CS 251 Assignment 3

Luis A. Perez

Problem 1. Idioms of use.

Solution:

a. Yes, an observer can identify the payee in transactions (1) quite easily. We can see that in transaction (1), Bob paid a single individual, creating a UTXO (we know this because we've identified the other output as a change address). This unidentified UTXO is later used, so we know that Bob must have paid Alice. Only Alice can spend her own change UTXO, so we know that Alice must have signed this transaction. As such, the unidentified UTXO must have belonged to Alice, which means Bob paid Alice in transaction (1).

b. It is hard to determine the identity of who paid Carol because of the fact that the previous transaction has multiple input addresses and multiple outputs (CoinJoin transaction). Because of this, it is no longer clear which inputs made their way to which outputs, and as such, we've lost information about whoever paid Carol.

Problem 2. Vulnerable 3-party payment channel.

Solution: Given the lack of clarification in this problem, we now clarify exactly what is being proposed. Specifically, the way the 3-way payment channel works is presented below. This is based on collecting information in Piazza:

From @654 in Piazza:

Assume that we have the following payment scheme which is a generalization of the non-expiring payment channel from class:

A creates a transaction that places some amount of money (say z bitcoins) into an escrow UTXO which is locked with a 3 of 3 multisig. A doesn't post this transaction yet though. First A also creates another transaction that refunds the entire escrow to herself and has both B and C sign it. (this refund is locked with some timelock or can be taken immediately by either B or C given knowledge of some preimage of a hash x_0). A doesn't sign it herself yet. Specifically, the TX has the UTXO outputs as:

- Refund UTXO with script pk: Alice can redeem in 48 hours, or B / C can claim immediately with preimage of $H(x_0)$
- Payment UTXO with script pk: P2PKH for Bob
- Payment UTXO with script pk: P2PKH for Carol

Then A posts the escrow and A can start making payments to B and C.

To make a payment (to say B for 1 bitcoin) A selects some new preimage x_1 and creates a tx_1:

- Refund: Alice can redeen in 48 hours $z - 1$ BTC, or B/C can claim immediately with preimage of $H(x_1)$.
- Pay B 1 btc.
- pay C 0 btc.

A doesn't sign this tx1 yet. A instead sends it to both B and C. They sign it and they send their signatures back. Once A receives whatever good she bought from B, A sends B and C $x0$, the most recent preimage which pretty much voids the previous most recent tx.

At any time A can decide to settle by posting the most recent tx. If A posts any other transaction then B or C can post the preimage for that tx and take all the money that was meant to go back to A.

With the above description out of the way (and a clear understanding of the set-up), we now explain how two colluding parties can steal funds from the third.

Note that following the above protocol, at timestep t there are three transactions which have not been invalidated. Let us suppose the account balances for A,B,C are $\{a_t, b_t, c_t\}$ respectively. Then the 3 transactions looks as follows.:

- TXA_t: Pay a_t coins to A in 48 hours or B/C given pre-image of x_t. Pay B b_t coins and pay C c_t coins.
- TXB_t: Pay b_t coins to B in 48 hours or A/C given pre-image of y_t. Pay A a_t coins and pay C c_t coins.
- TXC_t: Pay c_t coins to C in 48 hours or A/B given pre-image of z_t. Pay A a_t coins and pay B b_t coins.

After some internal transactions, we are now at state $t + k$, and we have the balances $\{a_{t+k}, b_{t+k}, c_{t+k}\}$, and similary, TXA_{t+k}, TXB_{t+k}, and TXC_{t+k}. Let us suppose we're in a situation where $a_{t+k} < a_t$ and $b_{t+k} < b_t$ (both A and B, in net, have paids funds to C).

Note that the above implies that x_t and y_t have both been reveleaed at some point, supposedly "invalidating" TXA_t and TXB_t. However, note that at state $t + k$ both A and B are worse off than they were at state t. This means that A and B are both now incentivized to somehow "settle" the channel at an earlier points.

WLOG, to do this, suppose that A talks to B and proposes that he is willing to submit TXA_t if B promises to claim the funds (instead of C) and gives these claimed funds to A. If this succeeds, then A and B will both end-up with more money than the current state, so B is incentivized to play along (there is some concern that B would steal the funds and keep them to himself, but if A does not trust B, he/she can simply not submit TXA_t). However, if they do successfully trust each other, let us analyze the result.

Without informing C, A submits TXA_t to the block-chain, and immediately, B claims both $a_t + b_t$ coins, leaving c_t coins for C (note that $c_t < c_{t+k}$). B then gives a_t coins to A offline, as payment for the collusion.

In this way, A and B have now colluded to steal funds from C, since they have "closed" the payment channel at an earlier, 'invalid' state.

This attack is preventable if C is constantly monitoring the block-chain, but even then, it will succeed with 50% probability (eg, even when C and B both submits claims, the block chain will arbitrarily pick a winner).

Problem 3

Solution: The Ethereum re-entrancy attack can occur when a contract (contract A) makes a call to another address (B) before it's own execution is complete, and in particular, before its own state has been updated to reflect the changes it wants to make. The attach involves the contract at address B re-entering the same function in contract A (before the original call has completed).

This can lead to a loss of funds in the following way. Suppose A has a function which is meant to withdraw a certain amount of funds from the contract, and send them to B. If A sends the funds to B before its local state is updated (eg, before it marks the funds as withdrawn), the sending of the funds can be intercepted by B and B can request another withdraw, before the first completes. As such, contract A will again withdraw the same funds and send them to B. This is an example of re-entrancy being abused and leading to a loss of funds.

This attack can be prevented by making sure that any calls to external addresses occur only *after* their relevant state have been updated in the contract code.

Problem 4

Solution:

a. As per the problem statement, we let $C_M(\lambda, \mathbf{v})$ be an arithmetic circuit that outputs $0 \in \mathbb{F}$ if and only if $M\mathbf{v} = \lambda\mathbf{v}$ where $\mathbf{v} \neq 0$.

We know propose a linear PCP (P, V_1, V_2) for C_M as follows. We note that the prover P will output the proof:

$$\pi = \mathbf{v}$$

- **How V_1 issues the queries:**
  Following the hint, V_1 will chose a random vector $\mathbf{r} \in \mathbb{F}^n$ such that $\mathbf{r}^T\pi \neq 0$ and $\mathbf{r} \neq 0$ (this is critical for the proof to work). It will then compute $\mathbf{u} = \mathbf{r}^T M \in \mathbb{F}^n$ and issue $\mathbf{u}$ as its first linear query. It's second linear query will be $\mathbf{r}$.
  How V_2 decides the output
  V_2 first receives $a_u = \langle \mathbf{u}, \pi \rangle \in \mathbb{F}$ and $a_r = \langle \mathbf{r}, \pi \rangle \in \mathbb{F}$. V_2 then compares a_u to $\lambda \cdot a_r$, and it outputs *yes* if the values are equal. Otherwise, it outputs *no*. Note that this requires V_2 to compute a constant number of arithmetic and comparison operations, independent of n.
- We now prove that with the above set-up, a malicious prover will not be able to fool the verifier. That is to say, if the prover has a $\mathbf{v}$ such that $M\mathbf{v} = \lambda\mathbf{v} + \Delta$ where $\Delta \neq 0$, then the probability of acceptance is bounded by $\frac{1}{|\mathbb{F}|}$.

$$\begin{aligned}
\Pr[V_2(a_u, a_r) = yes] &= \Pr[a_u = \lambda \cdot a_r] && \text{(By definition of } V_2\text{)} \\
&= \Pr[(\mathbf{u}^T\pi) = \lambda \cdot (\mathbf{r}^T\pi)] && \text{(Definition of } a_u \text{ and } a_r\text{)} \\
&= \Pr[(\mathbf{r}^T M\mathbf{v}) = \lambda \cdot (\mathbf{r}^T\mathbf{v})] \\
& && \text{(Definitions of } \pi \text{ and } \mathbf{r}^T \text{ as per } P \text{ and } V_1\text{)} \\
&= \Pr[r^T(\lambda\mathbf{v} + \Delta) = \lambda \cdot (\mathbf{r}^T\mathbf{v})] && \text{(Properties of } M \text{ and } \mathbf{v}\text{)} \\
&= \Pr[\lambda \cdot (\mathbf{r}^T\mathbf{v}) + (\mathbf{r}^T\Delta) = \lambda \cdot (\mathbf{r}^T\mathbf{v})] && \text{(Distribute results)} \\
&= \Pr[r^T\Delta = 0] \\
&\leq \frac{1}{|\mathbb{F}|}
\end{aligned}$$

The last line is due to the fact that $\Delta \neq 0$ and $\mathbf{r} \neq 0$. As such, the probability of $r^T\Delta = 0$ is given by at most $\frac{1}{||\mathbb{F}}$.

b. We now describe how to convert the above linear PCP into a pre-processing SNARK (S, P, V) for $C_M(\lambda, v)$ using linear-only encodings. We begin by describing how each of the algorithms $S(M) \to (S_P, S_V), P(S_P, \mathbf{x}, \mathbf{w}) = P(S_P, (M, \lambda), \mathbf{v}) \to \pi$ and $V(S_V, \mathbf{x}, \pi) = V(S_V, \lambda, \pi) \to \{yes, no\}$.

- The setup algorithm, $S(M) \to (S_P, S_V)$.

  The first step for the setup algorithm is to run the $Gen(\Lambda)$ producing pk (the public key) and C (the encoding space) for our linear-only encoding. Both of these will become public (accessible to everyone).

  We then chose a secret random $\mathbf{r} \in \mathbb{F}^n$, and compute $\mathbf{u} = r^T M \in \mathbb{F}^n$. We also chose random $\alpha, \beta \in \mathbb{F}$ values.

  We now encode the above values. Note that $Enc(pk, \mathbf{x})$ where $\mathbf{x} \in \mathbb{F}^n$ means that we simply construct the vector where each element is the encoded version of each element in $\mathbf{x}$. As such, we have the following:

$$S_P \leftarrow \left(Enc(pk, \mathbf{u}), Enc(pk, \mathbf{r}), Enc\left(pk, \begin{bmatrix} \alpha \\ \beta \end{bmatrix}\right)\right)$$
$$S_V \leftarrow \left(Enc\left(pk, \begin{bmatrix} \alpha \\ \beta \end{bmatrix}\right)\right)$$

  These are the final outputs of our algorithm, to be provided to the prover and verifier.

- The prover, $P(S_P, (M, \lambda), \mathbf{v})$.

  The prover is given $Enc(pk, \mathbf{u})$ and $Enc(pk, \mathbf{r})$, so by the fact that we're using a linear-only encoding, it can and does compute:

$$[a_u] = Enc(pk, \langle \mathbf{u}, \mathbf{v} \rangle)$$
$$[a_r] = Enc(pk, \langle \mathbf{r}, \mathbf{v} \rangle)$$

  Note that this is possible because $\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^n u_i v_i$ which is just a linear transformation. As such, even though the prover does not know the values $\mathbf{u}, \mathbf{r}$, since it knows its encodings, it can compute the encoding of the linear combinations of the vector entries. More specifically, this can be done by repeated applications of the *Add* primitive which allows us to compute the encoding of the result of the sum of two values with just their encodings.

  Finally, the prover also computes the following (to proof that it used the samve $\mathbf{v}$ always):

$$[d] = Enc(pk, \langle \alpha \mathbf{u} + \beta \mathbf{r}, \mathbf{v} \rangle)$$

  Again, since the above is ultimately just a linear combination of encoded values, the encoding is computable by just knowing the encoding of those values, and not the values themselves.

  The prover will final output the proof π as:

$$\pi = ([a_u], [a_r], [d])$$

- The verifiers, $V(S_V, \lambda, \pi)$.

  The verifier wants to check that $a_u = \lambda a_r$ and that $d = \alpha a_u + \beta a_r$. Thankfully, given that we're using a linear-only encoding, these checks can all be performed by using *QuadTest*. We recall that $\mathit{QuadTest}(pk, (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3), \eta) \to \{0,1\}$ outputs 1 if and only if $\mathbf{c}_i[j] = \mathit{Enc}(pk, \mathbf{x}_i[j])$ and $\langle \mathbf{x}_1, \mathbf{x}_2 \rangle = \langle \eta, \mathbf{x}_3 \rangle$.

  As such, the first thing the verifier checks is that $\mathit{Verify}(pk, \pi)$ and $\mathit{Verify}(pk, S_V)$ each return 1 (meaning that all of the input encodings are indeed valid encodings under the given pk).

  Next, the verifier checks that:

  $$\mathit{QuadTest}(pk, ([1], [a_u], [a_r]), \lambda) == 1$$

  This checks that $a_u = \lambda a_r$ (note that $[1] = \mathit{Enc}(pk, 1)$ where 1 is the multiplicative identity in $\mathbb{F}$).

  Then, to make sure that the prover used the same $\mathbf{v}$ for each of the queries, we check the following holds:

  $$\mathit{QuadTest}\left(pk, \left(\begin{pmatrix}[\alpha] \\ [\beta]\end{pmatrix}, \begin{pmatrix}[a_u] \\ [a_r]\end{pmatrix}, [d]\right), 1\right) == 1$$

  which simply serves to check that $\alpha a_u + \beta a_r == d$, implying that the same v was used by the prover with high probability.

  If all of the above checks pass, the verifier outputs 1. Otherwise, we output 0.

- We now make our linear PCP desinged in (a) honest-verifier zero-knowledge (HVZK). To do this, we first expand the proof π to:

  $$\tilde{\pi} = (s, \mathbf{v}) \in \mathbb{F}^{n+1}$$

  where s is chosen at random by the prover. We also expand the queries issued by V_1 as follows:

  $$\tilde{\mathbf{u}} = (\lambda, \mathbf{u}) \in \mathbb{F}^{n+1}$$
  $$\tilde{\mathbf{r}} = (1, \mathbf{r}) \in \mathbb{F}^{n+1}$$

  As such, V_2 will now receive the query responses $\tilde{a}_u = \langle \tilde{\mathbf{u}}, \tilde{\pi} \rangle \in \mathbb{F}$ and $\tilde{a}_r = \langle \tilde{\mathbf{r}}, \tilde{\pi} \rangle \in \mathbb{F}$. It then proceeds to check whether $\tilde{a}_u = \lambda \tilde{a}_r$, and if so, outputs 1. Otherwise, outputs 0. To see why this works, simply note the following (assuming honest prover):

  $$\begin{aligned} \tilde{a}_u &= \tilde{\mathbf{u}}^T \tilde{\pi} = s\lambda + \mathbf{u}^T \mathbf{v} = s\lambda + \mathbf{r}^T M \mathbf{v} = (s + \mathbf{r}^T \mathbf{v})\lambda \\ \lambda \tilde{a}_r &= \lambda \tilde{\mathbf{r}}^T \tilde{\pi} = \lambda(s + \mathbf{r}^T \mathbf{v}) \end{aligned}$$

  As such, a similar proof as that given in (a) leads to the probabilistic argument that this is the appropriate check to make, and that the verifier can only be fooled with at most probability $\frac{1}{|\mathbb{F}|}$ (even by a malicious prover).

We now prove that the proposed modification makes our linear PCP HVZK. We do this by constructing a fast simulator $\mathit{Sim}(M, \lambda)$ that outputs a tuple $(\tilde{\mathbf{u}}, \tilde{\mathbf{r}}, \tilde{a}_u, \tilde{a}_r)$ that is distributed indistinguishibly from the real protocol. The tuple is constructed as follows:

$$\begin{aligned}
\tilde{\mathbf{r}} &= (1, \mathbf{r}) && \text{(Where } \mathbf{r} \in \mathbb{F}^n \text{ is sampled at random)} \\
\tilde{\mathbf{u}} &= (\lambda, \mathbf{u}) && \text{(Where } \mathbf{u} = \mathbf{r}^T M\text{)} \\
\tilde{a}_r &= s && \text{(Where } s \in \mathbb{F} \text{ is sampled uniformly at random)} \\
\tilde{a}_u &= \lambda \tilde{a}_r
\end{aligned}$$

From the above, it's clear that the $\tilde{\mathbf{r}}$ and $\tilde{\mathbf{u}}$ are indistinguishable from the real protocol (they are, in fact, computed in the exact same way as in the real protocol). As for $\tilde{a}_r$, note that this is indistinguishable from the real protocol since in the real protocol we have a uniformly random number added to the product of $\mathbf{r}^T\mathbf{v}$, which means the resulting value is uniformly random. Similarly, the distribution of $\tilde{a}_u$ is indistinguishable since it satisfies the property that $a_u = \lambda a_r$.

As such, as we can see above, the running time of this simulator is entirely dominated by the computation of $\mathbf{u} = \mathbf{r}^T M$. As such, this simulator is fast which demonstrates that our new protocol is HVZK.

Problem 5

Solution:

a. An attacker can easily exploit this error to make money. For example, an attacker could purchase $150 USD worth of ether, and the lock this ether into a CDP and withdraw dai. Since the oracle mistakenly believe the price of either is $1000 USD rather than $100 USD, the attacker will be allowed to withdraw up to $1000 USD worth of dai (150% max ceiling). If the market for dai has not yet collapsed, the attacker can then exchange this for $1000 USD. Once the error is corrected (and the price of ether reverts to 100), the attacker can pay down the debt in the CDP by simply sending $150 US worth of dai to the contract, thereby paying down the entire.

b. Assuming the MakerDAO is not destroyed, the losses from such an attack would be bared by the holders of the MKR token. Such an attack would cause a depreciation in the value of Dai, leading to it under-tracking its target value. In this situation, the MKR tokens are used to purchase Dai thereby decreasing the supply and increasing the value. The holder of the MKR token would therefore bare the losses caused by such an attack.