputaddress_3 in output_addresses_3:
                                            outputaddress_3.mixer_results_load(transaction_3.
                                                 get_hash())
                                            if not outputaddress_3.get_is_cm():
                                                value_third = outputaddress_3.get_value()
                                                value_full = value_first + value_second +
                                                     value_third

                                                addressset = { # an efficently way to check if
                                                      addresses different
                                                    outputaddress.get_addresshash(),
                                                    outputaddress_2.get_addresshash(),
                                                    outputaddress_3.get_addresshash()
                                                }

                                                if value_minimum <= value_full <=
                                                     value_maximum\
                                                        and len(list(addressset)) == 3:
                                                    participation_list.append(
                                                        ParticipationPossibility(
                                                            [transaction, transaction_2,
                                                                 transaction_3],
                                                            [outputaddress, outputaddress_2,
                                                                 outputaddress_3],
                                                            output_range, 3)
                                                    )
    return participation_list


# =====================
# Deanonymizer
# =====================
class Deanonymizer:

    """
    This class tries to deanonymize Coinmixer.SE-transactions by providing
    possible input/out-transactions that belong to the (un)tainted coins

    """
    def __init__(self):
        raise Exception("Should not be initialized")
```

```
addressFee = 60000 # address-fee specified in Coinmixer.SE-FAQ
feeRange = [1, 3] # minimum and maximum service-Fee taken by Coinmixer.SE (specified in CM
     .SE-FAQ)

@staticmethod
def input_deanonymize(inputtransaction_hash, cm_address=None, forwards=3, max_delay=120):
    """
        This function takes the customers input-transaction to Coinmixer.SE (tainted coins
             ) and maps it
        to output-transactions from Coinmixer.SE (untainted coins).
        It tries to find the untainted coins of a given customer-transaction.
    :param inputtransaction_hash: The transaction_hash of the transaction from the
         customer to Coinmixer.SE
    :param cm_address: The Bitcoin-Address of Coinmixer.SE which can be found in the
         specified transaction.
    :param forwards: The maximum amount of forwards that should be checked
    :param max_delay: The maximum delay that should be checked
    :return:
    """

    if cm_address is None or inputtransaction_hash is None: # todo: cm-address could be
         identified by program
        return False

    inputtransaction = Transaction(inputtransaction_hash)
    if DEBUG:
        print("input transaction loaded.")
    address_list = inputtransaction.get_outputaddress_list() # load address_list of
         outputs
    if DEBUG:
        print("address_list loaded.")

    found = False
    output_range = []
    for address in address_list:
        if address.get_addresshash() == cm_address:
            found = True # cm_address provided by user has been found in transaction
            sent_value = address.get_value()
            output_range = Deanonymizer.cm_out_ranges_calculate(sent_value)

    if found is False:
        return False # provided address could not be found in transaction

    container = ParticipationPossibilityContainer(inputtransaction, output_range)

    return container.find_possibilities(forwards, max_delay)

@staticmethod
def cm_out_ranges_calculate(input_value):
    """
        Calculates the possible minimum and maximum value
        which could be sent by Coinmixer.SE to the customer
        The last four digits of the customers input-transactions are probably random.
        The smallest fee taken by Coinmixer.Se is 1%
        The highest fee taken by Coinmixer.Se is 3%
        Calculations:

        highest Value: (valueSent - addressFee*number of forwards) * 0.99 with last four
             digits of valueSent as 9
        smallest Value: (valueSent - addressFee*number of forwards) * 0.97 with last four
             digits of valueSent as 0
```

```
        :param input_value: Value sent by customer to Coinmixer.se
        :return: list of possible minimum/maximum (non-negative) values for one to five
             forwards
        """

        output_range = []

        for forwards_number in range(1, 6): # number of forwards (1-5)
            tmp_value_smallest = (input_value - Deanonymizer.addressFee * forwards_number)

            value_smallest = int((tmp_value_smallest - tmp_value_smallest % 10000) * 0.97)
            value_highest = int(((tmp_value_smallest - tmp_value_smallest % 10000) + 9999) *
                 0.99)

            if value_smallest < 0 or value_highest < 0:
                value_smallest = 0
                value_highest = 0

            if 0 < value_smallest < 100000: # coinmixer-minimum/maximum per forward-address
                value_smallest = 100000

            if 0 < value_highest < 100000:
                value_highest = 100000

            if value_highest > 500000000:
                value_highest = 500000000

            output_range.append([value_smallest, value_highest]) # index 0 -> 1 forward, index
                  1 -> 2 forwards, ...
        return output_range

    @staticmethod
    def output_deanonymize():
        """
            This functions should be able to map the untaint coins to the customers input-
                 transaction


        :return:
        """
        return False


# ====================
# Partition
# ====================
class Partition:
    """
        The fees of Coinmixer.SE-transactions are static for a individual timeframe.
        We call this timeframe "partition". This class checks, updates and handles these
             timeframes.
        Furthermore it shows if transactions are within a partition and shows if the
             transaction likely belongs
        to the Coinmixer.SE-network.
        All timestamps are unix-second-based.
    """

    def __init__(self, partition_id, time_start, time_end, fee):
        self._id = partition_id
        self._timeStart = time_start
        self._timeEnd = time_end
```

```
        self._fee = fee

    def is_in_partition(self, timestamp):
        """
            This function checks if a specified timestamp (unix-second-based) is withtin this
                partition.
        :param timestamp: The timestamp to check.
        :return: True if timestamp is within the partion. False is not.
        """
        if DEBUG:
            print ("Check if timestamp in partition. partition_id: " + str(self._id) + "
                timestamp: " +
                  str(timestamp))

        if self._timeStart <= timestamp <= self._timeEnd:
            return True
        else:
            return False

    def is_timestamp_newer(self, timestamp):
        """
            Checks if timestamp is newer than partition
        :param timestamp: timestmap to check.
        :return: True if timestamp is newer than partition. False if not.
        """
        if DEBUG:
            print ("Check if timestamp is newer then partition. partition_id: " + str(self._id
                ) + " timestamp: " +
                  str(timestamp))

        if timestamp > self._timeEnd:
            return True
        return False

    def is_timestamp_older(self, timestamp):
        """
            Checks if timestamp is older than partition
        :param timestamp: timestamp to check
        :return: True if timestamp is older than partition. False if not.
        """
        if DEBUG:
            print ("Check if timestamp is older then partition. partition_id: " + str(self._id
                ) + " timestamp: " +
                  str(timestamp))

        if timestamp < self._timeStart:
            return True
        return False

    def get_fee(self):
        return self._fee

    def get_time_start(self):
        return self._timeStart

    def get_time_end(self):
        return self._timeEnd

    def get_id(self):
        return self._id
```

```
# ====================
# PartitionContainer
# ====================
class PartitionContainer:
    """
        The PartitionContainer holds all partitions-objects.
        The Container is able to insert new partitions into the Mysql-Database, update
            existing partitions and
        provides the fee and its reliability (trustlevel) for a given timestamp (unix-seconds-
            based)
    """

    partitions = [] # holds all partitions

    def __init__(self):
        raise Exception("Should not be initialized")

    @staticmethod
    def partition_insert(fee, timestamp_start, timestamp_end=None):
        """
            Creates new partitions and inserts it into Mysql-DB
        :param fee: The fee of the partition.
        :param timestamp_start: Startingpoint of the partition
        :param timestamp_end: Endingpoint of the partition
        :return:
        """

        if timestamp_end is None:
            timestamp_end = timestamp_start

        if DEBUG:
            print ("New partition inserted. fee: " + str(fee) + " timestamp_start: " +
                   str(timestamp_start) + " timestamp_end:" + str(timestamp_end))

        sql = "INSERT INTO fee_partition (fee,timestamp_start, timestamp_end) " \
              "VALUES("+str(fee)+","+str(timestamp_start) + "," + str(timestamp_end) + ")"
        Database.sql_execute(sql)

    @staticmethod
    def partition_timerange_update(partition_id, timestamp_start=None, timestamp_end=None):
        """
            Updates an existing partition. Extends/Shortens the span of a partiton
        :param partition_id: The ID of the partition which should get updated.
        :param timestamp_start: The new startingpoint. Startingpoint wont get updated if not
            provided.
        :param timestamp_end: The new endingpoint. Endingpoint wont get updated if not
            provided.
        :return:
        """
        if timestamp_start is None and timestamp_end is None:
            return False
        sql = ""

        if timestamp_start is None:
            sql = "UPDATE fee_partition SET " \
                  "timestamp_end = "+str(timestamp_end) + " WHERE partition_id = " + str(
                      partition_id)

        if timestamp_end is None:
            sql = "UPDATE fee_partition SET " \
                  "timestamp_start = "+str(timestamp_start) + " WHERE partition_id=" + str(
                      partition_id)
```

```
    if DEBUG:
        print ("Partition updated. partition_id: " + str(partition_id) + " timestamp_start
             : " +
               str(timestamp_start) + " timestamp_end:" + str(timestamp_end))

    Database.sql_execute(sql)
    return True

@staticmethod
def partition_load():
    """
        Loads partitions from database.
    :return:
    """

    sql = "SELECT partition_id, fee, timestamp_start, timestamp_end FROM fee_partition"
    result = Database.sql_execute(sql)
    PartitionContainer.partitions = [] # empty current partitions

    for partition in result: # update Partitions
        PartitionContainer.partitions.append(Partition(partition[0], partition[2],
             partition[3], partition[1]))
    if DEBUG:
        print ("Partitions loaded.")

@staticmethod
def get_fee_and_trustlevel(timestamp, return_partition=False):
    """
        This function returns the fee to a given timestamp, based on the fee-partitions.
        Furthermore it shows how reliable the results are (trustlevel). A higher
             trustlevel shows a better
        trustworthiness of the provided fee-result.

        Trustlevels:

        0 ->   Timestamp is not in any partition. The timestamp is ahead or behind
             existing partitons.
               This is the normal case for forward-crawling. The fees can and will change
                    while crawling.
               The result-fee can be an indicator for a transaction which is within the
                    Coinmixer-network, but
               shouldnt be fully relied on.

        1 ->   Timestamp is between two partitions (gap). Usually the fee of a transaction
             , which is between
               two partitions, equals the fee of one of these partitions. However this
                    indicator is only usable if
               gaps are not to big. Whenever fees abruptly change, this indicator will
                    fail.

        2 ->   Timestamp is in a partition. In most cases this means that the fee of a
             transaction has to be the
               same (+/-variance) as the partition-fee if the transaction is in the
                    Coinmixer.SE-network.
               However this indicator could be wrong whenever the fees of cm.se-
                    transactions change rapidly

    :param timestamp: The input timestamp. Typically the timestamp of a transaction which
         should be checked.
    :param return_partition: If True, the whole partition(s) and trustlevel are returned.
                         If False, only Fee and trustlevel are returned.
    :return:
```

```
"""
if DEBUG:
    print ("Getting fees and trustlevels. timestamp: " + str(timestamp) + "
         return_partition: " +
          str(return_partition))

list_of_earlier_partitions = []
list_of_newer_partitions = []

for partition in PartitionContainer.partitions:
    if partition.is_in_partition(timestamp) is True:
        if return_partition is True:
            return [partition, 2] # timestamp is in partition
        return [partition.get_fee(), 2]

    # divide partition in two groups: earlier (before timestamp) and later (after
         timestamp) partitions

    if partition.is_timestamp_newer(timestamp):
        list_of_earlier_partitions.append(partition)
    if partition.is_timestamp_older(timestamp):
        list_of_newer_partitions.append(partition)
    # partitions are diveded in two groups:
    # [earlier Partitions] timestamp [later partitions]
    # however each group is not orderd
if not list_of_earlier_partitions and not list_of_newer_partitions: # no partition in
     database
    return None

# find partition right before timestamp
latest_possible_partition = Partition(None, None, 0, None) # default-partition

for partition in list_of_earlier_partitions:
    if partition.is_timestamp_newer(latest_possible_partition.get_time_end()) is False
         :
        latest_possible_partition = partition # updates whenever a newer partition is
             found

if not list_of_newer_partitions: # if no partition after timestamp found, the
     timestamp is newest:
    if return_partition is True:   # [earlier partition] timestamp
        return [latest_possible_partition, 0]
    return [latest_possible_partition.get_fee(), 0]

# find first partition of "later partitions"-group
earlist_possible_partition = list_of_newer_partitions[0]
for partition in list_of_newer_partitions:
    if partition.is_timestamp_older(earlist_possible_partition.get_time_start()) is
         False:
        earlist_possible_partition = partition # updates whenever the start-timestamp
             of an partition is older
        # then the starttimestamp of the partition which is currently earliest

if latest_possible_partition.get_time_end() == 0: # if its still the default-partition
      there is no earlier
    if return_partition is True:            # partition then the timestamp -> timestamp
          newer then every
        return [earlist_possible_partition, 0] # partition: timestamp [later partition]
    return [earlist_possible_partition.get_fee(), 0]

# timestamp is in the gap between two partitions: [earlier partition] timestamp [later
      partition]
```

```
        if return_partition is True:
            return [[latest_possible_partition, earlist_possible_partition], 1]
        return [[latest_possible_partition.get_fee(), earlist_possible_partition.get_fee()],
             1]


# ====================
# Analyzer
# ====================
class Analyzer:
    """

    """
    def __init__(self):
        raise Exception("Should not be initialized")

    ret = []
    bla = 0

    @staticmethod
    def cm_check_transaction_fee_correct_partition_and_update(transaction, force=False):
        """
            Check if fee is in correct partition and update partition if necessary (e. g.
                 update last timestamp).
            Automatically updates if CMcheckTransaction_feeInCorrectParition returns 1 or 2
            Updates are always applied if transaction lies within a gap:
            [earlier partitions] transaction [later partition]
            If force=True updates are also applied if the transaction is after or before
                 partitions:
            [earlier partition] transaction
            transaction [later partition]
        :param transaction: The transaction which should be checked.
        :param force: If transaction is newer/older then every partition, updates are only
             applied if force == True
        :return:
        """
        result = Analyzer.cm_check_transaction_fee_correct_partition(transaction)

        if result is None:
            PartitionContainer.partition_insert(transaction.get_fee_per_byte(), transaction.
                 get_time())
            return True

        PartitionContainer.partition_load()
        result = PartitionContainer.get_fee_and_trustlevel(transaction.get_time(), True) #
             return:
        #  [parition(s), trustlvl]

        if DEBUG:
            print ("updating Partition is necessary. force=" + str(force))

        if type(result[0]) is list: # transaction in in gap: [earlier partitions] transaction
             [newer partitions]

            for partition in result[0]:
                if abs(transaction.get_fee_per_byte() - partition.get_fee()) <= 1: # coinmixer.
                     SE-fee-Variance
                    # = 1 Sat/Byte
                    if transaction.get_time() > partition.get_time_end(): # update last
                         timestamp of earlier partition
                        # if timestamp is newer then last timestamp of returned partition
                        # (earlier partition gets updated)
```

```
                PartitionContainer.partition_timerange_update(partition.get_id(), None,
                     transaction.get_time())
                return True
            elif transaction.get_time() < partition.get_time_start(): # update first
                # timestamp of newer partition

                # if timestamp is older then first timestamp of returned partition
                # (newer partition gets updated)
                PartitionContainer.partition_timerange_update(partition.get_id(),
                     transaction.get_time(), None)
                return True

    else: # transaction is newer/older then every partition or lays within a partition
        partition = result[0]
        if abs(transaction.get_fee_per_byte() - partition.get_fee()) <= 1:
            if result[1] == 2: # transaction is in partition - no update required
                return True
            elif result[1] == 0 and force is True: # partition newer/older then every
                 partition
                # updates only applied if force == True
                if transaction.get_time() < partition.get_time_start():
                    PartitionContainer.partition_timerange_update(partition.get_id(),
                         transaction.get_time(), None)
                    return True
                elif transaction.get_time() > partition.get_time_end():
                    PartitionContainer.partition_timerange_update(partition.get_id(), None,
                         transaction.get_time())
                    return True

    return False

@staticmethod
def cm_check_transaction_all_inputs_cm():
    """
    Should check if all input-addreses are owned by Coinmixer.SE
    (not implemented yet)
    :return:
    """
    return False

@staticmethod
def cm_results_insert(outputadresses, transaction):
    """
        Saves checking-results (output-addresses) to database.
    :param outputadresses: outputaddresses which should be inserted in database
    :param transaction: transaction-obejct which belongs to outputaddresses
    :return:
    """
    if isinstance(outputadresses, list) is False: # outputAddresses-variable could be a
         single output-address
        tmp = list()
        tmp.append(outputadresses)
        outputadresses = tmp

    for output in outputadresses:

        sql_fee = output.get_is_cm_fee()
        sql_common = output.get_is_cm_common_value()
        sql_version_sequence_locktime = output.get_is_cm_version_sequence_locktime()
        sql_is_cm = output.get_is_cm()
        sql_is_spent = output.get_is_cm_spent()
        sql_is_cashin = output.get_is_cm_cashin_address()
```

```
        sql_next_transaction = output.get_next_transaction()
        sql_history = output.get_is_cm_transaction_history()

        if sql_fee is None:
            sql_fee = "NULL"

        if sql_common is None:
            sql_common = "NULL"

        if sql_version_sequence_locktime is None:
            sql_version_sequence_locktime = "NULL"

        if sql_is_cm is None:
            sql_is_cm = "NULL"

        if sql_is_spent is None:
            sql_is_spent = "NULL"

        if sql_is_cashin is None:
            sql_is_cashin = "Null"

        if sql_next_transaction is None:
            sql_next_transaction_hash = ""
        else:
            sql_next_transaction_hash = sql_next_transaction.get_hash()

        if sql_history is None:
            sql_history = "NULL"

        # insert results to database
        sql = "INSERT INTO coinmixer_analysis" \
              "(" \
              "address_hash, next_transaction_hash," \
              "transaction_hash, forward, fee, common, version_sequence_locktime, " \
              "connected, spent, is_cm, is_cashin, transaction_history) VALUES " \
              "('" + output.get_addresshash() + "','" + sql_next_transaction_hash + "','" +
                    \
              transaction.get_hash() + "'," + str(transaction.get_forward()) + "," + \
              str(sql_fee) + "," + str(sql_common) + "," + \
              str(sql_version_sequence_locktime) + ", True," + str(sql_is_spent) + "," + \
              str(sql_is_cm) + ","+str(sql_is_cashin)+", "+str(sql_history)+")"
        Database.sql_execute(sql)

    @staticmethod
    def cm_log_check_is_cashin(transaction_hash):
        """
            Checks whether a transaction_hash is already saved in analysis-results and if its
            a transaction done by customers
        :param transaction_hash: Hash of the transaction which should be checked
        :return:
        """

        sql = "SELECT count(analysis_id) FROM coinmixer_analysis WHERE " \
             "transaction_hash='"+str(transaction_hash) + "' and is_cm=0"
        res = Database.sql_execute(sql)[0][0]
        if res == 0:
            return False
        else:
            return True

    @staticmethod
    def cm_log_check_is_errorhash(transaction_hash):
```

```
    """
            Checks whether an error occured while proccessing this transactio in the past.
            If an error occured, this may have an effect on the further proccessing (
                endless-loop )
    :param transaction_hash:
    :return:
    """
    sql = "SELECT count(ID) FROM errorlog WHERE transaction_hash ='" + transaction_hash +
         "'"
    res = Database.sql_execute(sql)[0][0]
    if res == 0:
        return False
    else:
        return True

@staticmethod
def cm_log_check_is_first(previous_hash):
    """
        Checks if prevoius_hash is the first cm-network transaction
        (prior transactions are customer-intputtransactions)
    :param previous_hash:
    :return: True -> previous_transaction is the first cm-network transaction
             False-> there are cm-network transaction prior to this transaction
    """
    sql = "SELECT count(is_first) FROM coinmixer_analysis_log WHERE previous_hash='"+
         previous_hash+"'"
    result = Database.sql_execute(sql)
    if result[0][0] == 0:
        return False
    else:
        return True

@staticmethod
def cm_log_check(transaction_hash, forward=False):
    """
        Recursive function. Follows all next-transaction-hashes (forward=True) till no
             next-transaction-hash could
        be found. (Returns last state of previous forward-crawling-process)
        Follows all previous-transaction-hashes (forward=False) till no more previous-
             transaction-hash could be
        found. (Returns last state of previous backwards-crawling-process)
    :param transaction_hash: The transaction-Hash from which the crawling should start
    :param forward: True -> forward-crawling. False -> backwards-crawling
    :return:
    """
    ret = []
    sql = ""
    if forward is True:
        sql = "SELECT next_hash FROM coinmixer_analysis_log WHERE " \
              "transaction_hash = '" + transaction_hash + "' and previous_hash = ''"

    if forward is False:
        sql = "SELECT previous_hash, is_first FROM coinmixer_analysis_log WHERE " \
              "transaction_hash = '" + transaction_hash + "' and next_hash=''"
    result = Database.sql_execute(sql)

    # check: transaction has not been processed before and no error occured and its not a
         customers transaction
    if not result and not Analyzer.cm_log_check_is_first(transaction_hash) and\
            not Analyzer.cm_log_check_is_errorhash(transaction_hash):

        return [transaction_hash]
```

```
    else:
        for res in list(set(result)): # multitple list-elements are removed
            ret += Analyzer.cm_log_check(res[0], forward) # recursion
    return ret

@staticmethod
def cm_log_update_first(transaction_hash):
    """ This function should only be used with backward-crawling.
        If an transaction is the the first transaction in the cm-network
        (prior transaction is inputtransaction by customer) this function updates the
             coinmixer_analysis_log
        table for better performance (not neccessary checks of first-transactions will be
             done)
    :param transaction_hash:
    :return:
    """
    sql = "UPDATE coinmixer_analysis_log SET is_first=1 WHERE previous_hash = '"+
         transaction_hash+"'"
    Database.sql_execute(sql)

@staticmethod
def cm_log_insert(transaction_hash, next_hash=None, previous_hash=None, depth=0,):
    """
        Inserts new Transaction in Coinmixer-Log. Logs transactions that have been
             analyzed so they dont have to
        be checked again.
    :param transaction_hash: Transaction hash that have been analyzed
    :param next_hash: Hash of transaction that follows analyzed transaction (typically
         forward-crawling)
    :param previous_hash: Hash of transaction that is ahead of analyzed transaction (
         typically backwards-crawling)
    :param depth: The depth of recusion (only applied on backwards-crawling)
    :return:
    """
    if next_hash is None and previous_hash is None:
        return False

    if next_hash is None:
        next_hash = ""

    if previous_hash is None:
        previous_hash = ""

    sql = "INSERT INTO coinmixer_analysis_log (transaction_hash, previous_hash, next_hash,
          depth) " \
          "VALUES ('" + transaction_hash + "','" + previous_hash + "','" + next_hash + "',"
                + str(depth) + ")"

    Database.sql_execute(sql)

@staticmethod
def cm_check_transaction_fee_correct_partition(transaction):
    """
        Checks if fee is correct (based on partitions).

    :param transaction: Transaction which is checked
    :return:    None -> No partition found
                0 -> transaction-fee probably wrong,
                1 -> transaction-fee probably correct but can and will likely change in
                     future
                (transaction is older or newer then partitions)
```

```
                2 -> transaction-fee should be correct (transaction between gap or in
                     partition)
    """
    if DEBUG:
        print ("checking fee of transaction: " + transaction.get_hash())
    fee = transaction.get_fee()
    size = transaction.get_size()
    fee_per_byte = int(fee/size)

    PartitionContainer.partition_load()

    result = PartitionContainer.get_fee_and_trustlevel(transaction.get_time()) # return: [
         fee, trustlvl]
    if DEBUG:
        print ("fee-check result: " + str(result))

    if result is None:
        return None

    if type(result[0]) is list: # transaction in gap
        for correct_fee in result[0]:
            if abs(fee_per_byte - correct_fee) <= 1:
                return 2
    else:
        correct_fee = result[0]
        sec_level = result[1]
        if abs(fee_per_byte - correct_fee) <= 1:
            if sec_level == 0: # transaction ahead or before partitions
                return 1
            else:
                return 2 # transaction in partition

    return 0 # transaction-fee not correct

@staticmethod
def cm_check_address_version_sequence_locktime(address):
    """
        Checks whether version, sequence, locktime of the next transaction send by the
             address are correct.
        If version, sequence or locktime is wrong or more then one transaction is sent
             through the address
        its likely not an address controlled by Coinmixer.SE.
    :param address: Address to check
    :return: True -> Version, Sequence, Locktime correct (could be a Coinmixer.SE-
         transaction)
             False -> Version, Sequence, locktime wrong (cant be a Coinmixer.SE-transaction
                 )
    """

    sent_counts = address.sent_counts() # check how many transactions have been sent
         through this address

    if sent_counts != 1: # Coinmixer.SE-addresses typically dont send more then one
         transaction
        return False

    return Analyzer.cm_check_transaction_version_sequence_locktime(address.
         get_transactions()[0]) # check next
    #  transaction

@staticmethod
def cm_check_transaction_transaction_outputs(transaction, expected_outputs_int):
```

```
    """
        Check if the number of Outputs of the address is expected.
        Typically two outputs are expected (customer, and coinmixer.se-Network)

    :param transaction: transaction to check.
    :param expected_outputs_int: expected outputs (typically two).
    :return:
    """

    transactions = transaction.get_outputaddress_list()
    if len(transactions) > expected_outputs_int:
        return False
    return True

@staticmethod
def cm_check_transaction_common_value_backward(transaction, cm_address):
    """
        Checks whether an address has been RECEIVING an commonValue (typically this
             addresses are owned by costumer)
        or non-common values (typically this addresses are owned by coinmixer.se) in the
             provided transaction.
        A common value is a value thats specified up to five decimal places (e.g.
             0.57312000).
        All values are based in satoshis.
        This function is used for backwards-crawling. Its checks if its likely that
             cmAddress is an
        cashin-Address (used by customers to cashin to coinmixer.SE) or its an address
             which is used for outputs
        (paying customers)
    :param transaction: Transaction which should be a Coinmixer.SE-transaction.
    :param cm_address: An output-address of the transaction that is probably owned by
         Coinmixer.SE
    :return:
    """

    outputaddresses = transaction.get_outputaddress_list()
    for outputAddress in outputaddresses:
        if outputAddress.get_addresshash() != cm_address.get_addresshash(): # typicalle
             the second output-address
            if outputAddress.get_value() % 1000 == 0: # last three decimal places equals
                 zero
                return True
            else:
                return False
    return True # default (e.g. only 1 output-address)

@staticmethod
def cm_check_transaction_version_sequence_locktime(transaction):
    """
        Checks whether version, sequence and locktime are correct for coinmixer.SE-
             transactions
    :param transaction: Transaction that should be checked.
    :return:
    """
    if transaction.get_version() == 2 and transaction.get_locktime() > 0 \
            and transaction.get_sequence() == 4294967294:
        return True
    return False

@staticmethod
def cm_check_address_common_value(address):
    """
```

```
            Checks whether an address has SENT an commonValue (typically this addresses are
                owned by costumer)
            or non-common values (typically this addresses are owned by coinmixer.se)
            A common value is a value thats specified up to five decimal places (e.g.
                0.57312000).
            All values are based in satoshis.
            This function is used for forward-crawling. Its checks if its likely that the
                address is a customer-address.

        :param address:
        :return:
        """
        if address.get_value() % 1000 == 0:
            return True
        else:
            return False

    @staticmethod
    def cm_check_address_transaction_count(address, inputtransaction):
        """
                Checks whether the address has been sending transaction before the input-
                    transaction.
                Coinmixer-Addresses typically do only send a single transaction.
        :param address: The address to check.
        :param inputtransaction: the known input-transaction.
        :return:   0 -> input-Transaction is not the first transaction of address or there
            have been
                    more then one send-transaction
                    -> address MOST PROBABLY NOT controlled by Coinmixer.SE

                1 -> address has unspent outputs
                    -> address PROBABLY NOT controlled by Coinmixer.SE

                2 -> address has no unspent outputs and inputtransaction is the first and
                    only spent output
                   -> address IS PROBABLY controlles by Coinmixer.SE
        """

        if address.first_transaction_timestamp() < inputtransaction.get_time(): # checks
            whether inputtransaction
            return 0
        sent_counter = address.sent_counts()

        if sent_counter > 1: # checks whether there have been multiple send-transactions
            return 0
        if address.get_spent() is False: # checks whether addess-value is unspent
            return 1 # unspent output
        else:
            return 2 # spent output


# ====================
# Mapping
# ====================
class Mapping:
    """
        The database-structure saves address-hashes and values (spent-outputs) in seperated
            tables.
        This class provides a mapping between address-hashes and values.
        Values and addresses are mapped through their indices (e.g. addr1 and val1 belong
            together)
    """
```

```
    def __init__(self):
        self._addressList = [] # list of addresses [addr1, addr2, ...]
        self._value_list = [] # list of values [val1, val2, ...]
        self._id = None

    def entry_add(self, address, value):
        self._addressList.append(address)
        self._value_list.append(value)

    def mapping_insert(self):
        """
            Inserts mapping into database
        :return:
        """
        sql = "INSERT INTO address_and_value_mapping (address_list, value_list) VALUES " \
              "('"+json.dumps(self._addressList)+"', '"+json.dumps(self._value_list) + "')"
        Database.sql_execute(sql)
        sql = "SELECT LAST_INSERT_ID()"
        self._id = Database.sql_execute(sql)[0][0]

    def get_id(self):
        return self._id


# =====================
# Address
# =====================
class Address:
    """
        Represents a Bitcoin-address with its general Bitcoin-address-information
        and specific Coinmixer-information.
    """

    API_URL = "https://blockchain.info/rawaddr/"

    def __init__(self, addresshash, value=None, spent=None, is_inputaddress=None, sequence=
         None):
        self._addresshash = addresshash
        self._value = value
        self._is_inputaddress = is_inputaddress # is input-address -> True, is output-address
             -> false
        self._sequence = sequence
        self._spent = spent
        self._transactions = []
        self._nextTransaction = None
        self._SentTransactionsInt = 0
        self._received_transactions_int = 0
        self._is_cm = None
        self._is_cm_transaction_history = None
        self._is_cm_fee = None
        self._is_cm_version_sequence_locktime = None
        self._is_cm_common_value = None
        self._is_cm_spent = None
        self._is_cm_counter = 0
        self._is_cm_cashin_address = None
        self._is_cm_forward = None
        self._previous_transaction = None
        self._is_cm_connected = None
        self._address_id = None
        self._value_id = None
```

```
def inputtransaction_find(self):
    """
        If its an address controlled by Coinmixer.SE there should
        only exist one input-transaction to an address. This function returns this
            transaction.
    :return:
    """
    if DEBUG:
        print("find inputtransaction")
    counter = 0
    inputtransaction = None
    # Find the transaction where this address (self) is an output-address.
    # It has to output the whole value (get_value()), otherwise this address sent multiple
         transactions.
    for transaction in self._transactions:
        outputaddress_list = transaction.get_outputaddress_list()
        for outputaddress in outputaddress_list:
            if outputaddress.get_addresshash() == self.get_addresshash()\
                    and outputaddress.get_value() == self.get_value():
                inputtransaction = transaction
                counter += 1

    if counter > 1: # shouldnt be possible (double-spent)
        ErrorLog.log("Error occured at input-address: " + self.get_addresshash() + " (
            multiple spends of value!)",
                    None, self.get_addresshash())
        return False

    return inputtransaction

def mixer_results_load(self, transaction_hash):
    """
        Loads analysis-results of a specified transaction-hash into local variables.
    :param transaction_hash: Hash of which results will be loaded
    :return:
    """
    sql = "SELECT forward, fee, common, version_sequence_locktime,connected, spent, " \
          "is_cm, is_cashin, transaction_history FROM coinmixer_analysis WHERE " \
          "address_hash = '"+self._addresshash+"' and transaction_hash='" +
              transaction_hash + "' limit 0,1"
    result = Database.sql_execute(sql)
    if len(result) != 1: # there has to be exactly one analysis-result
        return False

    data = result[0]
    self._is_cm_forward = bool(data[0])
    self._is_cm_fee = data[1]
    self._is_cm_common_value = data[2]
    self._is_cm_version_sequence_locktime = data[3]
    self._is_cm_connected = data[4]
    self._is_cm_spent = data[5]
    self._is_cm = data[6]
    self._is_cm_cashin_address = data[7]
    self._is_cm_transaction_history = data[8]

    return True

def is_cm_counter_add(self, add):
    """
        Whether an addrss is controlled by a customer (customer-address) or by coinmixer (
            coinmixer-address)
        is based on this CM-counter.
```

```
        Based on this counter its chosen which address is controlled by coinmixer (higher
             CM-counter) or
        by customer (lower CMcounter).
    :param add: Adds an int to counter.
    :return:
    """
    self._is_cm_counter += add

def get_is_cm_counter(self):
    """
    Returns CM-Counter (see is_cm_counter_add(add)) for more information
    :return:
    """
    return self._is_cm_counter

def set_is_cm_counter(self, is_cm_counter):
    self._is_cm_counter = is_cm_counter

def set_is_cm(self, is_cm):
    """
        Sets if the address is controlled by Coinmixer.SE
    :param is_cm:
    :return:
    """
    self._is_cm = is_cm

def set_next_transaction(self, next_transaction):
    """
        Sets next transaction that was sent by Coinmixer.SE (forward-crawling)
    :param next_transaction:
    :return:
    """
    self._nextTransaction = next_transaction

def get_next_transaction(self):
    """
        Returns next transactions that was sent by Coinmixer.SE (forward-crawling)
    :return:
    """
    return self._nextTransaction

def get_is_cm(self):
    """
        Getter. For more informatin check set_is_cm(is_cm).
    :return:
    """
    return self._is_cm

def set_is_cm_fee(self, cm_fee):
    """
        Sets if the fee of the transaction sent by this address is correct for an
             transaction send by Coinmixer.SE
    :param cm_fee:
    :return:
    """
    self._is_cm_fee = cm_fee

def get_is_cm_fee(self):
    """
        Getter. For more information check set_is_cm_fee(CMfee).
    :return:
    """
```

```
        return self._is_cm_fee

    def set_is_cm_transaction_history(self, cm_history):
        """
            0 -> This address sent multiple transactions -> address MOST PROBABLY NOT
                controlled by Coinmixer

            1 -> Output is unspent -> address PROBABLY NOT controlled by Coinmixer

            2 -> Output is spent -> address COULD BE controlled by Coinmixer
        :param cm_history:
        :return:
        """
        self._is_cm_transaction_history = cm_history

    def get_is_cm_transaction_history(self):
        """
            Getter. For more information check set_is_cm_transaction_history(CMhistory)
        :return:
        """
        return self._is_cm_transaction_history

    def set_is_cm_version_sequence_locktime(self, version_sequence_locktime_bool):
        """
            True -> Version, Sequence, Locktime of next transaction indicates that this
                address is owned by coinmixer.SE
            False -> Version, Sequence, Locktime different from typical values set by
                coinmixer.SE
        :param version_sequence_locktime_bool:
        :return:
        """
        self._is_cm_version_sequence_locktime = version_sequence_locktime_bool

    def get_is_cm_version_sequence_locktime(self):
        """
        Getter. For more information check set_is_cm_version_sequence_locktime(verSegLock).
        :return:
        """
        return self._is_cm_version_sequence_locktime

    def set_is_cm_spent(self, cm_spent):
        """
            True: Address-values are spent
            False: Address-values are unspent (unusual for addresses controlled by Coinmixer.
                SE)
        :param cm_spent:
        :return:
        """
        self._is_cm_spent = cm_spent

    def get_is_cm_spent(self):
        """
        Getter. For more information check setis_cm_spent(CMspent).
        :return:
        """
        return self._is_cm_spent

    def set_is_cm_common_value(self, cm_common_bool):
        """
            True -> a common value has been sent through this address -> indicates costumer-
                address
```

```
        False -> an uncommon value has been sent through this address -> indicates
            coinmixer-address
    :param cm_common_bool:
    :return:
    """
    self._is_cm_common_value = cm_common_bool

def get_is_cm_common_value(self):
    """
    Getter. For more information check set_is_cm_common_value(CMcommon).
    :return:
    """
    return self._is_cm_common_value

def get_transactions(self):
    """
    Getter. For more information check transactions_load()
    :return:
    """
    return self._transactions

def transactions_load(self):
    """
        Loads every transaction of the address from Blockahin.info-API
        (todo: check if all transactions for an address are
        already saved in database and load them from there)
    :return:
    """
    if DEBUG:
        print ("loading all transactions")
    url = Address.API_URL + self._addresshash

    response = urllib.urlopen(url)
    try:
        data = json.loads(response.read())
        transaction_list = []
        for transactionData in data["txs"]:

            transaction_list.append(
                Transaction(
                    transactionData["hash"], None, None, None, None, None,
                    None, None, None, None, None, None, transactionData))

        self._transactions += transaction_list
        if DEBUG:
            print("all transactions have been loaded.")
    except ValueError:
        print("JSON-object could not be decoded. Probably your IP got blocked. Try again
            later.")
        exit(0)

def first_transaction_timestamp(self):
    """
        Returns the timestamp of the first transaction that has been sent/received through
            this address.
        Transactions have to be loaded before calling this function! (transactions_load)
    :return:
    """
    timestamp = self._transactions[0].get_time()
    for transaction in self._transactions:
        if transaction.get_time() < timestamp:
            timestamp = transaction.get_time()
```

```
    return timestamp

def sent_counts(self, return_transactions=False):
    """
        Returns the number of transactions that have been sent through this address
    :param return_transactions: True -> a list with transactions is appended to return-
         result
    :return:
    """
    sent_counter = 0
    transaction_list = []
    for transaction in self._transactions:
        for inputaddresses in transaction.get_inputaddress_list():
            if inputaddresses.get_addresshash() == self._addresshash:
                sent_counter += 1
                if return_transactions is True:
                    transaction_list.append(transaction)
    if return_transactions is True:
        return [sent_counter, transaction_list]

    return sent_counter

def get_addresshash(self):
    return self._addresshash

def get_sequence(self):
    """
        List of all sequences used in transactions???
    :return:
    """
    return self._sequence

def get_address_id(self):
    """
        Getter. For more information check set_address_id(address_id)
    :return:
    """
    return self._address_id

def set_address_id(self, address_id):
    """
        Values and addreses are connected through an mapping (see Mapping-Class). This is
             the id of the address
        which is saved in the database.
    :param address_id:
    :return:
    """
    self._address_id = address_id

def get_value_id(self):
    """
        Getter. For more information check set_value_id(value_id)
    :return:
    """
    return self._value_id

def set_value_id(self, value_id):
    """
        Values and addreses are connected through an mapping (see Mapping-Class). Value-ID
              maps an value to
        this address.
    :param value_id:
```

```
        :return:
        """
        self._value_id = value_id

    def get_value(self):
        """
        Getter. For more information check set_value_id(value_id).
        :return:
        """
        return self._value

    def is_inputaddress(self):
        """
            True -> address is loaded as an input-address to an transaction
            False -> address is loaded as an output-address to an transaction
        :return:
        """
        return self._is_inputaddress

    def get_spent(self):
        """
        Getter. Returns the value (satoshis) spent by this address
        :return:
        """
        return self._spent

    def get_previous_transaction(self):
        """
        Getter. For more information check set_previous_transaction(transaction)
        :return:
        """
        return self._previous_transaction

    def set_previous_transaction(self, transaction):
        """
            Sets previous input-transaction (backward-crawling)
            Typically its the first and only input of an address controlled by Coinmixer.SE.
        :param transaction: Transaction which will bet set a previous-transaction
        :return:
        """
        self._previous_transaction = transaction

    def get_is_cm_cashin_address(self):
        """
            True -> Address is an Coinmixer.SE-Address which is used by customers to use
                 Coinmixer.SE-Service
            False -> Address used by Coinmixer to forwards transactions
        :return:
        """
        return self._is_cm_cashin_address

    def set_is_cm_cashin_address(self, cashin_bool):
        """
            Getter. For more information check getis_cmCashinAddress().
        :param cashin_bool:
        :return:
        """
        self._is_cm_cashin_address = cashin_bool


# ====================
# Transaction
```

```
# ====================
class Transaction:
    """
        Represents a Bitcoin-transaction with its general
        Bitcoin-transaction-information and specific Coinmixer-information.
    """
    API_URL = "https://blockchain.info/de/rawtx/"

    def __init__(
            self, tx_hash, tx_id=None, blockheight=None,
            fee=None, size=None, time=None,
            version=None, sequence=None,
            locktime=None, inputaddress_list=None,
            outputaddress_list=None, is_cm=None, data=None):

        self._hash = tx_hash
        self._blockheight = blockheight
        self._fee = fee
        self._size = size
        self._time = time
        self._version = version
        self._sequence = sequence
        self._locktime = locktime
        self._inputaddress_list = inputaddress_list
        self._outputaddress_list = outputaddress_list
        self._is_cm = is_cm
        self._id = tx_id
        self._forward = True
        self._block_full = False # not used at the moment
        if DEBUG:
            print("new transaction-object has been created: " + self._hash)
        # todo: after loadFromDatabase() check if data could be loaded from normalsized/
             bigsized-transaction-table
        # todo: (change data structure of normalsized/bigsized-transaction-table previously if
              necessary)
        # todo: maybe the load_from_api(data) function can be used (data from normalsized/
             bigsized-table)
        # if an object is created and only the transaction-hash is passed,
        # the transaction will be loaded from the database or blockchain.info-API
        if all(parameter is None for parameter in [
            blockheight, fee, size, time,
            version, sequence, locktime,
            inputaddress_list, outputaddress_list,
            is_cm
        ]):

            if self.load_from_database() is False: # try to load transaction-data from mysql-
                 database
                if DEBUG:
                    print ("loading from database faild. Load data from API")
                self.load_from_api(data) # load data from blockchain.info

    def load_from_database(self):
        """
            Loads transaction-data from database (transaction_data-table).
        :return:
        """
        if DEBUG:
            print ("loading transaction-data from database")

        # Check if transaction is in transactionData
```

```
sql = "SELECT ID, in_size_big_table, in_transaction_data FROM
     list_of_all_transaction_hashes" \
      " WHERE transaction_hash ='" + self._hash + "'"
result = Database.sql_execute(sql)

if not result: # transaction-hash has never been seen before by this script
    if DEBUG:
        print ("couldnt load transaction-data. Not in database.")
    return False
if result[0][2] == 1: # transaction has been processed already and is saved to
     database

    sql = "SELECT * FROM transaction_data WHERE transaction_hash = '"+self._hash+"'"
    result = Database.sql_execute(sql)

    if len(result) == 1: # transaction_hash should be an unique-column
        result = result[0]
        self._id = result[0]
        self._blockheight = result[2]
        self._fee = result[3]
        self._size = result[4]
        self._time = result[5]
        self._version = result[6]
        self._sequence = result[7]
        self._locktime = result[8]
        inputaddress_mapping_id = result[9]
        outputaddress_mapping_id = result[10]
        self._is_cm = result[11]

        if self._sequence is None:
            sql = "SELECT sequences FROM multiple_sequences where transaction_id = " +
                 str(self._id)
            self._sequence = json.loads(Database.sql_execute(sql)[0][0])

        self._inputaddress_list, self._outputaddress_list = [], []
        # Load input-address_list-ids and valuelist-ids
        sql = "SELECT address_list, value_list from address_and_value_mapping" \
             " WHERE ID=" + str(inputaddress_mapping_id)
        result = Database.sql_execute(sql)[0]

        address_id_list = json.loads(result[0])
        value_id_list = json.loads(result[1])

        # Load inputaddresses for each address_id and value_id
        # intputaddress_list holds created address-objects of each input-address
        for index, value in enumerate(address_id_list):
            sql = "SELECT transaction_value,spent from transaction_values WHERE ID=" +
                 str(value_id_list[index])
            value_and_spent = Database.sql_execute(sql)[0]

            sql = "SELECT transaction_address from transaction_addresses" \
                  " WHERE ID=" + str(address_id_list[index])
            addresshash = Database.sql_execute(sql)[0][0]
            self._inputaddress_list.append(Address(addresshash, value_and_spent[0],
                 value_and_spent[1]))

        # Load output-address_list-ids and value_list-ids
        sql = "SELECT address_list, value_list from address_and_value_mapping" \
             " WHERE ID=" + str(outputaddress_mapping_id)
        result = Database.sql_execute(sql)[0]
        address_id_list = json.loads(result[0])
        value_id_list = json.loads(result[1])
```

```
            # load outputaddresses for each address_id and value_id
            # outputaddress_list holds created address-object of each output-address
            for index, value in enumerate(address_id_list):
                sql = "SELECT transaction_value,spent from transaction_values WHERE ID=" +
                     str(value_id_list[index])
                value_and_spent = Database.sql_execute(sql)[0]

                sql = "SELECT transaction_address from transaction_addresses " \
                      "WHERE ID=" + str(address_id_list[index])
                addresshash = Database.sql_execute(sql)[0][0]
                self._outputaddress_list.append(Address(addresshash, value_and_spent[0],
                     value_and_spent[1]))
            if DEBUG:
                print ("successfully loaded transaction-data")
            return True
    else:
        if DEBUG:
            print ("faild to load transaction-data."
                   "transaction has been found in database, but its in
                        transaction_size_normal"
                   " or transaction_size_big tables. Loading from these tables need to be
                        implemented.")
        return False

    if DEBUG:
        print ("faild to load transaction-data.")

    return False

def load_from_api(self, data=None):
    """
        Loads transaction-data from blockchain.info-API.
    :param data:
    :return:
    """
    if DEBUG:
        print("loading transaction-data from API")

    if data is None:
        url = Transaction.API_URL + self._hash
        response = urllib.urlopen(url)
        try:
            data = json.loads(response.read())
        except ValueError:
            print("JSON-object could not be decoded. Probably your IP got blocked. Try
                 again later.")
            exit(0)
    try:

        # copy data which can be copied without further processing
        self._locktime = data["lock_time"]
        self._version = data["ver"]
        self._time = data["time"]

        if "block_height" in data: # if blockheight is not accessible
            # (normally this should be an "unconfirmed transaction") -> set blockheight=1 (
                 default-value)
            self._blockheight = data["block_height"]
        else:
            self._blockheight = 1
```

```
        self._size = data["size"]
        self._block_full = False

        # create input-address-list, output-address-list, sequence(-list), fee
        self._inputaddress_list = []
        self._outputaddress_list = []
        sent_total = 0
        received_total = 0

        for inputaddress in data["inputs"]:
            prev_out = inputaddress["prev_out"]
            self._inputaddress_list.append(
                Address(
                    prev_out["addr"], prev_out["value"], prev_out["spent"], True,
                         inputaddress["sequence"]
                ))

            received_total += prev_out["value"] # total input-value of the transaction

        for output in data["out"]:
            self._outputaddress_list.append(
                Address(
                    output["addr"], output["value"], output["spent"], False
                ))

            sent_total += output["value"] # total spent-value of the transaction

        self._fee = received_total - sent_total

        if self._fee < 0: # should not be possible
            raise ValueError("Error: negative Fee. TX: " + self._hash)

        # get a list of all sequences and remove duplicates
        self._sequence = list(set([inputaddress.get_sequence() for inputaddress in self.
             _inputaddress_list]))

        if len(self._sequence) == 1: # if its only one element -> remove list
            self._sequence = self._sequence[0]

    except KeyError as e: # a key couldnt be found in json-object (e. g. transaction-size
         missing)
        print ("KeyError: reason " + str(e))
        exit(0)

    if DEBUG:
        print ("successfully loaded transaction-data from API.")
    self.save_to_database(data) # save data to data
    # base

def save_to_database(self, transaction_full_json):
    """
        Transaction is saved to database (Table: transactions_size_normal or
             bigSizedTransaction).
        Blockchain.info-api-calls should be reduced to a minimum (bottleneck), so every
             transaction will be stored
        in database.
        Transaction with an json-encoded length up to 40.000 Chars will be saved in "
             transactions_size_normal".
        Bigger transactions are saved in "transactions_size_big"
    :param transaction_full_json: The full transaction received by Blockchain.info-Api (
         JSON)
    :return:
```

```
    """
    if DEBUG:
        print("Saving transaction-data to database.")
    if self._blockheight is None or self._size is None:
        self.load_from_api()

    transaction_full_json_encoded = json.dumps(transaction_full_json)
    tablename = "transactions_size_normal"
    bigsized = False

    if len(transaction_full_json_encoded) > 40000: # could be up to 65535
        tablename = "transactions_size_big"
        bigsized = True

    sql = "INSERT IGNORE INTO " + tablename +\
          " (hash, blockheight, transaction, fullBlock) VALUES" \
          " ('"+self._hash+"',"+str(self._blockheight)+",'" + transaction_full_json_encoded
               + "',"\
          + str(self._block_full) + ")"
    Database.sql_execute(sql)

    sql = "SELECT LAST_INSERT_ID()"
    last_id = Database.sql_execute(sql)[0][0]

    sql = "INSERT IGNORE INTO list_of_all_transaction_hashes" \
          " (transaction_hash, ID, in_size_big_table, in_transaction_data)" \
          " VALUES ('"+self._hash+"',"+str(last_id)+","+str(bigsized) + ", 0)"
    Database.sql_execute(sql)
    if DEBUG:
        print("Saving to database done.")

def check_previous_addresses(self):
    """
        Checks if previous-addresses (backward-crawling) are cash-In-Addresses
        (Coinmixer-Addresses which are used by customers to cash-in) or
        coinmixer-addresses which are used to cashout-customers.
        This function should only be called when transaction is confirmed as an coinmixer-
             transaction.
        This function is the mainly used to determine which address should be checked next
              (backwards-crawling)
    :return:
    """
    if DEBUG:
        print ("Addreses to check: " + str(self._inputaddress_list))
    for inputAddress in self._inputaddress_list:
        if DEBUG:
            print("checking address: " + inputAddress.get_addresshash())

        inputAddress.transactions_load()
        inputAddress.set_is_cm(True) # since its an CM-transaction, each input has to be
             controlled by coinmixer.SE
        inputAddress.set_is_cm_spent(True) # since its an input, it has already been spent
              (in this transaction)

        inputtransaction = inputAddress.inputtransaction_find() # find input-transaction
        if inputtransaction is False: # mulitple outputs for this transaction. Really
             uncommon.
            if DEBUG:
                print("could not find input-transaction. Error gets logged.")
            ErrorLog.log("Error occured on input-address: " + inputAddress.get_addresshash
                 ()
                         + "(multiple inputs)", inputAddress.get_addresshash())
```

```
        return False

    if inputtransaction is None: # mutltiple signle input-transactions -> CM-Cashin-
         Address

        inputAddress.set_is_cm_cashin_address(True)
        continue
    if DEBUG:
        print("inputtransaction found: " + inputtransaction.get_hash())
        print("checking version, sequence, locktime of inputtransaction:")
    inputAddress.set_previous_transaction(inputtransaction)
    version_sequence_locktime_bool = Analyzer.
         cm_check_transaction_version_sequence_locktime(inputtransaction)

    if DEBUG:
        print("result: " + str(version_sequence_locktime_bool))

    inputAddress.set_is_cm_version_sequence_locktime(version_sequence_locktime_bool)
    if DEBUG:
        print("checking number of transactionsoutputs equals 2")

    transaction_counter = Analyzer.cm_check_transaction_transaction_outputs(
         inputtransaction, 2)
    if DEBUG:
        print("result: " + str(transaction_counter))

    if transaction_counter is False: # to many sent-transactions
        inputAddress.set_is_cm_cashin_address(True)
        continue # no further testing needed

    if version_sequence_locktime_bool is False: # version, sequence or locktime wrong
        inputAddress.set_is_cm_cashin_address(True)
        continue # no further testing needed

    if DEBUG:
        print("checking common-value:")
    common_value_bool = Analyzer.cm_check_transaction_common_value_backward(
         inputtransaction, inputAddress)
    if DEBUG:
        print("result:" + str(common_value_bool))

    if common_value_bool is True:
        #  CMcounter +=1
        inputAddress.set_is_cm_common_value(True)
    else:
        inputAddress.set_is_cm_common_value(False)

    if DEBUG:
        print ("checking fee:")
    fee_check = Analyzer.cm_check_transaction_fee_correct_partition(inputtransaction)

    if DEBUG:
        print("Fee-check: " + str(fee_check))

    if fee_check == 1 or fee_check == 2: # fee ok (gap or in partition)
        inputAddress.set_is_cm_fee(True)

    if DEBUG:
        print("full result: its a address used for customer-cashins")
    if inputAddress.get_is_cm_common_value() is True or inputAddress.get_is_cm_fee()
         is True:
        if DEBUG:
```

```
                print("full result: its probably NOT an address used for customer-cashins")
            inputAddress.set_is_cm_cashin_address(False) # if common value or fee-check
                 True -> its probably not
            # an cashin-address
        else:
            if DEBUG:
                print("full result: its probably an address used for customer-cashins")
            inputAddress.set_is_cm_cashin_address(True)

def check_next_addresses(self):
    """
        This function checks whether the next-address is an address which is probably
             owned by the coinmixer.se
        or its an address which is owned by customers.
        Differentiation whether address is coinmixer-address or customer-address is based
             on a counter.


        Counter-Rules:
            There should only be two addresses in output-list.
            The address with the highest counting-result is most probably an address
                 controlled by Coinmixer.SE.
            The address with the lower counting-result is most probably an address
                 controlled by the customer.

        More then 1 transaction sent from address -> counter = -1 (no further counting
             applied)
        Version, locktime, sequence not correct -> counter = -1 (no further counting
             applied)
        Unspent outputs availible -> counter += 0
        All outputs spent -> counter += 1
        Received an uncommon-value -> counter += 2
        Next transaction uses correct fee -> counter += 3
        Next transaction uses correct fee and is in partition (trustlevel = 2) -> += 1

        Problem: spent + common-value+correctfee == spent + correctfee (in partition)
        solving: check backwards + check next transactions..

    :return:
    """
    if DEBUG:
        print ("Addreses to check: " + str(self._outputaddress_list))
    for output in self._outputaddress_list:

        if DEBUG:
            print("Address to check: " + output.get_addresshash())
        output.transactions_load()

        if DEBUG:
            print("Analyzing address:\n Analyzing transaction-count")
        # transactionCount = 0 -> not a Coinmixer-Address (multiple spents) (strong
             indicator)
        # transactioncount = 1 -> probably not a Coinmixer-Address (unspent) (low
             indicator)
        # transactioncount = 2 -> could be a Coinmixer-address (spent) (low indicator)
        transaction_count_result = Analyzer.cm_check_address_transaction_count(output,
             self)
        if DEBUG:
            print("transaction-count result: " + str(transaction_count_result))
            print ("checking common value:")
        # True -> common-value sent (low indicator)
        # False -> uncommon-value sent (low indicator)
```

```
common_value_bool = Analyzer.cm_check_address_common_value(output) # True ->
     common value (LI)
if DEBUG:
    print("common-value result: " + str(common_value_bool))
    print("checking version, sequence, locktime")
# True -> version, sequence, locktime ok (low indicator)
# False -> version, sequence, locktime not ok (strong indicator)
version_sequence_locktime_bool = Analyzer.
     cm_check_address_version_sequence_locktime(output)
if DEBUG:
    print("version, sequence, locktime result: " + str(
         version_sequence_locktime_bool))

fee_check = None
output.set_is_cm(None)
# set next-transaction of address and apply fee-check if possible
if transaction_count_result == 2:
    sent_count_result = output.sent_counts(True)
    if sent_count_result[0] == 1:
        nexttransaction = sent_count_result[1][0]
        fee_check = Analyzer.cm_check_transaction_fee_correct_partition(
             nexttransaction) # fee-check:
        #  next tx
        output.set_next_transaction(nexttransaction)

    else: # Not coinmixer-address
        output.set_next_transaction(None)
else: # unspent -> probably not coinmixer-address (there is no next-transaction
     yet)

    output.set_next_transaction(None)

output.set_is_cm(None) # default-value

if DEBUG:
    print("results transaction-check:")

if transaction_count_result == 0: # to many transactions sent/received by address
     (hard indicator)
    output.set_is_cm(False)
    output.set_is_cm_transaction_history(0)
    output.is_cm_counter_add(-1)

    if DEBUG:
        print("Analysis of transaction-count faild ")
        print ("Counter: " + str(output.get_is_cm_counter()))

    continue
elif transaction_count_result == 2: # no unspent output available
    output.set_is_cm_spent(True)
    output.is_cm_counter_add(1)
    output.set_is_cm_transaction_history(2)

    if DEBUG:
        print("spent: True ")
        print ("Counter: " + str(output.get_is_cm_counter()))

elif transaction_count_result == 1: # unspent output available
    output.set_is_cm_spent(False)
    output.set_is_cm_transaction_history(1)

    if DEBUG:
```

```
            print("spent: False ")
            print ("Counter: " + str(output.get_is_cm_counter()))

        continue
    if version_sequence_locktime_bool is False: # version, sequence, locktime wrong (
         as coinmixer.SE-address)
        output.set_is_cm(False)
        output.set_is_cm_version_sequence_locktime(False)
        output.set_is_cm_counter(-1)

        if DEBUG:
            print("Analysis of version, sequence, locktime faild. aborting. ")
            print ("Counter: " + str(output.get_is_cm_counter()))

        continue
    elif version_sequence_locktime_bool is True: # vers., sequence, locktime correct (
         as coinmixer.SE-address)

        if DEBUG:
            print("Analysis of version, sequence, locktime ok. ")
            print ("Counter: " + str(output.get_is_cm_counter()))

        output.set_is_cm_version_sequence_locktime(True)

    if common_value_bool is False: # address an uncommon value (probably coinmixer.SE-
         address)
        output.is_cm_counter_add(2)
        output.set_is_cm_common_value(False)

        if DEBUG:
            print("common-value: False ")
            print ("Counter: " + str(output.get_is_cm_counter()))

    elif common_value_bool is True: # address received a commen value (probably a
         customer-address)
        output.set_is_cm_common_value(True)

        if DEBUG:
            print("common-value: True")
            print ("Counter: " + str(output.get_is_cm_counter()))

    if fee_check == 1: # fee ok (next transaction has same fee as previous transaction
         )
        output.is_cm_counter_add(3)
        output.set_is_cm_fee(1)

        if DEBUG:
            print("fee-check: OK (prob. last transaction) ")
            print ("Counter: " + str(output.get_is_cm_counter()))

    elif fee_check == 2: # fee ok and transaction relies in existing partition or gap
        output.is_cm_counter_add(4)
        output.set_is_cm_fee(2)

        if DEBUG:
            print("fee-check: good (in partition or gap)")
            print ("Counter: " + str(output.get_is_cm_counter()))

    elif fee_check == 0: # fee wrong
        output.set_is_cm_fee(0)

        if DEBUG:
```

```
                print("fee-check-bad ")
                print ("Counter: " + str(output.get_is_cm_counter()))

        if DEBUG:
            print("checking done. ")
            print ("Counter: " + str(output.get_is_cm_counter()))

def insert_into_cm_network(self, forward=True, depth=5):
    """
        Trough this function the results of the crawling-processes (forward/backward-
             crawling) are saved
        into database. It handles the checking-procedure of addresses/transactions and is
             responsible for
        recursive calls.

    :param forward:
    :param depth:
    :return:
    """
    if DEBUG:
        print ("crawling. forward: " + str(forward) + " depth: " + str(depth))

    if depth == 0 and forward is False: # recursion will only executed till depth == 0
        if DEBUG:
            print("maximum depth reached. stop crawling this path")
        return True

    self._forward = forward # defines if forward or backwards crawling

    if DEBUG:
        print("loading and saving address-data(value/hash-mapping/sequences). Transaction:
              " + self.get_hash())
    # create mapping for addresses and values
    inputaddress_mapping = Mapping()
    outputaddress_mapping = Mapping()

    for address in (self._inputaddress_list + self._outputaddress_list):
        # check if address is already in database
        sql = "SELECT id FROM transaction_addresses WHERE transaction_address = '"+address
             .get_addresshash() + "'"
        result = Database.sql_execute(sql)
        if not result: # address couldnt be found in database
            sql = "INSERT INTO transaction_addresses (transaction_address)" \
                  " VALUES ('"+address.get_addresshash() + "')"
            Database.sql_execute(sql)
            sql = "SELECT LAST_INSERT_ID()"
            address.set_address_id(Database.sql_execute(sql)[0][0])
        else:
            address.set_address_id(result[0][0])

        # check if value is already in database
        sql = "SELECT id FROM transaction_values WHERE transaction_value = "+str(address.
             get_value())
        result = Database.sql_execute(sql)
        if not result: # value coulndt be found in database
            sql = "INSERT INTO transaction_values (transaction_value,spent) " \
                  "VALUES ("+str(address.get_value()) + "," + str(address.get_spent()) + ")
                      "
            Database.sql_execute(sql)
            sql = "SELECT LAST_INSERT_ID()"
            address.set_value_id(Database.sql_execute(sql)[0][0])
        else:
```

```
            address.set_value_id(result[0][0])
        if address.is_inputaddress():
            inputaddress_mapping.entry_add(address.get_address_id(), address.get_value_id()
                )
        else:
            outputaddress_mapping.entry_add(address.get_address_id(), address.get_value_id
                ())

    inputaddress_mapping.mapping_insert()
    outputaddress_mapping.mapping_insert()

    tmp_sequence = self._sequence
    if type(self._sequence) == list:
        tmp_sequence = "NULL"

    sql = "INSERT IGNORE INTO transaction_data(" \
          "transaction_hash, blockheight, fee, size, " \
          "time, version, sequence, locktime, " \
          "inputaddress_value_mapping_id, " \
          "outputaddress_value_mapping_id, is_cm" \
          ") VALUES (" \
          "'"+self._hash + "', " + str(self._blockheight) + ", "+str(self._fee) + ", " +\
          str(self._size) + ", " + str(self._time) + ", " + str(self._version) + ", " +\
          str(tmp_sequence) + ", " + str(self._locktime) + ", " +\
          str(inputaddress_mapping.get_id()) + ", " + str(outputaddress_mapping.get_id()) +
              "," + str(self._is_cm) \
          + ")"

    Database.sql_execute(sql)
    sql = "SELECT LAST_INSERT_ID()"
    self._id = Database.sql_execute(sql)[0][0]

    if type(self._sequence) == list: # only single-sequences are saved in transaction_data
        -table
        sql = "INSERT INTO multiple_sequences (transaction_id, sequences) " \
              "VALUES ("+str(self._id)+",'"+json.dumps(self._sequence)+"')"
        Database.sql_execute(sql)

    # updates list which holds every transaction-data that has been seen by the crawler
    sql = "UPDATE list_of_all_transaction_hashes SET in_transaction_data = 1 " \
          "WHERE transaction_hash = '"+self._hash+"'"
    Database.sql_execute(sql)

    if DEBUG:
        print("data saved in database.")

    # Backwards-Crawling
    if self._forward is False:
        if DEBUG:
            print("backward-crawling starts.")
        checkresult = self.check_previous_addresses() # checks previous-addresses:
        # Every previous-address should be controlled by coinmixer.SE
        # however some of these addresses are cashin-addresses
        # which are used by customers to cash-in bitcoins
        # which are going to be "anonymized" and
        # others are addresses which are used by coinmixer to cashout "anonymized" coins
            to customers
        if checkresult is False: # at least one input_address-address seems not to be
            controlled by coinmixer.SE

            ErrorLog.log(
                "Error occured on transaction: " +
```

```
        self.get_hash() +
        " (transaction classified as Coinmixer.SE-transaction but at least one"
        " input_address-address seems not to be owned by Coinmixer.SE)", self.
            get_hash()
    )

    return False

list_of_cm_addresses = [] # previous coinmixer-addresses
list_of_previous_transactions = [] # previous coinmixer-transactions
for input_address in self._inputaddress_list:
    if input_address.get_is_cm_cashin_address() is False:
        list_of_cm_addresses.append(input_address)

        Analyzer.cm_check_transaction_fee_correct_partition_and_update(
            input_address.get_previous_transaction(), True
        )

        prev_transaction = input_address.get_previous_transaction()
        list_of_previous_transactions.append(prev_transaction)
        if input_address.get_is_cm_fee() is None:
            PartitionContainer.partition_insert(
                prev_transaction.get_fee_per_byte(), prev_transaction.get_time()
            )

Analyzer.cm_results_insert(self._inputaddress_list, self) # insert result to
     database
if DEBUG:
    print("list of previous transactions: " + str(list_of_previous_transactions))
# Every address on which the customer receives the "anonymized" coins should get
     logged too
if not list_of_previous_transactions:
    Analyzer.cm_log_update_first(self.get_hash())

for previous_transaction in list_of_previous_transactions:
    if DEBUG:
        print("tx to check: " + previous_transaction.get_hash())
    prev_output_list = previous_transaction.get_outputaddress_list()
    for prev_out in prev_output_list:

        found = False
        for inputaddress in self._inputaddress_list:

            if inputaddress.get_addresshash() == prev_out.get_addresshash():
                found = True

        if found is False:

            prev_out.set_is_cm(False)
            previous_transaction.set_forward(False)
            Analyzer.cm_results_insert(prev_out, previous_transaction)

    Analyzer.cm_log_insert(self.get_hash(), None, previous_transaction.get_hash(),
         depth)
    # insert analysis-result
    # todo(?): insert new partition if transaction is first and not in an partition
         ?

# recursion
for previous_transaction in list_of_previous_transactions:
    if DEBUG:
        print("recursion. depth: " + str((depth-1)))
```

```
            previous_transaction.set_is_cm(True)
            previous_transaction.insert_into_cm_network(False, depth - 1)

    if DEBUG:
        print ("maximum-depth reached or path has been fully analyzed. Try another path")
    # forward-crawling
    if self._forward is True:
        self.check_next_addresses() # checks whether next address is coinmixer-address or
             customer-address
        next_address = None
        non_cm_addresses = []
        for output in self._outputaddress_list:
            if next_address is None:
                next_address = output
                continue
            if output.get_is_cm_counter() > next_address.get_is_cm_counter():
                non_cm_addresses.append(next_address) # address with highest counter is
                     probably next cm-address
                next_address = output
        if DEBUG:
            print ("Next Coinmixer-address: " + str(next_address))
            print ("Next non-Coinmixer-address: " + str(non_cm_addresses))

        for output in self._outputaddress_list:
            if output.get_addresshash() != next_address.get_addresshash():
                output.set_is_cm(False) # unspent output
        if DEBUG:
            print("Finding next transaction:")
        next_transaction = next_address.get_next_transaction()

        if DEBUG:
            print("Next transaction: " + next_transaction.get_hash())
        next_transaction.set_is_cm(True)
        next_address.set_is_cm(True)
        Analyzer.cm_check_transaction_fee_correct_partition_and_update(next_transaction,
             True)

        if next_address.get_is_cm_fee() == 0: # insert new Fee partition (transaction
             newer then all partitions)
            if DEBUG:
                print("Transaction newer then all partitions. Inserting new Partition")
            PartitionContainer.partition_insert(next_transaction.get_fee_per_byte(),
                 next_transaction.get_time())
        if DEBUG:
            print("Saving results")
        Analyzer.cm_results_insert(self._outputaddress_list, self)
        Analyzer.cm_log_insert(self.get_hash(), next_transaction.get_hash())
        if DEBUG:
            print("Next iteration")
        next_transaction.insert_into_cm_network()

def set_hash(self, tx_hash):
    self._hash = tx_hash

def set_id(self, tx_id):
    self._id = tx_id

def set_blockheight(self, blockheight):
    self._blockheight = blockheight

def set_fee(self, fee):
```

```
        self._fee = fee

    def set_time(self, time):
        self._time = time

    def set_version(self, version):
        self._version = version

    def set_sequence(self, sequence):
        self._sequence = sequence

    def set_locktime(self, locktime):
        self._locktime = locktime

    def set_inputaddress_list(self, inputaddress_list):
        self._inputaddress_list = inputaddress_list # [address-object, address-object, ...]

    def set_outputaddress_list(self, outputaddress_list):
        self._outputaddress_list = outputaddress_list # [address-object, address-object, ...]

    def set_is_cm(self, is_cm):
        self._is_cm = is_cm

    def get_hash(self):
        return self._hash

    def get_id(self):
        return self._id

    def get_blockheight(self):
        return self._blockheight

    def get_fee(self):
        return self._fee

    def get_fee_per_byte(self):
        return int(self._fee/self._size)

    def get_time(self):
        return self._time

    def get_version(self):
        return self._version

    def get_sequence(self):
        return self._sequence

    def get_locktime(self):
        return self._locktime

    def get_inputaddress_list(self):
        return self._inputaddress_list

    def get_outputaddress_list(self):
        return self._outputaddress_list

    def get_is_cm(self):
        return self._is_cm

    def get_size(self):
        return self._size
```

```
    def get_forward(self):
        return self._forward

    def set_forward(self, forward):
        self._forward = forward


# ====================
# Network
# ====================
class Network:
    """
        Initiate the crawling-processes. Checks if first transaction provided is a Coinmixer-
             transactions, restores
        last crawling-processes, prints out network-graphs (matlab-commands)
    """

    def __init__(self):
        raise Exception("Should not be initialized")

    @staticmethod
    def load():
        """
        Not implemented yet.
        :return:
        """
        return False

    @staticmethod
    def transaction_crawling(inputtransaction_hash, forward=True, depth=5):
        """
            Starts crawling-process. Default: forward-crawling.
            Depth-parameter will only be used for backwards-crawling.
            Transaction provided by user has to be a
            Coinmixer.SE-transaction (version, sequence, locktime, fee will be checked).
            Warning: Fee-partition will be forcefully created!
            Further crawling-processes may generate wrong results if non-conmixer.SE-
                 transaction
            is used as input-transaction.
        :param inputtransaction_hash:
        :param forward:
        :param depth:
        :return:
        """
        print ("Start crwaling: hash: " + str(inputtransaction_hash) + " forward: " +
               str(forward) + " depth: " + str(depth))
        if DEBUG:
            print ("loading old results:")
        res = Analyzer.cm_log_check(inputtransaction_hash, forward) # checks if hash has
             already been processed

        # and loads last unprocessed hashes, faulty transactions are ignored
        if DEBUG:
            print ("old results have been loaded. New transactions to begin crawling with: " +
                  str(res))

        if not res:
            print ("No transaction-hash found that could be used for crawling! "
                   "(last hash produced error or is a cashin-transaction)")
        else:
            for inputtransaction_hash in res: # transactions provided by user have to be
                 Coinmixer.SE-transactions
```

```
            if DEBUG:
                print("Analyzing transaction: " + inputtransaction_hash)

            inputtransaction = Transaction(inputtransaction_hash)
            if DEBUG:
                print("Analyzing version, sequence, locktime of transaction: ")

            if Analyzer.cm_check_transaction_version_sequence_locktime(inputtransaction) is
                  False:
                if DEBUG:
                    print("Analyzing faild. Logging error.")

                ErrorLog.log(
                    "Transaction provided by user most probably not a coinmixer.SE-
                         transaction. Hash: "
                    + inputtransaction_hash + "(Sequence, Locktime, Version wrong)",
                         inputtransaction_hash)
                continue

            if DEBUG:
                print ("version, sequence, locktime ok.\nchecking if fee is ok:")

            result = Analyzer.cm_check_transaction_fee_correct_partition_and_update(
                 inputtransaction, True) # force

            if DEBUG:
                print ("version, sequence, locktime result: " + str(result))

            if result is True:
                inputtransaction.set_is_cm(True)
                inputtransaction.insert_into_cm_network(forward, depth) # start crawling
            else:
                ErrorLog.log("Wrong fee in transaction provided by user : "
                             + inputtransaction_hash +
                             " (fee inconsistent with fee-partitions)", inputtransaction_hash
                                  )

    @staticmethod
    def blockwise_crawling():
        """
        Not implemented yet.
        :return:
        """
        return False

    @staticmethod
    def graph_show():
        """
        Not implemented yet.
        :return:
        """
        return False

# ############ Forward crawling #############
# Network.transaction_crawling("104
    b7250d97294249fafd08fc2f7d0778ae213a4f0390cd8c23a7bc8de12f4a8")

# ############ Backward crawling ( depth= 4) #############
# Network.transaction_crawling("535
    b3c91bfc198e942254b950a2e3a89528eeb75e7c9f1b7aab0701bd71d6d82", False, 20)

# ############ Deanonymization (presumption: 1 forward, up to 12h delay) ############
```

```
inputtx = "74fb84d805fe35f00141fdca4f07a5a36e64b67fc1cab895033bb338c78d1d26"
inputaddress = "14avhevfq2wcfBHmHHJYtAuQW3Uowq1iXM"
participations = Deanonymizer.input_deanonymize(inputtx, inputaddress, 1, 12)

print ("Inputtransaction: " + inputtx)
print ("Results: ")
for participation in participations:
   print ("Results for " + str(participation.get_forwards_number()) + " forwards:")
   for addr in participation.get_outputaddress():
       print (addr.get_addresshash())
```