Romina Salveraglio

A) In Ethereum and blockchain ecosystem in general, what are the use-cases of hash functions?

Hash functions are used in the proof of work consensus. Miners need to achieve that after applying a hash function (sha256) to certain block header values (version, previous block hash, merkle root hash, timestamp and bits) and a nonce (the value that miner keeps varying until it solves the challenge), it results in a value minor that the defined target (bits).

Hash functions are also used to prove authenticity of data with digital fingerprints. Hashes have no place for collisions, which means that it is impossible for two different data to generate the same hash. Furthermore, they assure no bidirectionality, which means that it should not be possible to obtain the original data given its digital fingerprint. Public/private key generation (for encryption and digital signatures) also make use of hash functions. Once private keys are generated, the process of obtaining its corresponding public key uses hash functions and finally, the generation of addresses also involves hash functions. Ethereum uses keccak-256 when obtaining addresses, while Bitcoin uses sha-256.

Furthermore, every transaction added to the blockchain has an id (txid/transaction hash) to identify it and, for example, be able to find it on a block explorer. This txid is the hash of the transaction details (nonce, data, target account, sender,value).

Moreover, hashes are used in the process of constructing a block. Each block header has the hash of the previous block header in order to ensure that changing a block implies changing all subsequent ones. This way it increases the difficulty of corrupting the blockchain.
Also, each block summarizes all its transactions by constructing a merkle tree from them and saving the root hash on its header. As this procedure uses hash functions, it ensures that there is no way of having two same merkle tree root hashes. It is another way of protecting the integrity of the blockchain as it is not possible to corrupt any of the contained transactions without needing to modify all subsequent blocks.

B) What are the bottlenecks in terms of performance when operating on a network like Ethereum? What kind of solutions can be utilized to overcome them?

Main bottlenecks on a network like ethereum are its poor throughput time and the cost of its transactions (no matter how basic they are).
Ethereum processes, at its full capacity, 20 transactions per second. Even Though these transactions have the perk of not requiring third parties, centralized entities, like Visa for example, have the capacity of processing up to 24000 tx per second.
One possible solution would be increasing block size, making it possible to fit more transactions per block interval. Nonetheless, this would increase the computational requirements nodes must meet in order to participate in the network and could lead to fewer full nodes (making it a more centralized network).
Another solution is using some second layer to process transactions before needing to use the core blockchain. An example of this second layer could be an off-chain payment network, which consists in opening payment channels between parties that are willing to be constantly interacting (like a bar and

a customer) and only use the main blockchain when this channel is closed or batching multiple payments into one transaction. ZK-rollups are another second layer example which, by bundling transactions into batches to be executed off-chain, reduces the amount of data published to the blockchain and only submits summaries of all the changes to represent all the transactions in every batch. These proposed changes are accompanied by a proof of their validity in order to ensure that they truly correspond to the result of executing all transactions in the batch.

ETH 2.0 aims to tackle this performance problem. Even Though the incoming merge will not increase the amount of transactions per second (nor reduce the cost of gas), several proposals to improve transaction throughput have been published in the roadmap towards eth2.0. Sharding is one of the improvements which will lead to an increment on the transaction speed and thus, transaction costs will also reduce. It consists in partitioning (sharding) blockchain's data into smaller manageable pieces. This will significantly decrease the workload of validators as instead of managing the whole blockchain, they will only need to store and manage certain shards, enabling parallel processing (which increases transaction speed). This means that not all transactions, nor all the smart contracts, will be executed on all the network. Invoking smart contracts from within a same shard, will be more efficient than invoking one form an external shard. Thus, sharding could lead to certain centralization as the shards with the more popular contracts will tend to concentrate a greater amount of transactions.

C) What are the different types of bridges and how do they work ? Explain in detail step by step how bridging an ERC721 would work.

Bridges are the way of moving assets from one blockchain to another. They can be unidirectional, allowing only to port assets to the target blockchain (for example an unidirectional bridge from bitcoin to Ethereum, would let users send BTC, by wrapping them into an ERC-20 token, to ethereum but would not let users send ETH to BTC) or bidirectional (allowing to send assets in both ways).
Another classification is that they can be custodial(centralized) or noncustodial (decentralized) and the difference relies on who controls the tokens used to wrap bridge assets.

Bridging an ERC721 could work on the following way:

- An ERC721 contract is deployed in one blockchain (origin) and an NFT is minted on it.
- NFT owner, or some account with approval to transfer the token in matter, needs to move it from origin to another blockchain (target)
- There is a bridge contract deployed on both blockchains (origin and target)
- safeTransferFrom is called on ERC721 contract on home having bridge contract address as target (it previously verifies that bridge can actually receive ERC721 tokens)
- All necessary info of the token (token uri (metadata) and current owner) is obtained by the bridge contract
- By emitting events, and with a server that listens to them and communicate with bridges on both sides, it gets to make bridge on target network invoke minting on ERC721 contract on target network with the corresponding metadata and with original owner (owner of the NFT at home blockchain) as recipient.

The above mentioned bridge corresponds to one of the more centralized patterns, which is an aspect that it is not the more recommended one.

D) Describe the EIP-2771 standard with your own words and describe some use cases using this EIP.

This standard proposes an interface that contracts need to implement in order to be able to, through a trusted forwarder, receive meta transactions.
Meta transactions are what makes it possible to have one account as the one signing the transaction and another one as the one paying for its execution. This enables gasless transactions for users, which means that contracts could accept calls from EOA that do not have funds for paying gas.

Meta transactions recipient contracts rely on a trusted forwarder contract that must verify signature and nonce of original sender and append original sender address to calldata. Recipient contract must check that it trusts the forwarder before extracting the original sender address from calldata. If msg.sender is not a trusted forwarder or if msg.data is shorter than 20 bytes, it returns the original sender as it is. On the recipient contract, msg.sender must be replaced by _msgSender() wherever it is used.
In order to enable clients to know if some contract supports meta-transactions, a recipient contract must implement a function to retrieve whether an address is or is not a trusted forwarder.

Meta-transactions are a way of improving usability as it lets users interact with the dapp without having to engage with the blockchain from the start. It makes even more sense when a dappp enables payments with ERC20 tokens instead of native currency as it could let users pay invocations with the tokens generated on the dapp platform itself.
In order to achieve sending/receiving gasless transactions, changes must be done on the client side and also on the SC in order to make it meta-transaction compatible.

E) What are the centralization issues to be aware of when developing a smart contract ? Propose a technical solution for each of them.

Centralization issues when developing a smart contract are all those vulnerabilities that can lead to attacks by malicious developers of the project or by outsiders.
Having centralized ownership instead of a multisig key ownership is a centralization issue that could end with a malicious developer, for example, withdrawing all contract funds, or if it was a token contract and the owner also has minter role, it could mint as many tokens as he want and send them to any address of its interest.

Private key mismanagement is another centralization issue, which gains importance if there is no multisig wallet in the contract. For example, if some attacker ends up with contract owner private keys and there is no time lock wallet or multisig, it could lead to the attacker draining all contracts funds (for example, if after launching users kept balance inside contracts, founders could steal it and “disappear”. This could only be possible if a withdrawal method is implemented, otherwise funds would be kept stuck in the contract).

Romina Salveraglio

Not having access control on smart contracts and having only one central role is another centalization issue. If one account has the ability of for example minting tokens, withdrawing funds, granting certain privileges to accounts, etc, if the private key of that account ends in the wrong hands, or if the owner is malicious by itself, it could compromise everything on the smart contract. Nonetheless, despite having only one privilege role is worse, having multiple privilege roles could also be seen as a centralization issue.

In order to mitigate these risks, multisig should be implemented in order to withdraw funds from the contracts, or a time lock wallet to mitigate the risk of the contract getting drained. Moreover, a DAO model on certain methods could be implemented in order to prevent some team members, or some attackers with access to certain private keys, from compromising the contract.

Furthermore, when making use of oracles on smart contracts, it must be taken into consideration that if there is only one oracle as source of information, it would represent a centralization issue as certain functionalities will be relaying on the information provided from only one source which could be corrupted, shut down, etc; it would mean having a single point of failure. When using oracles it is important to have them decentralized and implement some sort of control to require a minimum of X out of Z (total number of oracles) equal answers in order to do something with the smart contract (a consensus mechanism).

Finally, using proxies could also represent a centralization issue. Suppose the smart contract provides a method to modify the address which defines the implementation being used, then if an attacker gets to execute this method successfully, it could define any contract he wants as the one being called by the proxy.

F) What process (step by step) do you follow to build a protocol (a set of smart contracts) from given specifications ?

Given the specifications (assuming this includes the blockchain in which it is going to be deployed, which would determine the programming language to be used), i would follow this process:

- Understand each specification in detail. Take notes on any point that I considered it should be discussed (to avoid ambiguity, border cases or anything I find that should be taken into consideration that it is not being mentioned)
- Identify the smart contracts that needs to be build
- Start designing methods based on each specification. Think of which conditions must be met in each case in order to enable the method execution. Identify which methods must have access control. Identify if there should be different roles to control access on certain methods and define them. Identify which variable states must be defined and which events would be useful.
- Define if some set of contracts, of openzeppelin for example, are going to be used as base contracts.
- Start coding one of the smart contracts
- Once I consider that I have reached a first version, implement its corresponding test file and test every method (tests on success cases and failure cases). If there is any problem when trying to compile the contract, it will come out here. Every functionality should be tested and every error found while testing should be modified and tested again until it satisfies contract specification associated with the method in question. This is an iterative process.

- Once I consider that I can move to the next smart contract, I repeat the process of coding, testing and I iterate on this process until it is finished.
- Check if there is repeated code between contracts and evaluate if it should be move to a library
- All the first tests while developing are done on some local testnet network, like the ones provided by Hardhat or Truffle for example. Then contracts should be deployed to public testnet networks and tested again before trying to deploy them to some mainnet.

G) After analyzing the file "Signature.sol", describe the use case of this contract, how to use it and all the technical steps (off-chain & on-chain) & key methods of the contract.

The use case of this contract is to make use of digital signatures and be able to verify not only that a given account has signed a specific message, but also verify that the message in question has not been modified. As an amount parameter is included, it could be used to fulfill payments (and use the nonce parameter and the message parameter to avoid replay attacks). In this case, the verification that no nonce is reused should be done off-chain asi it has not been included in the smart contract. Otherwise, it could be done inside the verify method and reverts if nonce passed as parameter appears as used.

Off-chain, using Web3js for example and using metamask as wallet, a method, with which a user can hash and sign a message of interest with its private key (using the \x19Ethereum Signed Message:\n32 prefix as "getEthSignedMessageHash" method) , passing the same parameters that the smart contract method "getMessageHash" expects (specifying type in each parameter. Nonce parameter could be generated automatically) must be implemented. This method would retrieve a signed message that the user could send to the account associated with the "to" address passed as parameter in order to enable this account to verify it, and if it corresponds, claim for example a payment/reward/etc. User could send this signed message via email, or as he/she wants.

Once the target user receives this message, there should be an off-chain method with which this user could verify the signed message it has received. Along with the signed message, "verify" method, which would be called in this stage of verification, also needs the exact parameters used when signing. Passing exactly what parameters were signed off chain, makes it possible to validate the signature for that exact message. If the method receives only a message, not each of the parameters, it would be verifying if that message was in fact signed by a particular address but it could be any message signed by that account, it would not be verifying its content.
"to" parameter can be obtain from account that is trying to verify the message (through metamask for example), assuming users who received a signed message will be doing the verification for themselves, "signer" is the address of the account that send the signed message, the other arguments (amount, message and nonce) should be send off-chain alongside the signature, or maybe they could be store in some database associated to its corresponding signature (in this case, a method to enable users to obtain them given the signed message they received must be implemented).

Key methods:

- split signature: use inline assembly to split the bytes array of the signature. It verifies that it has 65 byte length and then starts reading from byte32. r and s are 32 bytes each, that is why it starts at byte 32 for r, then at byte 64 for s and for v it only extracts the first byte of the bytes loaded with mload.

- recoverSigner: given v,r and s extracted from a given signature and the signed message hash, it obtains the address of the actual signer with the ecrecover method. It is important that the signature from which r,v and s are obtained is generated using the \x19Ethereum Signed Message:\n32 prefix.
- verify is the central method. It receives all the parameters used when signing off-chain, signer address which is trying to be validated, and also the signature itself. Then it obtains the hash , as it was obtained off-chain, and adds the \x19Ethereum Signed Message:\n32 prefix (as it is added when signing off-chain). Finally it uses the recoverSigner method to obtain the actual signer, given the signed message hash and the signature, and checks if it is equal to signer passed as parameter or not.

H) Perform an audit of the contract "Staking.sol", find at least 3 technical technical, logic issues or hacks, explain why it is an issue and provide a way to fix those.

To begin with, _ierc20 state variable could be defined as immutable, making it assignable at the constructor but ensuring it only gets assigned once. Also, the constructor should receive an address which implements IERC20, and instantiate the IERC20 token with it; it should not receive the IERC20 instance itself.

As there is a duration state variable, there should be a state variable to keep track of when the current staking period is ending. This is to be used in a requirement when trying to modify the duration. State variable "_duration" should only be modified if the previous staking period has ended. Otherwise, as it is now, the owner can adjust this variable with no control and as this variable is used when computing rewards, it cannot be changed without restrictions. When computing rewards, the amount is affected by the coefficient of deltaDuration/duration, being1 the maximum possible value. If a user makes a deposit thinking the staking period is 5 weeks, lets his deposit for 6 weeks (thus, maximizing the previous mentioned coefficient) and once he is willing to withdraw, its rewards are calculated using an 12 weeks duration, it would get less rewards than expected without having any clue that this was happening. Moreover, as _duration is a public variable, it automatically gets a getter created, there is no need to create one.

In relation to how rewards are treated, rewards should not be computed and assumed that they are part of account deposit (as it is happening now that reward is computed and _balances is updated by adding the amount the user is willing to stake and the reward it has generated until that moment). There should be another state variable that keeps track of the rewards associated to each of the accounts that have made a deposit (another mapping(address=>uint25)) .This variable should be updated every time a user makes a deposit (stake), before _balances gets updated. This is related to the fact that there should be a way for users to get their rewards without having to withdraw everything they have staked and for this to be possible, rewards and _balances must be two different mappings.
There should be a method " withdrawRewards() " and a "withdraw(amount)" for a user to decide how much is willing to withdraw. As these methods, together with the deposit method, would need to

update rewards, associated with the account in question, before their execution, updating the rewards could be done with a modifier.

The “deposit(amount)” method does not verify that the contract address has an allowance to move “amount” tokens from sender to itself, this should be a requirement in this method. Also, it uses tx.origin for the transferFrom, but _msgSender() for all other steps. _msgSender(), in this case, returns msg.sender which could be another contract while tx.origin retrieves the EOA that started the transaction; this could lead to inconsistencies.

“withdraw()” method sets “_lastDepositTime” as block.timestamp before calling computeReward, but the “lastDepositTime” used when computing the reward should be the one corresponding to an actual deposit. Otherwise, if the last deposit was 6 weeks ago, by doing this it is ignoring that and considering it as being deposited exactly before computing the reward.

Furthermore, it could be useful to have “account” as an indexed parameter on Deposit and Withdraw events.

“addReward” method does not verify the allowance that the contract address has on behalf of _msgSender() before trying to transferFrom it. This should be done with the IERC20 method allowance(owner,spender). Any account that has been approved for X amount on the approveReward method, could easily set contract allowance on his behalf to 0 without the contract being aware of it until the transferFrom, at addReward method, gets reverted. As “rewardTotal” (variable used on the “require” statement on deposit method) is only updated when some user adds a reward, this contract is completely dependent on users adding rewards to the contract in order to function,

I) How would you build a proxy contract where the implementation lives in another contract in solidity (do not worry about syntax error or mispel)

```
contract Proxy is Ownable{
   address implementation;  //address of the implementation contract

   //function to update implementation contract the proxy delegates it calls to
   function updateImplementation(address _newImpl) public onlyOwner{
         implementation = _newImpl;
   }

   //contract fallback function
   function () external payable{
     assembly {
      let implContract := sload(0) //load first variable in storage (implementation address)
      calldatacopy(0x0,0x0, calldatasize)
      let result:= delegatecall(gas, implContract, 0x0, calldatasize, 0x0,0)
/*delegate call to implContract passing function data (which includes function signature) that was copied on previous line*/
      returndatacopy(0x0, 0x0, returndatasize) //copy return value
      switch result case 0 {revert(0, 0)} default {return (0, returndatasize)}
       /*switch checks boolean variable returned by delegatecall: It reverts if it is negative and return the result to  the caller if it is positive*/
      }
    }
}
```