Blockchain Exercices

1 Question 1

1.1 Basics

The following questions address basic knowledge about decentralized blockchains (e.g. Bitcoin/Ethereum).

1. (1p, -1 per wrong answer) What type of Bitcoin/Ethereum clients/nodes exist? Please describe for each node how they operate and their main properties (space, CPU, network properties). If applicable, which one is the most secure?
   - ◯ Simple Payment Verification (SPV)
   - ◯ Supernodes
   - ◯ Full client storing the entire blockchain
   - ◯ Miner
   - ◯ Web Wallets

1.2 Cryptography

Open and decentralized blockchains are also referred to as cryptocurrencies. This name stems from their significant use of cryptography, in particular of elliptic curve cryptography, such as ECDSA.

1. What is encrypted in Bitcoin/Ethereum? (1p, -1 per wrong answer)
   - ◯ Nothing
   - ◯ Transactions
   - ◯ Blocks
   - ◯ The network communication
   - ◯ Merkle Tree
2. What is cryptographically signed in Bitcoin/Ethereum? (1p, -1 per wrong answer)
   - ◯ Nothing
   - ◯ Transactions
   - ◯ Blocks
   - ◯ The network communication
   - ◯ Merkle Tree
3. (3p) A Merkle Tree is a hash tree, culminating into a Merkle root hash. Please describe how a Merkle Tree in the Bitcoin block header is used to prove that a transaction is included in a block.
4. (2p) How much data is required to demonstrate that a leaf node is part of a given hash tree?

5. (1p) Given the Merkle Tree in Figure 2, assume Alice (full node) wants to prove to Bob (SPV client) that transaction 7 has been included in the respective block. What information must Alice provide to Bob?

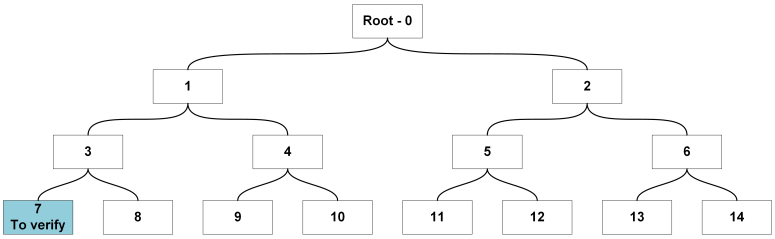

Figure 2: Merkle Tree as seen by Alice.

6. (1p) For what other purposes could a Merkle Tree be used? If you don't know about an example, please come up with one and explain your design.

1.3 Hash Functions

1. (3p) Given a hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{256}$, Alice's computer requires on average 16 minutes to find a hash with at least 24 leading zeros. What is her hashrate?
2. (1p) How long does she need on average to compute a hash with 25 leading zeros?
3. (2p) How long does she need on average to compute a hash with 30 zeros at the end?
4. (1p) Consider the hash function $\mathcal{H}' : \{0,...,9\}^* \to \{0,...,9\}$, which is calculated by taking the digital root[1] of the input. Find a second preimage to the input $x = (1,4,7,3,9,0,1,4)$.

1.4 Privacy

Related literature has shown that Bitcoin does not provide proper anonymity with it's transaction structure [2]. As such, Bitcoin addresses are linkable for example by linking the change output address with corresponding input addresses of a transaction. Many different privacy improvements for open an decentralized blockchains have been proposed (CoinJoin, Monero, Zerocash, etc.).

1.4.1 CoinJoin

CoinJoin allows to merge different transactions into a single transaction (cf. Figure 4). The underlying idea is that if you want to perform a payment, you find one or several other persons and you perform a payment together. To alleviate problems of peer discovery, a typically centralized service offers the possibility to merge transactions in a CoinJoin process.

1. (1p) Why does CoinJoin increase user privacy?
2. (1p) What problems do you see with CoinJoin?
3. (1p) How could CoinJoin be improved?

[1] i.e. repeatedly taking the digit sum until the result is one digit

1.5 Mining and Proof-of-Work

1. (2p) What does an *uncle block* refer to in Ethereum? Explain why, unlike Bitcoin, Ethereum rewards uncle blocks, as well as which actors receive uncle rewards.

2. (1p) How does the network difficulty adjustment mechanism differ between Bitcoin and Ethereum? What would be a possible drawback Bitcoin could face if it were to employ the same difficulty adjustment mechanism as Ethereum?

3. (1p) Mining pools allow miners to pool their computational resources together in order to decrease their payout variance. How do mining pool operators keep track of the amount of performed work per miner? What prevents a miner from receiving rewards by mining in two different mining pools simultaneously, submitting the same Proof-of-Work solutions to both pool operators?

4. (1p) Proof-of-Work blockchains require miners to generate a hash, which lies below some specified target value. In Bitcoin, a possible PoW solution is generated by modifying the *nonce* field in the block header. Given that the size of the nonce field is only 4 bytes big, why is the total number of hashes a miner can compute not bounded by 2^{32} hashes?

2 Wallets, Addresses and Transactions

A wallet is a software which contains one or multiple addresses of the user. An address is a unique identifier that can be both, the origin and the destination of a transfer of Bitcoin/Ethereum through a transaction.

In Bitcoin, an address corresponds to the Base58 encoded cryptographic hash of a public key [1]. An address's associated Bitcoin balance can be zero or positive, ranging from the smallest unit of 1 satoshi (10^{-8} Bitcoin) to 21 million Bitcoin, the maximum amount of Bitcoins that can be created according to the current consensus. Typically, Bitcoin addresses start with the decimal 1 (for P2PKH transactions), or 3 (for P2SH, e.g., multi-signature addresses) and typically range from 27 to 34 characters. An example Bitcoin address could be the following:

```
1BGdqciog9S67vauMWeGBfpaZhKi4afq3c

```

In Ethereum, an address similarly corresponds to the hash of a public key[2] and is a 160-bit identifier. Contrary to Bitcoin, however, Ethereum supports two types of addresses, also referred to as accounts: *(i)* user-owned accounts and *(ii)* contract-owned accounts. User-owned addresses correspond to typical Bitcoin addresses. Contract-owned accounts however are owned by so-called smart contracts.

```
0xdaed92add2e2eb04882d4169c6455792798db6f6

```

1. (1p) Why should an address contain a checksum?

[2] `https://ethereum.stackexchange.com/questions/3542/how-are-ethereum-addresses-generated`

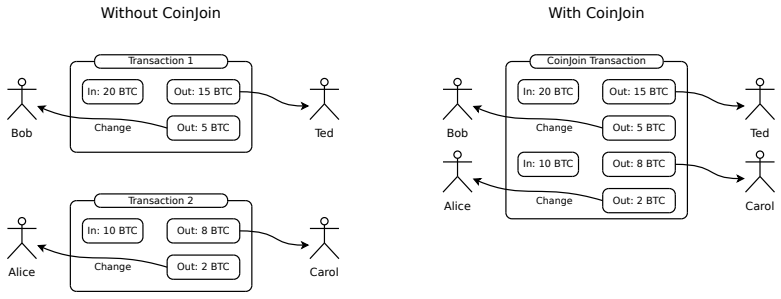

Figure 4: CoinJoin Example where two transactions are merged into one.

2. (1p) What's the advantage of a Base58 encoding?

3. (1p) Could you come up with a better address format? Explain your design and motivate your choice!

4. (1p) How does a web wallet operate? Does it store the users private keys? If not, how would that work?

2.1 Transactions

Transactions allow participants to exchange funds by specifying three key elements: (i) the funding source(s) and proof that the funds can be spent, (ii) the recipient(s) and (iii) the conditions for redemption.

In the following we discuss two possible designs for transaction

2.1.1 Bitcoin Transactions

In Bitcoin, a transaction is a data structure with inputs and outputs (cf. Figure 5). An input redeems the Bitcoins that are referenced in a previous transaction output. Transactions therefore effectively form a chain of transactions, and Bitcoins are technically only kept in transaction outputs, not within addresses.

A transaction output specifies, how many Bitcoins it contains as well as under which conditions a subsequent transaction can redeem the output. The subsequent transaction redeems the output of a previous transaction, by encoding the necessary spending information in a transaction input. These conditions, under which an output can be spent, are encoded with the help of Script and only the participants that are capable of providing input to Script, such that it evaluates to true upon execution, are allowed to spend Bitcoins of a particular Bitcoin transaction.

In Figure 5, a simplified transaction with one input and two outputs is visualized. This transaction spends Bitcoin from address X to both, address Y and Z.

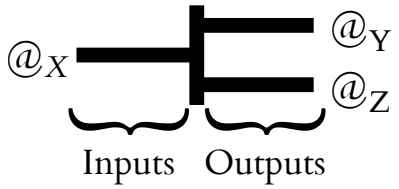

Figure 5: A transaction has in- and outputs, here one input from address X to adress Y and Z.

More precisely, the input from X, corresponds to a *former* output that has spent Bitcoins to X (cf. Figure 6). Transactions therefore create a chain of transactions. The outputs that have not yet been spent (in this example the two outputs of transaction 2), are commonly referred to as *unspent transaction outputs (UTXO).*

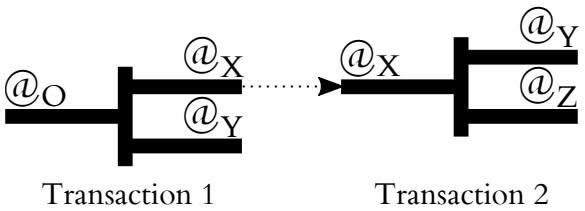

Figure 6: The input of transaction 2 refers to the output of transaction 1.

Transaction change and fees A transaction output requires being spent in its entirety. In practice, however, the output amount is unlikely to satisfy the exact transaction amount. A transaction output can, therefore, be split into n parts. An output that solely serves the purpose to refund the change to its originator is commonly referred to as *change output.*

As a sanity check, the sum of Bitcoins of the transaction outputs cannot exceed the sum of Bitcoins of the transactions inputs. The sum of Bitcoins of the transaction outputs, however, can be smaller than the sum of Bitcoins of the transaction inputs. This difference is paid as a fee to the Bitcoin miner that includes the transaction within a block.

In the following, we will discuss Bitcoin Script and different transaction types.

2.1.2 Bitcoin Script

Bitcoin introduced a custom stack-based scripting language Script in an attempt to allow different types of transactions. Script's flexibility allows extending the functionality of transactions beyond the simple transfer of funds. Script is stack-based, supports many functions (commonly referred to as opcodes) and either evaluates to true or false. The language supports dozens of different opcodes, ranging from simple comparison opcodes to cryptographic hash functions and signature verification. Because Script is supposed to be executed on any validating Bitcoin node, execution time is critical regarding Denial of Service attacks and therefore, many opcodes have been temporarily disabled. Consequently, Script is kept simple on purpose and therefore does not support the same complexity as general purpose programming languages.

An example Script program contains two constants (denoted by <...>) and one opcode (execution goes from the left to the right): `<signature> <publicKey> OP_CHECKSIG`. Constants are pushed by default on the stack, and upon execution, the stack would, therefore, contain `<signature> <publicKey>`. Then, `OP_CHECKSIG` is executed which verifies the `<signature>` under the provided `<publicKey>`. If the signature matches the provided public key, `OP_CHECKSIG` returns true, and in return, the script returns true.

Standard transaction types Bitcoin by default supports several *standard* transaction types. Only standard transaction types are broadcasted and validated within the network. Transactions that do not match the standard transaction type are generally discarded. Please note: since transactions can have multiple outputs, the different output types can be combined within one transaction.

- **Pay To Public Key Hash (P2PKH):** A P2PKH transaction output contains the following opcodes:

  `OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG`

  The corresponding input that would be eligible to spend the output, specifies the required signature and the full public key:

  `<Sig> <PubKey>`

- **Pay To Script Hash (P2SH):** A P2SH transaction output can only be redeemed by an input that provides a script, that hashes to the hash of the corresponding output. A P2SH output for example contains the following transaction output:

  `OP_HASH160 <Hash160(redeemScript)> OP_EQUALVERIFY`

  The redeeming input consequently needs to provide a *redeemScript*, that hashes to the input's hash. Every standard Script can be used for this purpose:

  `<sig> <redeemScript>`

  P2SH allows to create a transaction where the responsibility for providing the redeem conditions of a transaction is pushed from the sender to the redeemer of the funds. Consequently the sender is not required to pay an excess in transaction fees, if the redeem script happens to be complex and thus big in terms of bytes. In practice P2SH outputs are heavily used for multi-signature (multisig) transactions, but multi-signatures can both be accomplished with M-of-N output scripts as well as P2SH.

- **Multisig:** A multi-signature (or commonly referred to as multisig) transaction, requires multiple signatures in order to be redeemable. Multisig transaction outputs are usually denoted as m-of-n, m being the minimum number of signatures that are required for the transaction output to be redeemable, out of the n possible signatures that correspond to the public keys defined in the transaction output. An example transaction output corresponds to:

<m> <A pubkey> [B pubkey] [C pubkey..] <n> OP_CHECKMULTISIG

while the redeeming input follows this structure:

OP_0 <A signature> [B signature] [C signature..]

Script execution Given the background knowledge of how transactions are constructed, we now turn to the process of script execution. To validate a new transaction, the input (signature script) and the output of the previous transaction (pubkey script) are concatenated. Once concatenated, the script is executed according to the Script language. Constants, denoted by <..> are pushed on the stack, and opcodes execute their respective actions by taking into account the topmost stack value.

1. (4p) Given transaction 1 and it's output with the script OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG, and transaction 2 with it's input with the script <Sig> <PubKey>, please verify if the transaction 2 is allowed to spend the output of transaction 1. We assume here that the signature of the input of transaction 2 is valid. We expect you to draw the execution stack and the execution code for each instruction of the Script.
2. (1p) What happens if two transactions use the same previous transaction output as input?
3. (1p) How many bytes is a typical Bitcoin transaction?
4. (2p) Could you design a smaller Bitcoin transaction? Hint: Think about a different signature scheme.
5. (2p) Is it possible to change/update a transaction, after it has been broadcast to the network in Bitcoin? Explain your answer!
6. (2p) Why should you avoid to keep using an address you have already spent from, i.e., used the unspent transaction output as an input for another transaction?

2.1.3 Ethereum Transactions

Contrary to Bitcoin, Ethereum transactions are not based on transaction inputs and outputs. Each account in Ethereum has an associated *nonce*, indicating the number of transactions that this account has sent. Transactions from the same account with the same nonce are therefore considered to be double-spending transactions.

1. (1p) What is an UTXO (Unspent Transaction Output)?
2. (2p) Given the difference between a Bitcoin and an Ethereum transaction, how does this affect the scalability and simplicity of the underlying blockchain?
3. (2p) Could you think of another transaction structure that is fundamentally different from Bitcoin and Ethereum?

2.2 Blockchain Layer

1. (0.5p) How many minutes on average does it take to find a Bitcoin block?
   - ◯ 15 seconds
   - ◯ 1 minute
   - ◯ 10 minutes
   - ◯ 15 minutes
2. (0.5p) How many minutes on average does it take to find an Ethereum block?
   - ◯ 15 seconds
   - ◯ 1 minute

◯ 10 minutes

◯ 15 minutes

3. (2p) Please explain the benefits and drawbacks of having a faster or slower block interval. How would you motivate your choice of a block generation interval?

4. (1p) How is the main chain defined? Motivate your answer.

5. (2p) Could you come up with an alternative blockchain format, where it's not necessarily the longest chain which represents the current consensus?

6. (1p) Assume you want to implement the following protocol changes in Bitcoin. Read carefully and decide, whether the alterations can/must be implemented as a hard fork, soft fork or both.

   Increase of the blocksize from 1MB to 8MB.

   ◯ Hard fork

   ◯ Soft fork

   Introduction of a chainID field to differentiate between two forked version of the same chain for replay protection, e.g., Ethereum and Ethereum Classic.

   ◯ Hard fork

   ◯ Soft fork

   Redefinition of the `OP_VERIFY` opcode to mark a transaction invalid if the top stack value is true.

   ◯ Hard fork

   ◯ Soft fork

   Removal of the `OP_RETURN` opcode, previously used to attach extra data to transactions.

   ◯ Hard fork

   ◯ Soft fork

7. Explain the difference between hard and soft forks! Provide at least one example each.

3 Smart Contracts

3.1 Visibility

Ethereum offers different visibility settings such as *public* and *private*.

1. (2p) Given the contract below deployed on the public Ethereum network, who would be able to read "mySecret"? Elaborate your answer.

```
contract Example {

    address public owner;
    string private mySecret;

    constructor {
        owner = msg.sender;
    }

    function setSecret(string _secret) public {
        require(msg.sender == owner);
        mySecret = _secret;
    }

    function getSecret() public returns (string) {
        require(msg.sender == owner);
        return mySecret;
    }
}
```

3.2 Gas

Ethereum and other blockchains use gas as a metering mechanism. That means each operation has a defined gas cost associated with it. Anyone that executes a state changing operation on a smart contract or sends a transaction has to pay for its execution.

1. (0.5p) When does a user has to provide the gas required to execute the function or transaction?
   - ◯ At the end of the transaction or function execution
   - ◯ During the transaction or function execution with each operation
   - ◯ When sending the transaction or calling the function
2. (0.5p) What happens with the unused gas?
   - ◯ The left-over gas is sent to the miners
   - ◯ There is no left-over gas, it will all be used independent of the amount of gas being sent
   - ◯ The left-over gas is refunded to the sender of the transaction
3. (1p) Why is a mechanism such as gas required?

3.3 Security

Smart contracts have suffered from quite severe security flaws and bugs. There are major efforts to prevent future incidents.

1. (1p) One of the most famous security flaws is the "The DAO" incident. What is the problem with the contract *Vulnerable* below?

```
contract Vulnerable {

    mapping(address => bool) authorized;
    mapping(address => uint) balances;

    function refund(uint amount) public {
        require(authorized[msg.sender]);
        require(amount <= balances[msg.sender]);

        msg.sender.call.value(amount)("");
        balances[msg.sender] -= amount;
    }
}
```

2. (2p) The contract *Vulnerable* can be exploited. Write a short fallback function that would exploit the vulnerability.

```
contract Vulnerable {
    ...// as above
}

contract Exploit {

    Vulnerable v;

    function register(address contract) public {
        v = Vulnerable(contract);
    }

    function exploit() public {
        // your code here
    }

    // your code here
}
```

3. (2p) How could you fix the *refund* function?
4. (1p) What is a better alternative to the low-level *call* instructions and why would you rather use that?
5. (1p) What is the result of the following line of code in Solidity? Why?

```
uint8(255) + uint8(1)
```

References

[1] Bitcoin Developers. Bitcoin address generation. Available from: `https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses`.

[2] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[3] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. *IACR Cryptology ePrint Archive*, 2013(881), 2013.