First principles thinking is a term we often hear. It focuses on deeply understanding the foundational concepts of a subject to enable better thinking in the design space of the components which are built on top.

In the smart contract world, the “Ethereum Virtual Machine” along with its algorithms and data structures are the first principles. Solidity and the smart contracts we create are the components built on top of this foundation. To be a great Solidity dev one must have a deep understanding of the EVM.

This is the first in a series of articles that will deep dive into the EVM and build that foundational knowledge needed to become a “shadowy super coder”.

The Basics: Solidity → Bytecode → Opcode

Before we begin, this article assumes some basic knowledge of Solidity and how it’s deployed to the Ethereum chain. We’ll briefly touch on these subjects however if you’d like a refresher see this article

here.

As you know your Solidity code needs to be compiled into bytecode prior to being deployed to the Ethereum network. This bytecode corresponds to a series of opcode instructions that the EVM interprets.

This series will focus on specific parts of the compiled bytecode and illuminate how they work. By the end of each article, you should have a much clearer understanding of how each component functions. Along the way, you will learn lots of the foundational concepts relating to the EVM.

Today we’re going to look at a basic Solidity contract along with an excerpt of its bytecode/opcodes to demonstrate how the EVM selects functions.

The runtime bytecode created from Solidity contracts is a representation of the entire contract. Within the contract, you may have multiple functions that can be called once it is deployed.

A common question is how does the EVM know what bytecode to execute depending on which function of the contract is called. This is the first question we’ll use to help understand the underlying mechanics of the EVM and how this particular case is handled.

1_Storage.sol Breakdown

For our demo, we will use the 1_Storage.sol contract, which is one of the default contracts in the online Solidity IDE

Remix.

The contract has 2 functions store(uint256) and retrieve() that the EVM will have to decide between when a function call comes in. Below is the compiled runtime bytecode of the entire contract.

We are going to focus on the snippet of bytecode below. This snippet represents the function selector logic. Run “ctrl f” on the snippet to verify it is in the above bytecode.

This bytecode corresponds to a set of EVM opcodes and their input values. You can check out the list of EVM opcodes

here.

Opcodes are 1 byte in length leading to 256 different possible opcodes. The EVM only uses 140 unique opcodes.

The below shows the bytecode snippet broken into its corresponding opcode commands. These are run sequentially on the call stack by the EVM. You can navigate to the link above to verify the opcode number 60 = PUSH1 etc. By the end of the article, you’ll have a full understanding of what these do.

Smart Contract Function Calls & Calldata

Before diving deep into opcodes we need to quickly run through how we call a contract function.

When we call a contract function we include some calldata that specifies the function signature we are calling and any arguments that need to be passed in.

This can be done in Solidity with the following code.

Here we are making a contract call to the store function with argument 10. We use abi.encodeWithSignature() to get the calldata in the desired format. The emit logs our calldata for testing.

The above is what abi.encodeWithSignature("store(uint256)", 10) returns.

Earlier I mentioned function signatures, now let’s take a closer look at what they are.

Function signatures are defined as the first four bytes of the Keccak hash of the canonical representation of the function signature.

The canonical representation of the function signatures is the function name along with the function argument types ie. “store(uint256)” & “retrieve()”. Below are the function signatures of the 1_Storage.sol contract, try hashing store(uint256) yourself to verify the results

here.

Looking at our calldata above we can see that we have 36 bytes of calldata, the first 4 bytes of our calldata correspond to the function selector we just computed for the store(uint256) function.

The remaining 32 bytes correspond to our uint256 input argument. We have a hex value of “a” which is equal to 10 in decimal.

If we take the function signature 6057361d and refer back to the opcode section, run “ctrl f” on this value and see if you can find it.

Opcodes & The Call Stack

We now have everything we need to commence our deep dive into what goes on at the EVM level during function selection.

We are going to run through each of the opcode commands what they do and how they affect the call stack.

If you’re unfamiliar with how a stack data structure works watch this quick

videoas a primer.

We start with PUSH1 which tells the EVM to push the next 1 byte of data, 0x00 (0 in decimal), to the call stack. Why we do this will become apparent with the next opcode.

Next, we have CALLDATALOAD which pops off the first value on the stack (0) as input.

This opcode loads in the calldata to the stack using the “input” as an offset. Stack items are 32 bytes in size but our calldata is 36 bytes. The pushed value is msg.data[i:i+32] where “i” is this input. This ensures only 32 bytes are pushed to the stack but enables us to access any part of the calldata.

In this case, we have no offset (the value popped off of the stack was 0 from the previous PUSH1) so we push the first 32 bytes of the calldata to the call stack.

Remember earlier we logged our call data via an emit which equalled “0x6057361d000000000000000000000000000000000000000000000000000000000000000a”.

This means the trailing 4 bytes (“0000000a”) are lost. If we had wanted to access the uint256 variable we would have used an offset of 4 to omit the function signature but include the full variable.

Another PUSH1 this time with the hex value 0xe0 which has a decimal value of 224. Remember function signatures are 4 bytes long or 32 bits. Our loaded calldata is 32 bytes long or 256 bits. 256 - 32 = 224 you may see where this is going.

Next, we have SHR which is a bit shift right. It takes the first item off the stack (224) as an input of how much to shift by and the second item off the stack (0x6057361d0…00) represents what needs to be shifted. We can see after this operation we have our 4-byte function selector on the call stack.

If you are unfamiliar with how bit shifts work see

thisshort video.

Next is DUP1, a simple opcode that takes the value on the top of the stack and duplicates it.

PUSH4 pushes the 4 byte function signature of retrieve() (0x2e64cec1) onto the call stack.

In case you’re wondering how it knows this value, remember this is in the bytecode that was compiled from the solidity code. The compiler, therefore, had information on all function names and argument types.

EQ pops 2 values off of the stack, in this case, 0x2e64cec1 & 0x6057361d and checks if they’re equal. If they are it pushes a 1 back to the stack, if not a 0.

PUSH2 pushes 2 bytes of data onto the call stack 0x003b in hex which is equal to 59 in decimal.

The call stack has something called a program counter which specifies where in the bytecode the next execution command is. Here we set 59 because that is the location for the start of the retrieve() bytecode. (Note the EVM Playground section below will help crystallize how this works)

You can view the program counter location in a similar way to a line number location in your Solidity code. If the function is defined on line 59 you can use the line number as a way to tell the machine where to find the code for that function.

JUMPI stands for “jump if”. It pops 2 values off of the stack as input, the first (59) is the jump location and the second (0) is the bool value for whether this jump should be executed. Where 1 = true and 0 = false.

If it is true the program counter will be updated and the execution will jump to that location. In our case it is false, the program counter is not altered and the execution continues as normal.

DUP1 again.

PUSH4 pushes the 4 byte function signature of store(uint256) (0x6057361d) onto the call stack.

EQ again however this time the result is true as the function signatures match.

PUSH2, push the program counter location for the store(uint256) bytecode, 0x0059 in hex which is equal to 89 in decimal.

JUMPI, this time the bool check is true meaning the jump executes. This updates the program counter to 89 which will move the execution to a different part of the bytecode.

At this location, there will be a JUMPDEST opcode, without this opcode at the destination the JUMPI will fail.

There we have it, after this opcode executes you’ll be taken to the location of the store(uint156) bytecode and the execution of the function will continue as normal.

While this contract only had 2 functions the same principles apply to a contract with 20+ functions.

You now know how the EVM determines the location of the function bytecode that it needs to execute based on a contract function call. It’s actually just a simple set of “if statements” for each function in your contract along with their jump locations.

EVM Playground

I highly recommend visiting this

link.

The EVM playground will also help with your understanding of the program counter, in the code, you’ll see comments next to each command with its offset which represents its program counter location.

You’ll also see the calldata input to the left of the Run button, try changing this to the retrieve() call data 0x2e64cec1 to see how the execution changes. Just click Run and then the “step into” (curled arrow) button at the top right to jump through each opcode one by one.

Next, in the series, we take a trip down “Memory” lane in

EVM Deep Dives - Part 2.

See you soon.

noxx

Twitter

@noxx3xxon