IT486: Blockchains and Cryptocurrencies Autumn 2020-21
**Homework 1**
**Due:** Sep 23 23:59 via Google classroom

**Instructions.** This homework consists of 4 problems. You are to work individually. Only typed solutions in pdf will be accepted. Write your name and ID on the first page. Late submissions will not be accepted.

**Problem 1** (4 marks) **Blinding with RSA:** Consider the blind signature protocol using RSA. Suppose Alice tries the following fraud. She prepares several pairs of messages, $m_1$ and $m_2$, which say, respectively, "The Bank of Cryptopia Will Pay to the Bearer \$1", and, "The Bank of Cryptopia Will Pay to the Bearer \$100", each concatenated with whatever other information the protocol requires. For each of these pairs, she finds a pair $r_1, r_2$ of blinding factors such that

$$r^{e_1} m_1 \pmod n = r^{e_2} m_2 \pmod n.$$

When she presents the bills for signature and is challenged to present blinding factors, she will reveal $r_1$. When she receives the blinded signature, she unblinds with $r_2$. In this manner, she is able to buy a \$100 bill for a dollar.
(a) Show that such an attack is possible in principle using $n = 77, m_1 = 10, m_2 = 8$, and $e = 7$.
(b) Explain carefully why the attack is not likely to succeed in practice. Provide a mathematical justification for your answer.

**Problem 2** (4 marks) **Breaking ECDSA:** We use the notation used in the lecture.

(a) The case where $R = kG$ has $x$-coordinate equal to 0 is prohibited in ECDSA signatures. Show that if this check was omitted and if an adversary could find an integer $k$ such that $kG$ has $x$-coordinate zero, then the adversary could forge ECDSA signatures for any message.
(b) The case $s = 0$ is prohibited in ECDSA signatures since the `Verify` algorithm fails when $s$ is not invertible. Show that if a signer outputs a signature $(r, s)$ produced by the `Sign` algorithm with $s = 0$ then an adversary would be able to determine the private key $e$.

**Problem 3** (6 marks) **Multi-judge escrow:** In class we saw how to use 2-out-of-3 multisig to build an escrow service where Alice can buy a product from Bob and, if all goes well, Alice gets the product and Bob gets paid. Otherwise a judge can adjudicate the dispute. One issue with that protocol is that the judge may demand a service fee and the participants, Alice and Bob, have no choice but to pay. In this problem, your goal is to design an escrow system where, ahead of time, Alice and Bob agree on the set of three judges so that during adjudication they can choose any one of the three to adjudicate. We are assuming that the three judges are honest and consistent, that is, all three will always rule the same way.

Show how to implement a three-judge escrow system using a single standard multisig transaction to a multisig address that all parties agree on ahead of time. Your design must ensure that even if the three judges collude, they cannot steal the funds that Alice sends to Bob. Recall that if all goes well then the parties need not involve the judges. If something goes wrong, then any one of the three judges can adjudicate. *Hint:* You will need to use more than 5 keys, Alice and Bob will use more than one key each.

**Problem 4** (6 marks) **Bitcoin scripts:** Exploring the Bitcoin blockchain, you discover a bunch of interesting transactions with strange locking scripts. (1) Write unlocking scripts for each of them, and (2) Demonstrate your script will succeed by proving the stack is empty at the end of execution. To do so, show the state of the stack after each operation. (If the locking script is unredeemable, explain why.) Please refer to Appendix A for a list of useful Bitcoin opcodes.

(a) `OP_2DUP OP_ADD OP_5 OP_EQUALVERIFY OP_SWAP OP_DROP OP_3 OP_EQUAL`

(b) `OP_2DUP OP_2DUP OP_ADD OP_ADD OP_ADD OP_1 OP_EQUALVERIFY OP_ADD OP_0 OP_EQUAL`

(c) `OP_1 OP_0 OP_EQUAL OP_IF OP_3DUP OP_SUB OP_ADD OP_3 OP_EQUAL OP_ELSE OP_DUP OP_SUB OP_EQUAL OP_ENDIF`

## A Useful Bitcoin Opcodes

| **Opcode** | **Input** | **Output** | **Description** |
|---|---|---|---|
| `OP_1`–`OP_16` | Nothing | 1–16 | The number in the word name (1–16) is pushed onto the stack. |
| `OP_DUP` | `x` | `x x` | Duplicates the top stack item. |
| `OP_2DUP` | `x1 x2` | `x1 x2 x1 x2` | Duplicates the top two stack items. |
| `OP_3DUP` | `x1 x2 x3` | `x1 x2 x3 x1 x2 x3` | Duplicates the top three stack items. |
| `OP_SWAP` | `x1 x2` | `x2 x1` | The top two items on the stack are swapped. |
| `OP_ADD` | `a b` | `a+b` | `a` is added to `b`. |
| `OP_SUB` | `a b` | `a-b` | `b` is subtracted from `a`. |
| `OP_DROP` | `x` | Nothing | Removes the top stack item. |
| `OP_EQUAL` | `x1 x2` | True/false | Returns 1 if the inputs are exactly equal, 0 otherwise. |
| `OP_VERIFY` | True/false | Nothing/fail | Marks transaction as invalid if top stack value is not true. |
| `OP_EQUALVERIFY` | `x1 x2` | Nothing/fail | Same as `OP_EQUAL`, but runs `OP_VERIFY` afterward. |
| `OP_IF` | `x` | Nothing | if [statements] [else [statements]]* endif If the top stack value `x` is not False, the statements are executed. The top stack value is removed. |
| `OP_ELSE` | N/A | N/A | if [statements] [else [statements]]* endif If the preceding `OP_IF` or `OP_NOTIF` or `OP_ELSE` was not executed then these statements are and if the preceding `OP_IF` or `OP_NOTIF` or `OP_ELSE` was executed then these statements are not. |
| `OP_ENDIF` | N/A | N/A | if [statements] [else [statements]]* 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. |