Cryptocurrencies (WS 2021) Univ-Prof. Dr. Matteo Maffei

Final Exam: Part B

Date: 27/01/2022 TU Wien

Name: ***Matriculation Number:***

- You should be alone and keep your camera switched on during the exam.
- Part B of the exam takes 90 minutes. You can get at most 30 points. The number of points you can get for an exercise thus gives you a hint about how much time you should spend on that exercise.
- **Write legibly and be concise**. It is in your best interest that we understand your answers. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it.
- When finished, the solution should be uploaded back in TUWEL in the form of a PDF file as well. You can use dedicated apps (e.g., Adobe Scan, MS Office Lens), print and scan, or whichever solution you find more convenient.
- Good luck!

Problem	1	2	**Total**
Score			
Points	10	20	

Name:	/ Matriculation Number:

Problem 1: Atomic swaps in payment channels (10 points)

As seen in the lecture, the hash time-lock contract (HTLC) is a useful tool in payment channels. In particular, we denote by $HTLC(Alice, Bob, x, y, t)$ a HTLC between Alice and Bob. A HTLC contract implements two clauses:

1. If Bob presents Alice a random value r such that $H(r) = y$ before time t has elapsed, Bob gets x coins from Alice.
2. Otherwise, if time t elapses, Alice gets x coins back.

HTLC can be used to carry out an atomic swap between Alice and Bob. In an atomic swap between Bitcoin and Litecoin, Alice pays β bitcoins to Bob if and only if Bob pays γ litecoins to Alice. Moreover, Alice and Bob are rational players meaning that they won't follow the protocol steps if any other strategy gives them more benefit.

Assume that Alice and Bob have already opened an unidirectional payment channel in Bitcoin of the form Alice $\rightarrow$ Bob and that Alice has enough balance on such channel to pay β bitcoins. Similarly, assume that Alice and Bob have already opened an unidirectional payment channel in Litecoin of the form Bob $\rightarrow$ Alice and that Bob has enough balance on such channel to pay γ litecoins.

Adaptor Signature (AS) As seen in the lecture, an adaptor signature (AS) is a digital signature scheme that allows to embed a secret inside. In particular, we denote here by $AS(Alice, Bob, x, y, t)$ an adaptor signature between Alice and Bob. An AS implements two clauses:

1. If Bob presents Alice a random value r, such that $g^r = y$, before time t has elapsed, Bob gets x coins from Alice. Here, assume that g is a known group generator and that computing a value r (i.e., discrete logarithm) such that $g^r = y$ is hard (i.e., similar to how computing $H(r) = y$ is hard if one does not know r).
2. Otherwise, if time t elapses, Alice gets x coins back.

Exercises

1. **(5 points)** Describe how HTLC can be used to perform an atomic swap where Alice sends β bitcoins to Bob if and only if Bob pays γ litecoins to Alice. For that, you should describe:
   - The inputs for the HTLC contracts that you use in your protocol. Argue why you have chosen those inputs. You can use up to two HTLC contracts (i.e., 0, 1 or 2).
   - How Alice and Bob can get each of those inputs.
   - In which order the different HTLC contracts are applied to *successfully* perform an atomic swap.
   - Describe also the release phase.
2. **(5 points)** Is it possible to design an atomic swap using adaptor signatures? If yes, please give such a construction by specifying all the steps used to perform an atomic swap using adaptor signatures. If not, please motivate your answer as why this is not possible.

Name:	/ Matriculation Number:

Space for your answer

Name:	/ Matriculation Number:

Extra space

Name:	/ Matriculation Number:

Problem 2: Bitcoin scripts (20 points)

Bitcoin blockchain adopts the UTXO transaction model. In this model, transaction outputs are constrained by *locking scripts* (i.e., scripts consisting of a set of operations and data that dictate how funds can be spent). To be able to spend the funds, *unlocking scripts* (also known as *redeem scripts*) have to be provided as inputs to the spending transaction. Notably, an unlocking script is provided in a future transaction's input and is then executed before the locking scripts.

Bitcoin scripting language is stack-based, this means that every instruction is executed exactly once, in a linear manner. There are two types of instructions: *data instructions* and *opcodes*. When a data instruction appears in a script, that data is simply pushed onto the top of the stack. In contrast, opcodes perform some function, often taking as input the data on top of the stack. Only two possible outcomes can result when a Bitcoin script is executed. It either executes successfully with no errors, in which case the transaction is valid. Or, if there is any error while the script is executing, the whole transaction will be invalid and should not be accepted into the blockchain.

In this problem you have to deal with some Bitcoin scripts. More precisely, you have to understand scripts, write unlocking scripts, pinpoint flaws and fix them. Please refer to Table 1 to help you with the opcode usage and description. To highlight data within scripts, we will use brackets: `<data>`. Please use the same notation and employ general but meaningful names, for instance: `<pk>` for a public key, `<sig>` for a signature, `<pk_hash>` for the hash of a public key and so on. Also, as long as the data/names you use refer to the correct string type, assume that `OP_EQUAL` and `OP_EQUALVERIFY` return 1 and *nothing* respectively, i.e. the checks pass.

Exercises

1. **(5 points)** Have a careful look at the locking script below. It corresponds to a widely used Bitcoin logic. Which one is it? Write the unlocking script that allows to redeem the funds by entering the `OP_IF` condition (lines 2 and 3), and then the unlocking script that redeems the funds by entering the `OP_ELSE` (lines 5 and 6). Show (for the first case only) how the stack is populated/consumed step by step .

```
1 OP_IF
2     OP_SHA256 <preimageHash> OP_EQUALVERIFY
3     OP_DUP OP_HASH160 <redeem_address>
4 OP_ELSE
5     <absTL> OP_CHECKLOCKTIMEVERIFY OP_DROP
6     OP_DUP OP_HASH160 <refund_address>
7 OP_ENDIF
8 OP_EQUALVERIFY
9 OP_CHECKSIG
```

2. **(5 points)** The script that follows has the same logic as the previous one, but has a slightly different implementation. It has three flaws. Show the flaws and their fixes.

```
1 OP_HASH160 <preimage> OP_EQUALVERIFY
2 OP_IF
3     <payee_address> OP_CHECKSIG
4 OP_ELSE
5     <locktime> OP_CHECKLOCKTIMEVERIFY
6     <payee_address> OP_CHECKSIG
7 OP_ENDIF
```

3. **(5 points)** The locking script below is meant to be a Pay-To-Public-Key-Hash (P2PKH) script (i.e., "pay to Bitcoin address"), which is the most common locking script in Bitcoin. However, this script contains one subtle flaw. First, show the flaw and its fix. Then, write the unlocking script.

```
OP_DUP OP_HASH256 <pk_hash> OP_EQUALVERIFY OP_CHECKSIG
```

4. **(5 points)** In this last part of the problem you have to deal with the three transactions you find in Table 2: $\texttt{tx}_1$, $\texttt{tx}_2$ and $\texttt{tx}_3$. The input of $\texttt{tx}_2$ spends the output of $\texttt{tx}_1$, and the input of $\texttt{tx}_3$ spends the output of $\texttt{tx}_2$. Is this really the case in the transaction scripts below or will something go wrong? Elaborate on your answer.

Opcode	Input	Output	Description
OP_IF	-	-	If the top stack value is 1, the statements are executed. The top stack value is removed.
OP_ELSE	-	-	If the preceding OP_IF, OP_NOTIF or OP_ELSE was not executed then these statements are. If the preceding OP_IF, OP_NOTIF or OP_ELSE was executed then these statements are not.
OP_ENDIF	-	-	Ends an if/else block. All blocks must end, or the transaction is invalid. An OP_ENDIF without OP_IF earlier is also invalid.
OP_SHA256	input	256-bit hash	The input is hashed using SHA-256.
OP_HASH160	input	160-bit hash	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_HASH256	input	256-bit hash	The input is hashed twice with SHA-256.
OP_EQUAL	x1 x2	True/False	Returns 1 if the inputs are exactly equal, 0 otherwise.
OP_EQUALVERIFY	x1 x2	Nothing/Fail	Same as OP_EQUAL, but runs OP_VERIFY afterward.
OP_VERIFY	True/False	Nothing/Fail	Marks transaction as invalid if top stack value is not true. The top stack value is removed.
OP_DUP	x	x x	Duplicates the top stack item.
OP_CHECKLOCKTIMEVERIFY	x	x /Fail	Marks transaction as invalid if the top stack item is greater than the transaction's LockTime field, otherwise does nothing.
OP_DROP	x	Nothing	Removes the top stack item.
OP_CHECKSIG	sig pk	True/False	The entire transaction's outputs, inputs, and script are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise.
OP_CHECKMULTISIG	sig1 sig2 ... OP_M pk1 pk2 ... OP_N	True/False	OP_M indicates M signatures and OP_N indicates N public keys. Compares the first signature against each public key until it finds a valid match. Starting with the subsequent public key, it compares the second signature against each remaining public key until it finds a valid match. The process is repeated until all signatures have been checked or not enough public keys remain to produce a successful result. All signatures need to match a public key. Signatures must be placed in the *unlockingScript* in the same order as their corresponding public keys were placed in the *redeemScript*. If all signatures are valid, 1 is returned, 0 otherwise.

Table 1: Bitcoin's opcodes usage and description.

Tx	Input script	Output script
$\texttt{tx}_1$	Not relevant for the exercise	`OP_2 <pk_1> <pk_2> <pk_3> OP_3 OP_CHECKMULTISIG`
$\texttt{tx}_2$	`<sig_2> <sig_3>`	`<pk_i> OP_EQUALVERIFY OP_DROP <pk_i> OP_CHECKSIG`
$\texttt{tx}_3$	`<sig_i> <pk_k> <pk_i>`	Not relevant for the exercise

Table 2: The inputs and outputs of $\texttt{tx}_1$, $\texttt{tx}_2$ and $\texttt{tx}_3$ are shown.

Space for your answer

Name:	/ Matriculation Number:

Extra space