Bonus Assignment 1: Blockchain Technology

In this assignment you are asked to:

- Implement an off-chain payment channel.
- Answer some questions about your task and blockchain technology.

This assignment is not theory-heavy, instead it aims to provide an exciting glimpse of the cool things you can do with cryptography today.

Important Note You will have to upload three files on Fire: a modified `interface.js` off-chain frontend file, a modified `StateChannel.solo` smart contract file, and a `report.pdf` file with answers in English to non-coding questions.

Background

An exciting application of cryptography which has emerged in recent years is distributed ledger technology (aka "blockchain technology" or crypto). Requiring only the basic cryptographic primitives of hashes and digital signatures, blockchains provide computer scientists with a fundamentally new capability: a publicly available decentralized database which is resistant to tampering and censorship. That is achieved by replicating the database (or "ledger") across thousands of nodes; a consensus protocol is used to determine who may apply the next update. By storing programs that users can interact with in the blockchain database, blockchains give users access to "programmable money", allowing payment or even the execution of complex legal/financial contracts without the need for approval by a third party. However, the massively replicated nature of blockchains means that the ledger is computationally costly to update and update capacity is limited, leading to high transaction fees. In response, developers have invented protocols for moving state and computation off the blockchain, while still making use of its consistency to protect the state. The first and simplest example of this is the off-chain payment channel.

Off-chain payment channels The default way to make payments from one user (let us call her Alice) to another (Bob) on a blockchain is to run a signed transaction on it, authorizing the transfer of some cryptocurrency from Alice to Bob. Payment channels instead allow two or more users to lock some money in an on-chain program (termed "smart contract"), which will eventually disburse the funds given a message approved by both/all parties. By signing such a message off-chain and sending it to the other party, the user can pay the other without performing any on-chain work. That reduces the execution cost of a payment from the order of dollars (see etherscan for the current Ethereum transaction cost) to effectively zero, and the latency for performing a payment from minutes to milliseconds.

Assignment Body

Setup and tutorial

1. Install node.js from here. After this, you should have **node** and **npm** (the node package manager) available on your path.
2. Install the truffle Ethereum client and the Ganache private blockchain GUI. You should now be able to start the Ganache GUI, and **truffle** should be available on your path.

3. Create your Truffle project in a new folder by running **truffle init** in it. Add the **2_deploy_state_channel.js** file provided by us to the **migrations** folder in your new Truffle directory. Add the Solidity skeleton code to the **contracts** folder, and `interface.js`.js to the root of the Truffle directory (where **truffle-config.js** is).

4. Start your Ganache GUI and create a new workspace. In the Chain tab, set the gas price to 0. That ensures that accounts aren't charged a fee whenever they send a transaction, which makes it easier to test whether the payment channel is working correctly.

While the Ganache private blockchain GUI is running and set to your newly created workspace, you should be able to run **truffle deploy** to compile the `StateChannel.solo` contract and upload it to the chain. When you edit the StateChannel, you'll need to run it again. Then run **truffle console** in your Truffle project folder to connect to the chain with a JavaScript shell. The **StateChannel** variable should be set it you deployed the contract successfully. That can be checked by entering the expression:

```
typeof StateChannel
```

which should not throw a `ReferenceError`. Check that the **ethereumjs-util** package is installed using:

```
eu = require("ethereumjs-util")
```

If not, you need to install it using **npm** and then restart the console.
Get the array of (addresses of) accounts created by Ganache using:

```
accs = await web3.eth.getAccounts()
```

Note that JavaScript doesn't have blocking calls; functions which interact with remote state (such as the blockchain) return promises, a type of object which can be `await`ed to get its eventual value. To bind the result of a promise to a variable in the console, use await as above.

Then assign the first two addresses to the global variables `Alice` and `Bob`:

```
Alice = accs[0]; Bob = accs[1]
```

You can check the ether balance of Alice using `web3.eth.getBalance(Alice)`. Creating new payment channels, depositing to them and withdrawing from them will change the balances of Alice and Bob. You can find documentation on the **web3** library used for interfacing with Ethereum blockchains via JavaScript here.

Now go to the Ganache GUI and look at the accounts. On the right-hand side of each row is a key icon; click it to bring up the private key of the corresponding account. Copy the private keys of the first two accounts (Alice and Bob) as strings `"<yourKeyHere>"` to the variables `AliceSK` and `BobSK`.

Load the `interface.js` file and pass in the StateChannel class like this:

```
i = require("./interface.js"); i.init(StateChannel)
```

You will need to edit the interface file, but requiring the same module twice will give the same result even if you modify the file in-between. To get around that, we've provided a utility function `reload` to the interface which lets you remove the outdated code from `require`'s cache and load the latest version.

The interface skeleton contains a number of other useful functions predefined; for example, you can get the public key (i.e. address) corresponding to an Ethereum private key using the function `pubKeyOf`. As a sanity check, test that:

```
i.pubKeyOf(AliceSK) == Alice
```

If that returns false, you may have made a mistake when copying `AliceSK`.

Now try creating and interacting with a payment channel contract:

```
sc = await StateChannel.new(Bob,{value: 100})
```

That should create a new state channel from the default transaction sender (the first account, i.e. Alice) with the counterparty Bob and filled with 100 wei (wei is the smallest, indivisible unit of ether; one ether is a billion billion wei). Note the object argument; you can add an object `{from: addr, value: x}` to any smart contract method and it will set the address that calls the method to `addr` and the value they deposit to `x`. Now you can interact with that state channel by calling its methods, for example:

```
sc.Alice() //Should return Alice
sc.Bob() //The counterparty, should be Bob
sc.deposit({from: Alice, value: x}) //Alice deposits x wei
```

That lets you debug the smart contract as you're editing it. You will also need to complete the off-chain protocol for payment channels defined in `interface.js`. The function **newConvo(AliceSK,BobSK,value)** creates a new instance of the protocol between Alice and Bob. The convo object is a simplified model of a real payment channel protocol; in real code Alice and Bob's secret keys would of course not be stored on the same computer!

The basic payment channel protocol

In this lab we'll only allow off-chain payments of the native cryptocurrency ether, in order to keep the off-chain state simple. The off-chain state will be the OffChainLedger struct defined in the Solidity skeleton code; helper functions for signing off-chain ledgers have been provided in the JavaScript skeleton code.

> **Task 1** Implement the method **deposit()** in the StateChannel smart contract. It should increment Alice's balance by the amount of wei deposited if the caller is Alice, and Bob's balance otherwise. The signature of the function has already been provided. *(Tip: the address of the caller is referred to as **msg.sender**, and the amount of wei deposited **msg.value**)*

> **Task 2** Implement the startWithdrawal(L, sig) and completeWithdrawal() Solidity methods. The former accepts an off-chain ledger L and the other party's signature sig.

The caller doesn't need to provide their signature, since they must sign the transaction that makes the call anyway; reject calls unless the caller ($msg.caller$) is Alice or Bob. Alice should only be allowed to upload Bob's signed ledger and vice versa (this prevents an attack). The storage variable `tentativeBalanceDelta` should record the amount that will be transferred from Alice to Bob if the withdrawal is completed successfully. The `deadline` variable should record the last time the withdrawal attempt may be "appealed" before it becomes final; it should be set to the time that the withdrawal attempt is made plus the `WARNING_PERIOD` constant (which you may set to any value for testing). Tip: the `now()` inbuilt function in Solidity returns the current UNIX time in seconds.

Ledgers with higher nonces should dominate lower ones; if the other party uploads a later signed ledger while a withdrawal is in progress, the current withdrawal attempt should be overwritten with the new one. If the uploaded ledger is older (i.e. its nonce is lower), then it should instead be ignored. When a withdrawal is complete (i.e. the deadline is nonzero and passed), startWithdrawal should fail and completeWithdrawal should immediately pay out the locked funds according to the tentative ledger's state.

Make use of the helper functions provided in the Solidity module, for example the **verifySignature** method. When writing **startWithdrawal**, you may take inspiration from the **withdrawWithoutSignature** method.

Task 3 Implement the off-chain logic for **startWithdrawal** and **completeWithdrawal** in the correspondingly named functions in `interface.js`. The comments in the file specify correct behavior.

Task 4 Now that startWithdrawal and completeWithdrawal are implemented, implement the off-chain method **pay(ix,value,c)** in the JavaScript code skeleton. Use the helper functions provided. Read the comments for its specification.

Task 5 Given the format of the off-chain ledger (that the users sign), what happens if Alice creates a payment channel with both Bob and Charles? Explain an attack that Charles can use against Alice and Bob. Write your answers in `report.pdf`.

Task 6 How would you mitigate the attack from the previous task? Write your thoughts in the `report.pdf` file.

If you've implemented the payment channel protocol directly, it should be impossible for either party to steal funds from the other or lock funds indefinitely, *if both parties are responsive.* That is, if one party sees the other attempt to make a withdrawal using an outdated off-chain state, they must be able to respond in time with the latest one.

Solving **Tasks 1-6** is worth one point. For the second point, you'll have to innovate!

Towards an advanced payment channel

A problem with the payment channel you've implemented so far is that withdrawals take a minimum amount of time, even if both Alice and Bob want to withdraw immediately. That may cost Alice and Bob valuable arbitrage opportunities!

Task 7 Enhance your payment channel to allow immediate withdrawal of *part of the funds* given unanimous agreement by the parties and implement support for this functionality in the off-chain JavaScript code `interface.js`.

You may need to add new state to the off-chain and on-chain code, and will certainly need to extend the `OffChainLedger` struct and modify the `signOCL` function accordingly.

Task 8 Why is your new protocol secure? Include an explanation of why this modification prevents either party from stealing funds from the other if they're able to respond to on-chain transactions within some maximum time T in `report.pdf`.