EAS 583: Blockchains, Summer 2024

Author: Ivy Xie

Project: Build A Token Bridge between Two EVM-compatible Blockchains (Avalanche and BSC testnets)

In this project, I built a token bridge between two EVM-compatible blockchains (Ethereum Virtual Machine), specifically the Avalanche and BSC testnets, and sent transactions on these chains. The project can be divided into five parts.

In this first part of the project, I created accounts on Avalanche and BSC using the Python web3 library to create a private key for an account(s) on both Avalanche and BSC. Since I worked with two EVM chains, it was enough to generate one account. The private key resolved to the same address on the two networks. Once I created the account, I obtained funds from both BSC and Avalanche faucets sent to my new account. An account in Ethereum is an ECDSA private key (64 random bytes), and an account address, which is the 20 bytes derived from the ECDSA public key. Both the eth_accounts and web3 libraries provide tools for creating Ethereum accounts for me to use. I stored the private keys (or mnemonics) for the accounts I generated, because I needed to be able to access the funds in these accounts after I received funds from the faucets. To sign a message using a private key, I used the sign_message() function from web3. Alternatively, I can use sign_message() from eth_accounts. To verify a signature, I could use eth_account.Account.recover_message() or w3.eth.account.recover_message(), which has the same syntax.

In the second part of the project, I wrote the contract that controls the “wrapped” tokens on the destination side of the bridge. When a user deposits “real” tokens on the source side of the bridge, the bridge operator will mint “bridge-wrapped” tokens on the destination side. My bridge did the same thing, deploying a new bridge-wrapped token contract for each new type of asset that is bridged. Although the *contract* for each of these bridge-wrapped assets is different, the underlying *code* is the same, i.e., they are all controlled by copies of the same contract. The first time a user wishes to transfer an ERC20 (e.g. USDC) over the bridge, the owner of the destination contract will need to create new BridgeToken instance on the destination chain. Only “creator,” i.e., someone assigned the CREATOR_ROLE on the destination contract should be allowed to call this function. This allows the bridge owner to control what type of assets are bridged. When the createToken () function is called, it will deploy a new BridgeToken contract and return the address of the newly created contract. For access control and to prevent abuse, the bridge should only bridge “registered” tokens. Similarly, the destination contract should only issue a

“wrapped” token when there was a deposit on the source side. Since the deposit contract on the source chain cannot call the wrap () function on the destination chain directly, some intermediary must monitor the source chain and call the wrap () function on the destination chain. This intermediary is often called a “guardian” or a “warden.” In principle, the authority who registers new tokens could also be the same as the warden who announces deposit events, but a basic security principle is that it is good to separate roles.

In the third part of the project, I wrote the deposit contract for the source side of the chain. My token bridge accepts ERC20 deposits on the source chain. The bridge operator can register new tokens by calling the registerToken () function. The deposit contract needs to keep track of the list of registered tokens and will only accept deposits of tokens that have been registered by the bridge operator (i.e., the address that deployed the deposit contract). The registered tokens represent the list of token addresses that can be bridged. Users can deposit ERC20 tokens from the deposit contract. If the token being deposited is on the list of registered tokens, the deposit contract needs to emit a Deposit event so that the bridge operator can see that it is time to mint a BridgeToken on the destination side. Note that if a user simply “transfers” a token to the deposit contract (i.e., by calling the “transfer” function on the ERC20 contract), the token will not be sent over the bridge. Users must call “deposit” on the bridge contract. This is because calling transfer on the ERC20 contract does not notify the receiver. Instead, the deposit contract needs to use the approve + transfer pattern. When a user initiates a withdrawal on the destination side, the bridge operator will burn bridge-wrapped tokens on the destination side, then call “withdraw ()” on the deposit contract to release the “real” assets. The withdraw function needs to send the tokens to the target recipient, but the transaction should be rejected if the caller does not have the correct Role.

In the fourth part of the project, I used the web3 library to scan the blockchain for events emitted by contracts on the blockchain. I read data from the Avalanche and BSC testnets. Both networks provide free RPC endpoints that I can use. Ethereum nodes provide a way to “filter” transactions based on certain events using the web3 library, I can do this using the createFilter command. The web3 library does not check if the ABI I provided is correct, or complete. Instead, it only includes the description of the “Transfer” event. I recorded all the events I saw to the file “deposit_logs.csv” which have the following 6 columns: chain - String (either “avax” or “bsc”), token - Address of deposit token (in hex), recipient - Address of recipient (in hex), amount - Number of tokens being transferred, transactionHash - transaction hash (in hex) and address - The address of the contract that emitted the event (in hex).

In the final part of the five-part Bridge project, I integrated all the bridge code I wrote into a working bridge. I put all four parts together so that when I see an event on the source (destination) side, I sign a message and send it to the destination (source) side, by completing the file “bridge.py” to pass messages back and forth. The ScanBlocks () function takes a chain (either “avax” or “bsc”), a range of blocks and a contract address and scan the chain for ‘Deposit’ events emitted by the contract at the specified address. This script needs access to the signing key of the “warden” address, as well as the addresses of the source and destination contracts that I deployed.