CS251 Final exam

George Charles Davis

TOTAL POINTS

91 / 100

QUESTION 1

Problem 1 21 pts

1.1 A 3 / 3

✓ **- 0 pts** *Correct*

- 1.5 pts Did not explain how changing the transaction value would cause the transaction to be rejected

- 2 pts Did not mention the invalidation of Alice's signature

- 3 pts Incorrect/Blank

1.2 B 3 / 3

✓ **- 0 pts** *Correct*

- 1 pts Wrong type of attack specified

- 1.5 pts Didn't mention that replay/double-spending attacks could happen

- 2 pts Didn't state that transactions can be posted on multiple chains without a ChainID

- 3 pts Incorrect

1.3 C 3 / 3

✓ **- 0 pts** *Correct*

- 2 pts Didn't mention energy costs associated with PoW

- 3 pts Incorrect/Blank

1.4 D 3 / 3

✓ **- 0 pts** *Correct*

- 3 pts no submission / wrong answer

- 2 pts miss the important idea: happens in a single transaction

1.5 E 3 / 3

✓ **- 0 pts** *Correct*

- 1 pts wrong definition of external function

- 1 pts wrong definition of public function

- 1 pts A public function must first copy all the calldata to memory, and it therefore costs more gas than an external function

- 3 pts no submission / wrong answer

1.6 F 3 / 3

✓ **- 0 pts** *Correct*

- 3 pts No answer

- 2 pts The correct answer is only (B). Both (A) and (C) are collision resistant.

1.7 G 3 / 3

✓ **- 0 pts** *Correct*

- 3 pts No / incorrect answer

- 2 pts no / incorrect justification

QUESTION 2

Problem 2 16 pts

2.1 A 8 / 8

✓ **- 0 pts** *There was a missing assumption in the problem statement ($$f<n/2$$), so we gave everyone full credit for this problem.*

- 8 pts No answer.

2.2 B **4 / 8**

- 0 pts Correct

- 8 pts no answer

- 4 pts The goal is to build a new protocol Pi' using Pi as a black box. The idea is to have each of the three given parties emulate n/3 parties in its head, so overall the protocol has n virtual parties (three groups of n/3). Since at most one real party is corrupt, we know that at most 4 virtual parties are corrupt.

✓ **- 4 pts** *You are given a protocol that works for n parties (say n=12) and your goal is to build a protocol that works for 3 parties. The idea is to have each of the three given parties emulate four parties in its head, so overall the protocol has 12 virtual parties (three groups of four). Since at most one real party is corrupt, we know that at most 4 virtual parties are corrupt.*

- 6 pts The protocol is under specified. The idea is to have each of the three given parties emulate n/3 parties in its head, so overall the protocol has n virtual parties (three groups of n/3). Since at most one real party is corrupt, we know that at most 4 virtual parties are corrupt.

- 2 pts Who runs the extra node(s)? The idea is to have each of the three given parties emulate n/3 parties in its head, so overall the protocol has n virtual parties (three groups of n/3).

QUESTION 3

Problem 3 13 pts

3.1 A **5 / 5**

✓ **- 0 pts** *Correct*

- 1 pts Answer is not "less tokens"

- 2 pts Function minor mistake

- 4 pts No function provided or major mistake

- 5 pts Incorrect or blank

3.2 B **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect or blank

3.3 C **2 / 2**

✓ **- 0 pts** *Correct*

- 2 pts Incorrect/blank

3.4 D **2 / 2**

✓ **- 0 pts** *Correct/Demonstrates Understanding*

- 1 pts Attack Described is not a sandwich attack and/or still requires Sam

- 2 pts Incorrect/Blank

3.5 E **3 / 3**

✓ **- 0 pts** *Correct*

- 3 pts Blank\Incorrect

QUESTION 4

Problem 4 20 pts

4.1 A **7 / 7**

✓ **- 0 pts** *Correct*

- 7 pts Incorret/No submission

- 1 pts Minor error: Insufficient explanation-

Alice's second createBid call can frontrun acceptBid with a higher maxPriorityFee. For the attack to succeed, acceptBid should be posted before createBid.

- 2 pts The seller has already called acceptBid. Alice can call createBid even after acceptBid has been called, by frontrunning, i.e. with a higher maxPriorityFee so that the new createBid is executed first

- 3 pts Technically, Bob can know if Alice modified the mapping value to 1 if Alice's createBid is executed on-chain. Alternatively, Alice can check the mempool to see if createBid has been called and and then call createBid to frontrun the transaction with a higher maxPriorityFee

- 5 pts Alice can call createBid once again after acceptBid has been added to the mempool. By frontrunning, i.e. with a higher maxPriorityFee so that the new createBid is executed first

- 3 pts Click here to replace this description.

- 1 pts Click here to replace this description.

- 2 pts Did not call createBid again to change the mapping

- 3 pts Click here to replace this description.

- 4 pts Click here to replace this description.

4.2 B **6 / 7**

- 0 pts Correct

- 7 pts Incorrect/Incomplete

- 2 pts bidder is not Alice

- 1 pts erc721 token id is not 34

✓ **- 1 pts** *price is not 53*

- 2 pts Did not swap order of the token names

- 4 pts Invalid Params/Did not specify acceptBid call

- 1 pts minor error

- 2 pts Click here to replace this description.

- 5 pts Click here to replace this description.

4.3 C **6 / 6**

✓ **- 0 pts** *Correct*

- 2 pts Did not specify an alternative implementation

- 4 pts Error

- 6 pts Incorrect/ No submission

- 2 pts Click here to replace this description.

QUESTION 5

Problem 5 20 pts

5.1 A **4 / 4**

✓ **- 0 pts** *Correct*

- 1 pts Doesn't discuss how this leads to lower B than its true value.

- 2 pts Partial Credit

- 4 pts Nothing/Incorrect

5.2 B **0 / 4**

- 0 pts Correct. Links user identifier with their balance (e.g. concatenates them in the leaf).

- 2 pts Good try. Still doesn't link balances to users.

- 2 pts Good try, but still doesn't prevent the attack. Still doesn't link balances to users.

- 2 pts Good try, but assumes that users can communicate, or have to interact, or know how many of them exist, or do anything past verifying

their own proofs. Still doesn't link balances to users.

- 2 pts Good try, but assumes users know how many of them exist. Still doesn't link balances to users.

- 2 pts Doesn't specify how to use the added user identifier within the proof. Still doesn't link balances to users.

- 3 pts Answer too vague/high-level. Still doesn't link balances to users.

✓ **- 4 pts** *Incorrect/Incomplete. Still doesn't link balances to users. Still doesn't prevent the attack.*

5.3 C **6 / 6**

✓ **- 0 pts** *Correct*

- 2 pts Minor Error

- 4 pts Major Error

- 6 pts Blank / Not Attempted

5.4 D **6 / 6**

✓ **- 0 pts** *Correct*

- 0 pts Correct given 5c

- 2 pts Minor Error

- 4 pts Major Error

- 6 pts Blank / Not Attempted

QUESTION 6

Problem 6 10 pts

6.1 A **3 / 3**

✓ **- 0 pts** *Correct*

- 2 pts Incorrect (duplicate addresses would imply a collision on keccak256)

- 3 pts Empty

6.2 B **3 / 3**

✓ **- 0 pts** *Correct*

- 1 pts Incorrectly states that the contract at Y must also self-destruct

- 1 pts Insufficient explanation

- 3 pts Empty

- 1 pts Does not mention that the contract at X must self-destruct

6.3 C **4 / 4**

✓ **- 0 pts** *Correct*

- 1 pts Does not mention that C also must self-destruct

- 1 pts Does not mention that B also must self-destruct

- 4 pts Empty

Problem 1 - All Over

Part A

A) Suppose Alice wants to pay Bob 2 ETH via an Ethereum transaction. She sends a transaction to the Ethereum network whose value is 2×10^{18} Wei. What prevents a malicious validator from changing the transaction value to 3×10^{18} Wei, thereby making Alice pay 3 ETH to Bob?

Alice signs her transaction tx using her private key pk. Anyone can use her public key pk to verify that her signature sig is correct for the given transaction. A malicious validator can change the transaction but will be unable to create a signature for it that can be validated using Alice's public key. If they try to promote this invalid transaction, they will get slashed and lose ETH.

Part B

B) There are many active EVM blockchains: the Ethereum mainnet, Ethereum testnets (e.g., Goerli), other EVM chains (e.g., Polygon mainnet), several L2 chains and their testnets, and many others. Each chain is identified by a ChainID: Ethereum mainnet is 1, Goerli is 5, Polygon mainnet is 137, etc. Every EVM transaction contains the ChainID of the intended chain. What would go wrong if the ChainID field were not included in the transaction data?
Hint: Recall that on all EVM chains, Alice's address is derived from a hash of her public key, and nothing else.

The chain ID must be included in the transaction so that a signed transaction can only be used for the specified chain. If it were not included, a malicious actor could effectively steal funds from a user through "replay attacks". For example, suppose Alice signs a transaction to send 10 ETH to Bob on a test-net (not the mainnet). Bob could then take that signed transaction and post it on the mainnet, resulting in Alice actually sending 10 real ETH to Bob.

By including the chain ID in the transaction, miners can see that a transaction is meant for a different chain and not post it in a block.

1.1 A **3 / 3**

✓ **- 0 pts** *Correct*

- 1.5 pts Did not explain how changing the transaction value would cause the transaction to be rejected

- 2 pts Did not mention the invalidation of Alice's signature

- 3 pts Incorrect/Blank

Problem 1 - All Over

Part A

A) Suppose Alice wants to pay Bob 2 ETH via an Ethereum transaction. She sends a transaction to the Ethereum network whose value is 2×10^{18} Wei. What prevents a malicious validator from changing the transaction value to 3×10^{18} Wei, thereby making Alice pay 3 ETH to Bob?

Alice signs her transaction tx using her private key pk. Anyone can use her public key pk to verify that her signature sig is correct for the given transaction. A malicious validator can change the transaction but will be unable to create a signature for it that can be validated using Alice's public key. If they try to promote this invalid transaction, they will get slashed and lose ETH.

Part B

B) There are many active EVM blockchains: the Ethereum mainnet, Ethereum testnets (e.g., Goerli), other EVM chains (e.g., Polygon mainnet), several L2 chains and their testnets, and many others. Each chain is identified by a ChainID: Ethereum mainnet is 1, Goerli is 5, Polygon mainnet is 137, etc. Every EVM transaction contains the ChainID of the intended chain. What would go wrong if the ChainID field were not included in the transaction data?
Hint: Recall that on all EVM chains, Alice's address is derived from a hash of her public key, and nothing else.

The chain ID must be included in the transaction so that a signed transaction can only be used for the specified chain. If it were not included, a malicious actor could effectively steal funds from a user through "replay attacks". For example, suppose Alice signs a transaction to send 10 ETH to Bob on a test-net (not the mainnet). Bob could then take that signed transaction and post it on the mainnet, resulting in Alice actually sending 10 real ETH to Bob.

By including the chain ID in the transaction, miners can see that a transaction is meant for a different chain and not post it in a block.

1.2 B **3 / 3**

✓ **- 0 pts** *Correct*

- 1 pts Wrong type of attack specified

- 1.5 pts Didn't mention that replay/double-spending attacks could happen

- 2 pts Didn't state that transactions can be posted on multiple chains without a ChainID

- 3 pts Incorrect

Part C

C) What is the main reason that Ethereum moved from Proof of Work conensus to Proof of Stake consensus?

Ethereum moved from PoW to PoS in order to waste less energy (mining is energy-intensive) and to be more scalable (support more transactions per second).

Part D

D) Briefly explain what is a flash loan and give one application.

A flash loan is an atomic transaction where a user can borrow a large amount of tokens, do something with them, and repay them (with interest) all at once. There is no lender risk and the borrower doesn't need collateral.

One application is risk-free arbitrage. For example, Alice can take a very large loan to arbitrage conversion rate differences on different exchanges. She could borrow 1 million USDC, exchange it for DAI on one exchange, and then swap that DAI back for USDC on another exchange with a lower exchange rate. After paying back the USDC she initially borrowed (plus interest / fee), she will have a small profit.

Part E

E) In Solidity, what is the difference between an `external` function and a `public` function? Which one typically costs more gas to call?

In `external` functions, the functions can read the arguments directly from calldata. In `public` functions, the arguments are copied into memory, which is much more expensive gas-wise; however, this allows `public` functions to also be used internally (whereas `external` functions can only be called externally). Therefore, `public` functions are typically more expensive to call.

Part F

1.3 C **3 / 3**

✓ **- 0 pts** *Correct*

- 2 pts Didn't mention energy costs associated with PoW

- 3 pts Incorrect/Blank

Part C

C) What is the main reason that Ethereum moved from Proof of Work conensus to Proof of Stake consensus?

Ethereum moved from PoW to PoS in order to waste less energy (mining is energy-intensive) and to be more scalable (support more transactions per second).

Part D

D) Briefly explain what is a flash loan and give one application.

A flash loan is an atomic transaction where a user can borrow a large amount of tokens, do something with them, and repay them (with interest) all at once. There is no lender risk and the borrower doesn't need collateral.

One application is risk-free arbitrage. For example, Alice can take a very large loan to arbitrage conversion rate differences on different exchanges. She could borrow 1 million USDC, exchange it for DAI on one exchange, and then swap that DAI back for USDC on another exchange with a lower exchange rate. After paying back the USDC she initially borrowed (plus interest / fee), she will have a small profit.

Part E

E) In Solidity, what is the difference between an `external` function and a `public` function? Which one typically costs more gas to call?

In `external` functions, the functions can read the arguments directly from calldata. In `public` functions, the arguments are copied into memory, which is much more expensive gas-wise; however, this allows `public` functions to also be used internally (whereas `external` functions can only be called externally). Therefore, `public` functions are typically more expensive to call.

Part F

1.4 D **3 / 3**

✓ **- 0 pts** *Correct*

- 3 pts no submission / wrong answer

- 2 pts miss the important idea: happens in a single transaction

Part C

C) What is the main reason that Ethereum moved from Proof of Work conensus to Proof of Stake consensus?

Ethereum moved from PoW to PoS in order to waste less energy (mining is energy-intensive) and to be more scalable (support more transactions per second).

Part D

D) Briefly explain what is a flash loan and give one application.

A flash loan is an atomic transaction where a user can borrow a large amount of tokens, do something with them, and repay them (with interest) all at once. There is no lender risk and the borrower doesn't need collateral.

One application is risk-free arbitrage. For example, Alice can take a very large loan to arbitrage conversion rate differences on different exchanges. She could borrow 1 million USDC, exchange it for DAI on one exchange, and then swap that DAI back for USDC on another exchange with a lower exchange rate. After paying back the USDC she initially borrowed (plus interest / fee), she will have a small profit.

Part E

E) In Solidity, what is the difference between an `external` function and a `public` function? Which one typically costs more gas to call?

In `external` functions, the functions can read the arguments directly from calldata. In `public` functions, the arguments are copied into memory, which is much more expensive gas-wise; however, this allows `public` functions to also be used internally (whereas `external` functions can only be called externally). Therefore, `public` functions are typically more expensive to call.

Part F

1.5 E **3 / 3**

✓ **- 0 pts** *Correct*

- 1 pts wrong definition of external function

- 1 pts wrong definition of public function

- 1 pts A public function must first copy all the calldata to memory, and it therefore costs more gas than an external function

- 3 pts no submission / wrong answer

Part C

C) What is the main reason that Ethereum moved from Proof of Work conensus to Proof of Stake consensus?

Ethereum moved from PoW to PoS in order to waste less energy (mining is energy-intensive) and to be more scalable (support more transactions per second).

Part D

D) Briefly explain what is a flash loan and give one application.

A flash loan is an atomic transaction where a user can borrow a large amount of tokens, do something with them, and repay them (with interest) all at once. There is no lender risk and the borrower doesn't need collateral.

One application is risk-free arbitrage. For example, Alice can take a very large loan to arbitrage conversion rate differences on different exchanges. She could borrow 1 million USDC, exchange it for DAI on one exchange, and then swap that DAI back for USDC on another exchange with a lower exchange rate. After paying back the USDC she initially borrowed (plus interest / fee), she will have a small profit.

Part E

E) In Solidity, what is the difference between an `external` function and a `public` function? Which one typically costs more gas to call?

In `external` functions, the functions can read the arguments directly from calldata. In `public` functions, the arguments are copied into memory, which is much more expensive gas-wise; however, this allows `public` functions to also be used internally (whereas `external` functions can only be called externally). Therefore, `public` functions are typically more expensive to call.

Part F

F) In class we used collision resistant hash functions to construct commitment schemes. Let $H : \mathcal{X} \to \mathcal{Y}$ be a collision resistant hash function, where $\mathcal{X} = \{0,1\}^n$ and $\mathcal{Y} = \{0,1\}^t$ for some $t < n$. Which of the following three hash functions is not collision resistant (circle all that apply):

A. $H_1(x) := H(x)\|0$ for $x \in \mathcal{X}$

B. $H_2(x\|b) := H(x)$ for $x \in \mathcal{X}$ and $b \in \{0,1\}$

C. $H_3(x\|b) := H(x)\|b$ for $x \in \mathcal{X}$ and $b \in \{0,1\}$

Recall that $\|$ indicates concatenation, for example, $010\|1$ is 0101.

B is not collision resistant because $H_2(x||0) = H(x)$ and $H_x(x||1) = H(x)$.

Part G

G) The EVM supports both volatile and non-volatile memory. Data is written to volatile memory using the `MSTORE` instruction, and is written to non-volatile memory using the `SSTORE` instruction. The minimum gas cost of one of these instructions is much higher than the other. Which is higher and why is it higher?

Writing to persistent memory using `SSTORE` is more expensive because this the opcodes that store / load persistent data from the data array cost more gas (because data is stored in / loaded from the blockchain). Loading from / storing in memory is cheaper because the data is lost / reset after smart contract execution.

Problem 2 - Consensus

Consider n parties, where $n \geq 3$, and where one of the parties is designated as a *sender*. The *sender* has a bit $b \in \{0,1\}$. A *broadcast protocol* is a protocol where the parties send messages to one another, and eventually every party outputs a bit b_i, for $i = 1, \ldots, n$, or outputs nothing. The parties are connected by a synchronous network.

- We say that the protocol has **consistency** if for every two honest parties, if one party outputs b' and the other outputs b'', then $b' = b''$.
- We say that the protocol has **validity** if when the *sender* is honest, the output of all honest parties is equal to the *sender*'s input bit b.
- We say that the protocol has **totality** if whenever some honest party outputs a bit, then eventually all honest parties output a bit.

A *reliable broadcast protocol* (RBC) is a broadcast protocol that satisfies all three properties.

1.6 F **3 / 3**

✓ **- 0 pts** *Correct*

- 3 pts No answer

- 2 pts The correct answer is only (B). Both (A) and (C) are collision resistant.

F) In class we used collision resistant hash functions to construct commitment schemes. Let $H : \mathcal{X} \to \mathcal{Y}$ be a collision resistant hash function, where $\mathcal{X} = \{0,1\}^n$ and $\mathcal{Y} = \{0,1\}^t$ for some $t < n$. Which of the following three hash functions is not collision resistant (circle all that apply):

A. $H_1(x) := H(x)\|0$ for $x \in \mathcal{X}$

B. $H_2(x\|b) := H(x)$ for $x \in \mathcal{X}$ and $b \in \{0,1\}$

C. $H_3(x\|b) := H(x)\|b$ for $x \in \mathcal{X}$ and $b \in \{0,1\}$

Recall that $\|$ indicates concatenation, for example, $010\|1$ is 0101.

B is not collision resistant because $H_2(x||0) = H(x)$ and $H_x(x||1) = H(x)$.

Part G

G) The EVM supports both volatile and non-volatile memory. Data is written to volatile memory using the `MSTORE` instruction, and is written to non-volatile memory using the `SSTORE` instruction. The minimum gas cost of one of these instructions is much higher than the other. Which is higher and why is it higher?

Writing to persistent memory using `SSTORE` is more expensive because this the opcodes that store / load persistent data from the data array cost more gas (because data is stored in / loaded from the blockchain). Loading from / storing in memory is cheaper because the data is lost / reset after smart contract execution.

Problem 2 - Consensus

Consider n parties, where $n \geq 3$, and where one of the parties is designated as a *sender*. The *sender* has a bit $b \in \{0,1\}$. A *broadcast protocol* is a protocol where the parties send messages to one another, and eventually every party outputs a bit b_i, for $i = 1, \ldots, n$, or outputs nothing. The parties are connected by a synchronous network.

- We say that the protocol has **consistency** if for every two honest parties, if one party outputs b' and the other outputs b'', then $b' = b''$.
- We say that the protocol has **validity** if when the *sender* is honest, the output of all honest parties is equal to the *sender*'s input bit b.
- We say that the protocol has **totality** if whenever some honest party outputs a bit, then eventually all honest parties output a bit.

A *reliable broadcast protocol* (RBC) is a broadcast protocol that satisfies all three properties.

1.7 G **3 / 3**

✓ **- 0 pts** *Correct*

- 3 pts No / incorrect answer

- 2 pts no / incorrect justification

Part A

A) Suppose we have a broadcast protocol Π among n parties that is reliable as long as at most f of the parties are corrupt. Use protocol Π to construct a reliable broadcast protocol Π' for $m > n$ parties that can tolerate up to f corrupt parties. Make sure to explain why your Π' has consistency, validity, and totality.

#todo

Part B

B) Continuing with part (B), suppose the given broadcast protocol Π is reliable for $n > 3$ parties as long as at most $f = n/3$ are corrupt (you may assume n is divisible by three). Use Π to construct a broadcast protocol for exactly three parties that is reliable as long as at most one of the three parties is corrupt. Make sure to explain why your protocol has consistency, validity, and totality.

- The sender sends its input bit and signature to all other parties.
- The two non-sender parties echo what they heard to all other non-sender parties (i.e., to each other). If a non-sender party didn't hear anything from the sender, or if the sender's message is malformed (e.g., bad signature), then it does nothing during this step.
- The two non-sender parties review the messages from each other; they should each have received one message from *sender* and one from the other non-sender party. They compare their messages:
    - if the messages are correctly signed by the sender but for different bits, the party outputs 0
    - if more than one message was received from the sender or from the non-sender party, the party outputs zero
- Otherwise, the non-sender parties output the matching bits

This is valid because the outputs of all non-sender parties will be equal to the sender's bit.

It is consistent because if both parties are honest, they will output the same bit.

2.1 A **8 / 8**

✓ **- 0 pts** *There was a missing assumption in the problem statement ($$f<n/2$$), so we gave everyone full credit for this problem.*

- 8 pts No answer.

Part A

A) Suppose we have a broadcast protocol Π among n parties that is reliable as long as at most f of the parties are corrupt. Use protocol Π to construct a reliable broadcast protocol Π' for $m > n$ parties that can tolerate up to f corrupt parties. Make sure to explain why your Π' has consistency, validity, and totality.

#todo

Part B

B) Continuing with part (B), suppose the given broadcast protocol Π is reliable for $n > 3$ parties as long as at most $f = n/3$ are corrupt (you may assume n is divisible by three). Use Π to construct a broadcast protocol for exactly three parties that is reliable as long as at most one of the three parties is corrupt. Make sure to explain why your protocol has consistency, validity, and totality.

- The sender sends its input bit and signature to all other parties.
- The two non-sender parties echo what they heard to all other non-sender parties (i.e., to each other). If a non-sender party didn't hear anything from the sender, or if the sender's message is malformed (e.g., bad signature), then it does nothing during this step.
- The two non-sender parties review the messages from each other; they should each have received one message from *sender* and one from the other non-sender party. They compare their messages:
    - if the messages are correctly signed by the sender but for different bits, the party outputs 0
    - if more than one message was received from the sender or from the non-sender party, the party outputs zero
- Otherwise, the non-sender parties output the matching bits

This is valid because the outputs of all non-sender parties will be equal to the sender's bit.

It is consistent because if both parties are honest, they will output the same bit.

2.2 B **4 / 8**

- 0 pts Correct

- 8 pts no answer

- 4 pts The goal is to build a new protocol Pi' using Pi as a black box. The idea is to have each of the three given parties emulate n/3 parties in its head, so overall the protocol has n virtual parties (three groups of n/3). Since at most one real party is corrupt, we know that at most 4 virtual parties are corrupt.

✓ **- 4 pts** *You are given a protocol that works for n parties (say n=12) and your goal is to build a protocol that works for 3 parties. The idea is to have each of the three given parties emulate four parties in its head, so overall the protocol has 12 virtual parties (three groups of four). Since at most one real party is corrupt, we know that at most 4 virtual parties are corrupt.*

- 6 pts The protocol is under specified. The idea is to have each of the three given parties emulate n/3 parties in its head, so overall the protocol has n virtual parties (three groups of n/3). Since at most one real party is corrupt, we know that at most 4 virtual parties are corrupt.

- 2 pts Who runs the extra node(s)? The idea is to have each of the three given parties emulate n/3 parties in its head, so overall the protocol has n virtual parties (three groups of n/3).

Problem 3 - Sandwich Attacks

Consider two assets X and Y on the Uniswap exchange. The X pool contains x tokens of type X and the Y pool contains y tokens of type Y. Recall that Uniswap v2 ensures that $x \cdot y = k$ for some constant k. Suppose that Uniswap charges no fees.

Alice submits a transaction Tx that sends δx tokens of type X to Uniswap, for some $\delta \geq 0$ (here δx means δ times x). When Tx executes, Uniswap will send back $\delta' y$ tokens of type Y to Alice, where $\delta' = \frac{\delta}{1+\delta}$. This ensures that the constant product is maintained.

Searcher Sam sees Alice's transaction in the mempool and decides to execute a sandwitch attack. Sam issues two transactions Tx_1 and Tx_2 and arranges with the current block proposer that Tx_1 will appear before Alice's transaction in the proposed block and Tx_2 will appear after.

Part A

A) Suppose Sam's Tx_1 sends ϵx tokens of type X to Uniswap, for some $\epsilon \geq 0$. Now, when Alice's Tx executes, she will receive back $\delta'' y$ tokens of type Y. What is δ'' as a function of ϵ and δ? Is she getting more or less tokens of type Y than before?

When Sam sends ϵx tokens to Uniswap, Uniswap sends back $\epsilon' y$ tokens, where $\epsilon' = \frac{\epsilon}{1+\epsilon}$. The exchange now has $x(1+\epsilon)$ tokens of type X and $y(1-\frac{\epsilon}{1+\epsilon})$ tokens of type Y. This increases the ratio of X tokens to Y tokens (Y tokens are now more valuable.

When Alice submits her transaction, she will get back $\delta'' y$ tokens where $\delta'' = (\frac{\delta}{1+\delta})(1-\frac{\epsilon}{1+\epsilon}) = \delta'(1-\frac{\epsilon}{1+\epsilon})$. This is fewer tokens than before.

Part B

B) Sam's Tx_2, which executes after Alice, will send just enough Y tokens to Uniswap to get back his ϵx tokens of type X that he spent in Tx_1. Thus, his position in X tokens is unchanged after the block executes. How did Sam make a profit from this?

When Sam did Tx_1, he altered the exchange rate of X/Y and made Y tokens more valuable. Alice then buys Y tokens at a higher cost and further raises the value of Y tokens. Sam then sells some of his Y tokens to get back his ϵx tokens of type X from his first transaction. Because Y tokens are more valuable, he doesn't have to sell as many to get back his original number of X tokens. This means that Sam now has his

3.1 A **5 / 5**

✓ **- 0 pts** *Correct*

- 1 pts Answer is not "less tokens"

- 2 pts Function minor mistake

- 4 pts No function provided or major mistake

- 5 pts Incorrect or blank

Problem 3 - Sandwich Attacks

Consider two assets X and Y on the Uniswap exchange. The X pool contains x tokens of type X and the Y pool contains y tokens of type Y. Recall that Uniswap v2 ensures that $x \cdot y = k$ for some constant k. Suppose that Uniswap charges no fees.

Alice submits a transaction Tx that sends δx tokens of type X to Uniswap, for some $\delta \geq 0$ (here δx means δ times x). When Tx executes, Uniswap will send back $\delta' y$ tokens of type Y to Alice, where $\delta' = \frac{\delta}{1+\delta}$. This ensures that the constant product is maintained.

Searcher Sam sees Alice's transaction in the mempool and decides to execute a sandwitch attack. Sam issues two transactions Tx_1 and Tx_2 and arranges with the current block proposer that Tx_1 will appear before Alice's transaction in the proposed block and Tx_2 will appear after.

Part A

A) Suppose Sam's Tx_1 sends ϵx tokens of type X to Uniswap, for some $\epsilon \geq 0$. Now, when Alice's Tx executes, she will receive back $\delta'' y$ tokens of type Y. What is δ'' as a function of ϵ and δ? Is she getting more or less tokens of type Y than before?

When Sam sends ϵx tokens to Uniswap, Uniswap sends back $\epsilon' y$ tokens, where $\epsilon' = \frac{\epsilon}{1+\epsilon}$. The exchange now has $x(1+\epsilon)$ tokens of type X and $y(1-\frac{\epsilon}{1+\epsilon})$ tokens of type Y. This increases the ratio of X tokens to Y tokens (Y tokens are now more valuable.

When Alice submits her transaction, she will get back $\delta'' y$ tokens where $\delta'' = (\frac{\delta}{1+\delta})(1-\frac{\epsilon}{1+\epsilon}) = \delta'(1-\frac{\epsilon}{1+\epsilon})$. This is fewer tokens than before.

Part B

B) Sam's Tx_2, which executes after Alice, will send just enough Y tokens to Uniswap to get back his ϵx tokens of type X that he spent in Tx_1. Thus, his position in X tokens is unchanged after the block executes. How did Sam make a profit from this?

When Sam did Tx_1, he altered the exchange rate of X/Y and made Y tokens more valuable. Alice then buys Y tokens at a higher cost and further raises the value of Y tokens. Sam then sells some of his Y tokens to get back his ϵx tokens of type X from his first transaction. Because Y tokens are more valuable, he doesn't have to sell as many to get back his original number of X tokens. This means that Sam now has his

original number of X tokens plus some leftover number of Y tokens (where he makes his profit).

Part C

C) Does Sam's profit increase or decrease with the amount he spends in Tx_1? In other words, is a larger ϵ better or worse for Sam (assuming he stays below Alice's max exchange rate)?

A larger ϵ is better for Sam because this gives him more room to extract value from Alice's transaction; i.e., if Alice allows a higher slippage / maximum exchange rate in her transaction, then Sam can spend more in his Tx_1 to capture more of the value that Alice is willing to give up.

Part D

D) The validator Victor who is proposing the block sees both Sam's transactions. Victor realizes that he can issue both transactions itself and keep all the profit to itself. However, Victor has no X tokens, as needed for Tx_1. Can Victor mount the sandwitch attack without Sam?

Yes, Victor can take out a flash loan to get X tokens and perform a sandwich attack on Alice.

Part E

E) Can Alice use MEV-boost to protect herself from these shenanigans by Sam and Victor?

Yes — Alice could build a block including her transactions (and no sandwich transaction) and include an MEV offer to a validator. If this offer is high enough, a relay would choose her block over others (without exposing her transaction) and send the block header and MEV offer to a proposer. The proposer would then sign the block and send it to the network.

Neither Sam nor Victor would be able to see that her proposed transaction until it was accepted by a proposer and sent to the network (at which point it is too late to perform a sandwich attack).

3.2 B **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect or blank

original number of X tokens plus some leftover number of Y tokens (where he makes his profit).

Part C

C) Does Sam's profit increase or decrease with the amount he spends in Tx_1? In other words, is a larger ϵ better or worse for Sam (assuming he stays below Alice's max exchange rate)?

A larger ϵ is better for Sam because this gives him more room to extract value from Alice's transaction; i.e., if Alice allows a higher slippage / maximum exchange rate in her transaction, then Sam can spend more in his Tx_1 to capture more of the value that Alice is willing to give up.

Part D

D) The validator Victor who is proposing the block sees both Sam's transactions. Victor realizes that he can issue both transactions itself and keep all the profit to itself. However, Victor has no X tokens, as needed for Tx_1. Can Victor mount the sandwitch attack without Sam?

Yes, Victor can take out a flash loan to get X tokens and perform a sandwich attack on Alice.

Part E

E) Can Alice use MEV-boost to protect herself from these shenanigans by Sam and Victor?

Yes — Alice could build a block including her transactions (and no sandwich transaction) and include an MEV offer to a validator. If this offer is high enough, a relay would choose her block over others (without exposing her transaction) and send the block header and MEV offer to a proposer. The proposer would then sign the block and send it to the network.

Neither Sam nor Victor would be able to see that her proposed transaction until it was accepted by a proposer and sent to the network (at which point it is too late to perform a sandwich attack).

3.3 C **2 / 2**

✓ **- 0 pts** *Correct*

- 2 pts Incorrect/blank

original number of X tokens plus some leftover number of Y tokens (where he makes his profit).

Part C

C) Does Sam's profit increase or decrease with the amount he spends in Tx_1? In other words, is a larger ϵ better or worse for Sam (assuming he stays below Alice's max exchange rate)?

A larger ϵ is better for Sam because this gives him more room to extract value from Alice's transaction; i.e., if Alice allows a higher slippage / maximum exchange rate in her transaction, then Sam can spend more in his Tx_1 to capture more of the value that Alice is willing to give up.

Part D

D) The validator Victor who is proposing the block sees both Sam's transactions. Victor realizes that he can issue both transactions itself and keep all the profit to itself. However, Victor has no X tokens, as needed for Tx_1. Can Victor mount the sandwitch attack without Sam?

Yes, Victor can take out a flash loan to get X tokens and perform a sandwich attack on Alice.

Part E

E) Can Alice use MEV-boost to protect herself from these shenanigans by Sam and Victor?

Yes — Alice could build a block including her transactions (and no sandwich transaction) and include an MEV offer to a validator. If this offer is high enough, a relay would choose her block over others (without exposing her transaction) and send the block header and MEV offer to a proposer. The proposer would then sign the block and send it to the network.

Neither Sam nor Victor would be able to see that her proposed transaction until it was accepted by a proposer and sent to the network (at which point it is too late to perform a sandwich attack).

3.4 D **2 / 2**

✓ **- 0 pts** *Correct/Demonstrates Understanding*

- 1 pts Attack Described is not a sandwich attack and/or still requires Sam

- 2 pts Incorrect/Blank

original number of X tokens plus some leftover number of Y tokens (where he makes his profit).

Part C

C) Does Sam's profit increase or decrease with the amount he spends in Tx_1? In other words, is a larger ϵ better or worse for Sam (assuming he stays below Alice's max exchange rate)?

A larger ϵ is better for Sam because this gives him more room to extract value from Alice's transaction; i.e., if Alice allows a higher slippage / maximum exchange rate in her transaction, then Sam can spend more in his Tx_1 to capture more of the value that Alice is willing to give up.

Part D

D) The validator Victor who is proposing the block sees both Sam's transactions. Victor realizes that he can issue both transactions itself and keep all the profit to itself. However, Victor has no X tokens, as needed for Tx_1. Can Victor mount the sandwitch attack without Sam?

Yes, Victor can take out a flash loan to get X tokens and perform a sandwich attack on Alice.

Part E

E) Can Alice use MEV-boost to protect herself from these shenanigans by Sam and Victor?

Yes — Alice could build a block including her transactions (and no sandwich transaction) and include an MEV offer to a validator. If this offer is high enough, a relay would choose her block over others (without exposing her transaction) and send the block header and MEV offer to a proposer. The proposer would then sign the block and send it to the network.

Neither Sam nor Victor would be able to see that her proposed transaction until it was accepted by a proposer and sent to the network (at which point it is too late to perform a sandwich attack).

3.5 E **3 / 3**

✓ **- 0 pts** *Correct*

- 3 pts Blank\Incorrect

Part A

A) Alice loves TokenId 10 in the Zebra collection and places a bid of 100 USDC for this NFT. That is, Alice posts a transaction that calls

`createBid(Zebra, 10, USDC, 100)`

The seller of TokenId 10 is willing to sell the NFT at this price, and calls

`acceptBid(alice, Zebra, 10, USDC)`

Show that Alice can now obtain the NFT by paying only 1 USDC for it (not 100 USDC).

Alice can keep an eye out for the seller's acceptance of the bid; when Alice sees that the seller has submitted a transaction accepting the bid, Alice can front-run the seller and post another bid for the same token, this time for only 1 USDC.

This will update the `price` in `bids[Alice's addr][key][10]` and will cause the seller to transfer the token for only 1 USDC instead of the original 100 that were offered.

To fix the problem from part (A) the implementation of `acceptBid()` is changed to the following: (all the changes are underlined)

```
function acceptBid(address bidder, ERC721 erc721Token, uint erc721TokenId,
                   ERC20 erc20Token, uint256 price) external {
  uint160 key = _getKey(erc20Token, erc721Token);
  uint256 bidPrice = bids[bidder][key][erc721TokenId]; // get bid price
  require(bidPrice != 0, "no bid");  // ensure bid exists
  require(price <= bidPrice, ‘‘bid too low’’);
  delete bids[bidder][key][erc721TokenId];
  // do the exchange:  ERC721 TokenId for the ERC20 payment
  require(erc20Token.transferFrom(bidder, msg.sender, price));
  require(erc721Token.transferFrom(msg.sender, bidder, erc721TokenId));
}
```

This lets the seller specify the sale price and ensures that the item is not sold below its asking price.

Alice is happy with the modified contract and uses it to buy and sell several NFTs. To do so she calls the `setApprovalForAll` function of the Zebra ERC-721 to indicate that the `InsecureMarket` contract is authorized to sell NFTs on her behalf.

4.1 A **7 / 7**

✓ **- 0 pts** *Correct*

- 7 pts Incorret/No submission

- 1 pts Minor error: Insufficient explanation- Alice's second createBid call can frontrun acceptBid with a higher maxPriorityFee. For the attack to succeed, acceptBid should be posted before createBid.

- 2 pts The seller has already called acceptBid. Alice can call createBid even after acceptBid has been called, by frontrunning, i.e. with a higher maxPriorityFee so that the new createBid is executed first

- 3 pts Technically, Bob can know if Alice modified the mapping value to 1 if Alice's createBid is executed on-chain. Alternatively, Alice can check the mempool to see if createBid has been called and and then call createBid to frontrun the transaction with a higher maxPriorityFee

- 5 pts Alice can call createBid once again after acceptBid has been added to the mempool. By frontrunning, i.e. with a higher maxPriorityFee so that the new createBid is executed first

- 3 pts Click here to replace this description.

- 1 pts Click here to replace this description.

- 2 pts Did not call createBid again to change the mapping

- 3 pts Click here to replace this description.

- 4 pts Click here to replace this description.

Part B

B) There is still another attack on this contract. Suppose Alice recently bought TokenId 53 in the Zebra collection and wants to keep it. She also likes TokenId 34 and places a bid of 80 USDC for it by calling

`createBid(Zebra, 34, USDC, 80)`

Show that Bob can now call the `acceptBid()` function and steal Alice's TokenId 53 by paying only 34 USDC for it.
Hint: use the fact that $a \operatorname{xor} b = b \operatorname{xor} a$.

- Alice places a bid on TokenId 34 by calling `createBid(Zebra, 34, USDC, 80)`
- The bid is recorded: `bids[Alice's addr][Zebra addr ^ USDC addr][34] = 80`
- Bob calls `acceptBid(Alice's addr, USDC, 34, Zebra, 53)`
  - this results in the `key` being the same as above because `Zebra addr ^ USDC addr = USDC addr ^ Zebra ADDR`.
  - The `bidPrice` is then retrieved from `bids[Alice's addr][key][34] = 80`
  - The `price` is less than `bidPrice`
  - the `require(erc20Token.transferFrom()` line transfers TokenId 53 from Alice to Bob
  - the `require(erc721Token.transferFrom()` line transfers 34 USDC from Bob to Alice

Part C

C) Explain how to fix the implementation by changing the `_getKey()` function.

An easy way to fix this would be to hash the two token addresses — because hashes are collision-resistant, $H(\text{Zebra addr}, \text{USDC addr})$ will be very different than $H(\text{USDC addr}, \text{Zebra addr})$. The updated function could be:

```
function _getKey(ERC20 erc20Token, ERC721 erc721Token) private pure returns (uint160 key) {
  uint160 hash = uint160(keccak256(erc20Token, erc721Token));
  return hash;
```

4.2 B **6 / 7**

- **0 pts** Correct
- **7 pts** Incorrect/Incomplete
- **2 pts** bidder is not Alice
- **1 pts** erc721 token id is not 34

✓ **- 1 pts** *price is not 53*

- **2 pts** Did not swap order of the token names
- **4 pts** Invalid Params/Did not specify acceptBid call
- **1 pts** minor error
- **2 pts** Click here to replace this description.
- **5 pts** Click here to replace this description.

Part B

B) There is still another attack on this contract. Suppose Alice recently bought TokenId 53 in the Zebra collection and wants to keep it. She also likes TokenId 34 and places a bid of 80 USDC for it by calling

`createBid(Zebra, 34, USDC, 80)`

Show that Bob can now call the `acceptBid()` function and steal Alice's TokenId 53 by paying only 34 USDC for it.
Hint: use the fact that $a \operatorname{xor} b = b \operatorname{xor} a$.

- Alice places a bid on TokenId 34 by calling `createBid(Zebra, 34, USDC, 80)`
- The bid is recorded: `bids[Alice's addr][Zebra addr ^ USDC addr][34] = 80`
- Bob calls `acceptBid(Alice's addr, USDC, 34, Zebra, 53)`
  - this results in the `key` being the same as above because `Zebra addr ^ USDC addr = USDC addr ^ Zebra ADDR`.
  - The `bidPrice` is then retrieved from `bids[Alice's addr][key][34] = 80`
  - The `price` is less than `bidPrice`
  - the `require(erc20Token.transferFrom()` line transfers TokenId 53 from Alice to Bob
  - the `require(erc721Token.transferFrom()` line transfers 34 USDC from Bob to Alice

Part C

C) Explain how to fix the implementation by changing the `_getKey()` function.

An easy way to fix this would be to hash the two token addresses — because hashes are collision-resistant, $H(\text{Zebra addr}, \text{USDC addr})$ will be very different than $H(\text{USDC addr}, \text{Zebra addr})$. The updated function could be:

```
function _getKey(ERC20 erc20Token, ERC721 erc721Token) private pure returns (uint160 key) {
  uint160 hash = uint160(keccak256(erc20Token, erc721Token));
  return hash;
```

4.3 C **6 / 6**

✓ **- 0 pts** *Correct*

- 2 pts Did not specify an alternative implementation

- 4 pts Error

- 6 pts Incorrect/ No submission

- 2 pts Click here to replace this description.

Problem 5

A proof of solvency lets an exchange prove that it has enough assets to cover all of its obligations to its clients. For simplicity, suppose that the exchange is willing to reveal its on-chain assets (as some exchanges have recently done). This leaves the question of how to provably reveal its total obligation to its clients, but do it privately without revealing every client's balance. We will do so using a zk-SNARK.

To simplify, suppose the exchange only accepts deposits in Ether. Let $bal_i \in \{0, \ldots, 2^{20}\}$ be the balance in Ethers of client number i. The exchange builds a Merkle tree where leaf number i is the balance bal_i of client number i, for $i = 1, \ldots, n$. For a technical reason, leaf number 0 is set to a random 256-bit number. Let us assume that the total number of leaves is a power of two. Let R be the resulting Merkle root, and let $B := \sum_{i=1}^{n} bal_i$ be the sum of all the balances.

In addition, the exchange builds a zk-SNARK proof π that B is equal to the sum of all the leaves in the tree. More precisely, the zk-SNARK statement is defined by:

- the public statement is (T, B),
- the secret witness is $\{(bal_i, \Pi_i)\}_{i=1}^{n}$, and
- the arithmetic circuit C outputs 0 if and only if (i) $\sum_{i=1}^{n} bal_i = B$, and (ii) Π_i is a valid Merkle proof for bal_i for all $i = 1, \ldots, n$.

The zk-SNARK proof π proves that the circuit C outputs 0 given the public statement and secret witness as input.

The exchange publishes the triple (T, B, π). Anyone can verify that π is a valid proof, and that B is smaller than the total assets that the exchange owns. In addition, the exchange uses a secure chat to send to client i its balance bal_i and its Merkle proof Π_i. Every client can verify that its balance is correct and that the Merkle proof is correct with respect to the published root R. This convinces the client that its correct balance was included in the calculation of the sum B. If no client complains, then the public can conclude that the exchange is solvent: its assets are greater than its obligations.

There are a number of vulnerabilities in this simplistic construction.

Part A

A) Suppose Alice and Bob have the same balance. Then a malicious exchange can use the same leaf for both Alice and Bob. Explain why this would cause the total obligation B to be lower than its true value, and yet no client will complain that the proof is invalid.

Say that both Bob and Alice have a balance of 1000 ETH. The exchange can conduct a secure chat with each of Bob and Alice, showing them their balances and the proof for each of their balances.

Because their balances are the same, the exchange can show Alice and Bob the same leaf of the tree and the same proof for that leaf. Neither Bob nor Alice will complain, but now, supposing that the exchange showed them Alice's leaf and proof, the exchange can effectively use Bob's balance (which is not included as a leaf in the tree or the in secret witness). In the end, neither Alice nor Bob will complain, but the total obligation B could be 1000 ETH less than the actual obligation.

Part B

B) How would you enhance the proof of solvency to mitigate the attack from part (A)?

asdf

Part C

C) Suppose client Eve is collaborating with the exchange (and both are malicious). Show that by working together they can cause the total obligation B to be much lower than its true value, say half its true value, and yet no client will complain that the proof is invalid.

Eve can have a large negative balance — this will reduce the total obligation B and Eve won't complain because she is also malicious.

Part D

D) How would you enhance the proof of solvency to mitigate the attack from part (C)?

The proof could be modified in order to prove that the balance in each leaf is non-negative, i.e., the circuit C would have an additional constraint in that it only outputs 0 if $bal_i \neq 0$ for all i.

Problem 6

5.1 A **4 / 4**

✓ **- 0 pts** *Correct*

- 1 pts Doesn't discuss how this leads to lower B than its true value.

- 2 pts Partial Credit

- 4 pts Nothing/Incorrect

Because their balances are the same, the exchange can show Alice and Bob the same leaf of the tree and the same proof for that leaf. Neither Bob nor Alice will complain, but now, supposing that the exchange showed them Alice's leaf and proof, the exchange can effectively use Bob's balance (which is not included as a leaf in the tree or the in secret witness). In the end, neither Alice nor Bob will complain, but the total obligation B could be 1000 ETH less than the actual obligation.

Part B

B) How would you enhance the proof of solvency to mitigate the attack from part (A)?

asdf

Part C

C) Suppose client Eve is collaborating with the exchange (and both are malicious). Show that by working together they can cause the total obligation B to be much lower than its true value, say half its true value, and yet no client will complain that the proof is invalid.

Eve can have a large negative balance — this will reduce the total obligation B and Eve won't complain because she is also malicious.

Part D

D) How would you enhance the proof of solvency to mitigate the attack from part (C)?

The proof could be modified in order to prove that the balance in each leaf is non-negative, i.e., the circuit C would have an additional constraint in that it only outputs 0 if $bal_i \neq 0$ for all i.

Problem 6

5.2 B **0 / 4**

- 0 pts Correct. Links user identifier with their balance (e.g. concatenates them in the leaf).

- 2 pts Good try. Still doesn't link balances to users.

- 2 pts Good try, but still doesn't prevent the attack. Still doesn't link balances to users.

- 2 pts Good try, but assumes that users can communicate, or have to interact, or know how many of them exist, or do anything past verifying their own proofs. Still doesn't link balances to users.

- 2 pts Good try, but assumes users know how many of them exist. Still doesn't link balances to users.

- 2 pts Doesn't specify how to use the added user identifier within the proof. Still doesn't link balances to users.

- 3 pts Answer too vague/high-level. Still doesn't link balances to users.

✓ **- 4 pts** *Incorrect/Incomplete. Still doesn't link balances to users. Still doesn't prevent the attack.*

Because their balances are the same, the exchange can show Alice and Bob the same leaf of the tree and the same proof for that leaf. Neither Bob nor Alice will complain, but now, supposing that the exchange showed them Alice's leaf and proof, the exchange can effectively use Bob's balance (which is not included as a leaf in the tree or the in secret witness). In the end, neither Alice nor Bob will complain, but the total obligation B could be 1000 ETH less than the actual obligation.

Part B

B) How would you enhance the proof of solvency to mitigate the attack from part (A)?

asdf

Part C

C) Suppose client Eve is collaborating with the exchange (and both are malicious). Show that by working together they can cause the total obligation B to be much lower than its true value, say half its true value, and yet no client will complain that the proof is invalid.

Eve can have a large negative balance — this will reduce the total obligation B and Eve won't complain because she is also malicious.

Part D

D) How would you enhance the proof of solvency to mitigate the attack from part (C)?

The proof could be modified in order to prove that the balance in each leaf is non-negative, i.e., the circuit C would have an additional constraint in that it only outputs 0 if $bal_i \neq 0$ for all i.

Problem 6

5.3 C **6 / 6**

✓ **- 0 pts** *Correct*

- 2 pts Minor Error

- 4 pts Major Error

- 6 pts Blank / Not Attempted

Because their balances are the same, the exchange can show Alice and Bob the same leaf of the tree and the same proof for that leaf. Neither Bob nor Alice will complain, but now, supposing that the exchange showed them Alice's leaf and proof, the exchange can effectively use Bob's balance (which is not included as a leaf in the tree or the in secret witness). In the end, neither Alice nor Bob will complain, but the total obligation B could be 1000 ETH less than the actual obligation.

Part B

B) How would you enhance the proof of solvency to mitigate the attack from part (A)?

asdf

Part C

C) Suppose client Eve is collaborating with the exchange (and both are malicious). Show that by working together they can cause the total obligation B to be much lower than its true value, say half its true value, and yet no client will complain that the proof is invalid.

Eve can have a large negative balance — this will reduce the total obligation B and Eve won't complain because she is also malicious.

Part D

D) How would you enhance the proof of solvency to mitigate the attack from part (C)?

The proof could be modified in order to prove that the balance in each leaf is non-negative, i.e., the circuit C would have an additional constraint in that it only outputs 0 if $bal_i \neq 0$ for all i.

Problem 6

5.4 D **6 / 6**

✓ **- 0 pts** *Correct*

- 0 pts Correct given 5c

- 2 pts Minor Error

- 4 pts Major Error

- 6 pts Blank / Not Attempted

A new contract can be created in the EVM using two opcodes `CREATE` and `CREATE2`.

- When a new contract is created with `CREATE` the address assigned to the new contract is computed as `keccak256(sender, nonce)`, where `sender` is the address of the contract that executed `CREATE`, and `nonce` is the sender's current nonce value. Recall that the nonce of a contract is initialized to zero when it is first created, and is incremented by one whenever it creates a new contract.
- When a new contract is created with `CREATE2` the address of the new contract is computed as `keccak256(0xFF, sender, salt, bytecode)`, where `0xFF` is a fixed string, `sender` is as before, `salt` is supplied by the sender, and `bytecode` is the hash of the initialization code of the new contract, as supplied by the sender.

Recall that `keccak256` is a hash function that is believed to be collision resistant. The sender provides both `CREATE` and `CREATE2` with the initialization code and the main body code for the new contract.

Part A

A) Suppose `CREATE2` did not exist (i.e., there was no such opcode). Can two different contracts created with `CREATE` collide and end up at the same address?

Theoretically, yes, but not by the same user. Assuming that $keccak256$ is collision resistant and user A created contract X and then contract Y, the two contracts would have different addresses because $keccak256(addr_A + nonce_1) \neq keccak256(addr_A + nonce_2)$ and $nonce_1 \neq nonce_2$ for sequential calls of $CREATE$.

If two different users had similar addresses, they could create contracts with the same address. Say user A has address 0x1234 and has never created another contract (its nonce is 0), and user B has address 0x1233 and has only created one other contracted (its nonce is 1), then their respective hashes will end up being the same because the inputs are the same (i.e., $hash(0x1234 + 0) = hash(0x1233 + 1)$.

This would be very difficult to do in practice due to the low probability of two users having numerically close address values.

Part B

6.1 A **3 / 3**

✓ **- 0 pts** *Correct*

- 2 pts Incorrect (duplicate addresses would imply a collision on keccak256)

- 3 pts Empty

A new contract can be created in the EVM using two opcodes `CREATE` and `CREATE2`.

- When a new contract is created with `CREATE` the address assigned to the new contract is computed as `keccak256(sender, nonce)`, where `sender` is the address of the contract that executed `CREATE`, and `nonce` is the sender's current nonce value. Recall that the nonce of a contract is initialized to zero when it is first created, and is incremented by one whenever it creates a new contract.
- When a new contract is created with `CREATE2` the address of the new contract is computed as `keccak256(0xFF, sender, salt, bytecode)`, where `0xFF` is a fixed string, `sender` is as before, `salt` is supplied by the sender, and `bytecode` is the hash of the initialization code of the new contract, as supplied by the sender.

Recall that `keccak256` is a hash function that is believed to be collision resistant. The sender provides both `CREATE` and `CREATE2` with the initialization code and the main body code for the new contract.

Part A

A) Suppose `CREATE2` did not exist (i.e., there was no such opcode). Can two different contracts created with `CREATE` collide and end up at the same address?

Theoretically, yes, but not by the same user. Assuming that $keccak256$ is collision resistant and user A created contract X and then contract Y, the two contracts would have different addresses because $keccak256(addr_A + nonce_1) \neq keccak256(addr_A + nonce_2)$ and $nonce_1 \neq nonce_2$ for sequential calls of $CREATE$.

If two different users had similar addresses, they could create contracts with the same address. Say user A has address 0x1234 and has never created another contract (its nonce is 0), and user B has address 0x1233 and has only created one other contracted (its nonce is 1), then their respective hashes will end up being the same because the inputs are the same (i.e., $hash(0x1234 + 0) = hash(0x1233 + 1)$.

This would be very difficult to do in practice due to the low probability of two users having numerically close address values.

Part B

B) Bob regularly interacts with a contract at address X. He inspected its code and trusts it. Suppose that X was created using `CREATE2` by a contract at address Y. Describe a sequence of events that can cause the code running at address X to be replaced by some new code, without Bob realizing it.
Note that `CREATE2` will fail if it tries to create a new contract at an address already occupied by an existing contract. However, if the existing contract has previously called `SELFDESTRUST` (so that the address is currently vacant) then `CREATE2` will succeed. You may assume that at time T the current contract at address X self destructs.

If user A created the contract at address X, they know the `salt` and `bytecode` that were used to create the contract address. User A could trigger this contract to self-destruct and then write another contract with $CREATE2$ using the same constructor function (meaning it produces the same `bytecode`) and salt that were used during the initial creation.

This would put another (potentially malicious) contract at the same address X and users of this contract would not be aware without inspecting the code there.

Part C

C) Let's show that your answer from part (B) can also apply to a contract created with `CREATE`, despite your answer from part (A). Suppose a contract at address A uses `CREATE2` to create a contract at address B. The contract at address B uses `CREATE` to create a contract at address C. Describe a sequence of events that could cause the code body of the contract at address C to be replaced by entirely new code. You may assume any self destruct activity that you need.

The contract at address B can self-destruct at some time T. The contract at address A can then use $CREATE2$ to create another contract at the same address B because it knows the `salt` and `bytecode` (which is determined by the contract's `constructor` function) that were used in the original.

This new contract at address B will have its nonce reset to 0, meaning it can call $CREATE$ to update the contract at address C (assuming that the contract at address C self-destructs first).

6.2 B **3 / 3**

✓ **- 0 pts** *Correct*

- 1 pts Incorrectly states that the contract at Y must also self-destruct

- 1 pts Insufficient explanation

- 3 pts Empty

- 1 pts Does not mention that the contract at X must self-destruct

B) Bob regularly interacts with a contract at address X. He inspected its code and trusts it. Suppose that X was created using `CREATE2` by a contract at address Y. Describe a sequence of events that can cause the code running at address X to be replaced by some new code, without Bob realizing it.
Note that `CREATE2` will fail if it tries to create a new contract at an address already occupied by an existing contract. However, if the existing contract has previously called `SELFDESTRUST` (so that the address is currently vacant) then `CREATE2` will succeed. You may assume that at time T the current contract at address X self destructs.

If user A created the contract at address X, they know the `salt` and `bytecode` that were used to create the contract address. User A could trigger this contract to self-destruct and then write another contract with $CREATE2$ using the same constructor function (meaning it produces the same `bytecode`) and salt that were used during the initial creation.

This would put another (potentially malicious) contract at the same address X and users of this contract would not be aware without inspecting the code there.

Part C

C) Let's show that your answer from part (B) can also apply to a contract created with `CREATE`, despite your answer from part (A). Suppose a contract at address A uses `CREATE2` to create a contract at address B. The contract at address B uses `CREATE` to create a contract at address C. Describe a sequence of events that could cause the code body of the contract at address C to be replaced by entirely new code. You may assume any self destruct activity that you need.

The contract at address B can self-destruct at some time T. The contract at address A can then use $CREATE2$ to create another contract at the same address B because it knows the `salt` and `bytecode` (which is determined by the contract's `constructor` function) that were used in the original.

This new contract at address B will have its nonce reset to 0, meaning it can call $CREATE$ to update the contract at address C (assuming that the contract at address C self-destructs first).

6.3 C **4 / 4**

✓ **- 0 pts** *Correct*

- 1 pts Does not mention that C also must self-destruct

- 1 pts Does not mention that B also must self-destruct

- 4 pts Empty

💯

CS251 - 2022 Final - George Davis

Date: @December 15, 2022

Problem 4 - Solidity Bugs

Consider the following (abbreviated) implementation of an NFT marketplace. Anyone can call `createBid()` to create a bid for an NFT and offer payment in an ERC-20 token. The NFT is represented by the address of an ERC-721 contract and a TokenId. Recall that an ERC-721 contract can hold many NFTs, and the tokenId identifies the specific NFT in the collection.

Once the bid is recorded, the NFT owner can call `acceptBid()` to accept payment from the bidder and transfer the NFT to the bidder.

```
contract InsecureMarket {

  // a mapping holding all active bids
  mapping(address => mapping(uint160 => mapping(uint256 => uint256))) bids;

  function createBid(ERC721 erc721Token, uint256 erc721TokenId,
                     ERC20 erc20Token, uint256 price) external
    {
        // pack the two token addresses into a single field
        uint160 key = _getKey(erc20Token, erc721Token);
        // record the bid
        bids[msg.sender][key][erc721TokenId] = price;
    }


  function acceptBid(address bidder, ERC721 erc721Token,
                     uint erc721TokenId, ERC20 erc20Token) external
    {
        uint160 key = _getKey(erc20Token, erc721Token);
        uint256 price = bids[bidder][key][erc721TokenId]; // get bid price
        require(price != 0, "no bid");  // ensure bid exists
        delete bids[bidder][key][erc721TokenId];
        // do the exchange:  ERC721 TokenId for the ERC20 payment
        require(erc20Token.transferFrom(bidder, msg.sender, price));
        require(erc721Token.transferFrom(msg.sender, bidder, erc721TokenId));
    }

  function  _getKey(ERC20 erc20Token, ERC721 erc721Token)
              private pure returns (uint160 key)
    {
        // pack the two addresses into one by computing their XOR
        return uint160(address(erc20Token)) ^ uint160(address(erc721Token));
    }
```

While this marketplace appears to work correctly, it contains several devastating vulnerabilities. Consider the fictitious Zebra collection, an ERC-721 contract.