The Ethereum Virtual Machine

At the heart of the Ethereum protocol and operation is the Ethereum Virtual Machine, or EVM for short. As you might guess from the name, it is a computation engine, not hugely dissimilar to the virtual machines of Microsoft's .NET Framework, or interpreters of other bytecode-compiled programming languages such as Java. In this chapter we take a detailed look at the EVM, including its instruction set, structure, and operation, within the context of Ethereum state updates.

What Is the EVM?

The EVM is the part of Ethereum that handles smart contract deployment and execution. Simple value transfer transactions from one EOA to another don't need to involve it, practically speaking, but everything else will involve a state update computed by the EVM. At a high level, the EVM running on the Ethereum blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with its own permanent data store.

The EVM is a quasi–Turing-complete state machine; "quasi" because all execution processes are limited to a finite number of computational steps by the amount of gas available for any given smart contract execution. As such, the halting problem is "solved" (all program executions will halt) and the situation where execution might (accidentally or maliciously) run forever, thus bringing the Ethereum platform to halt in its entirety, is avoided.

The EVM has a stack-based architecture, storing all in-memory values on a stack. It works with a word size of 256 bits (mainly to facilitate native hashing and elliptic curve operations) and has several addressable data components:

- An immutable *program code ROM*, loaded with the bytecode of the smart contract to be executed
- A volatile *memory*, with every location explicitly initialized to zero
- A permanent *storage* that is part of the Ethereum state, also zero-initialized

There is also a set of environment variables and data that is available during execution. We will go through these in more detail later in this chapter.

The Ethereum Virtual Machine (EVM) Architecture and Execution Context shows the EVM architecture and execution context.

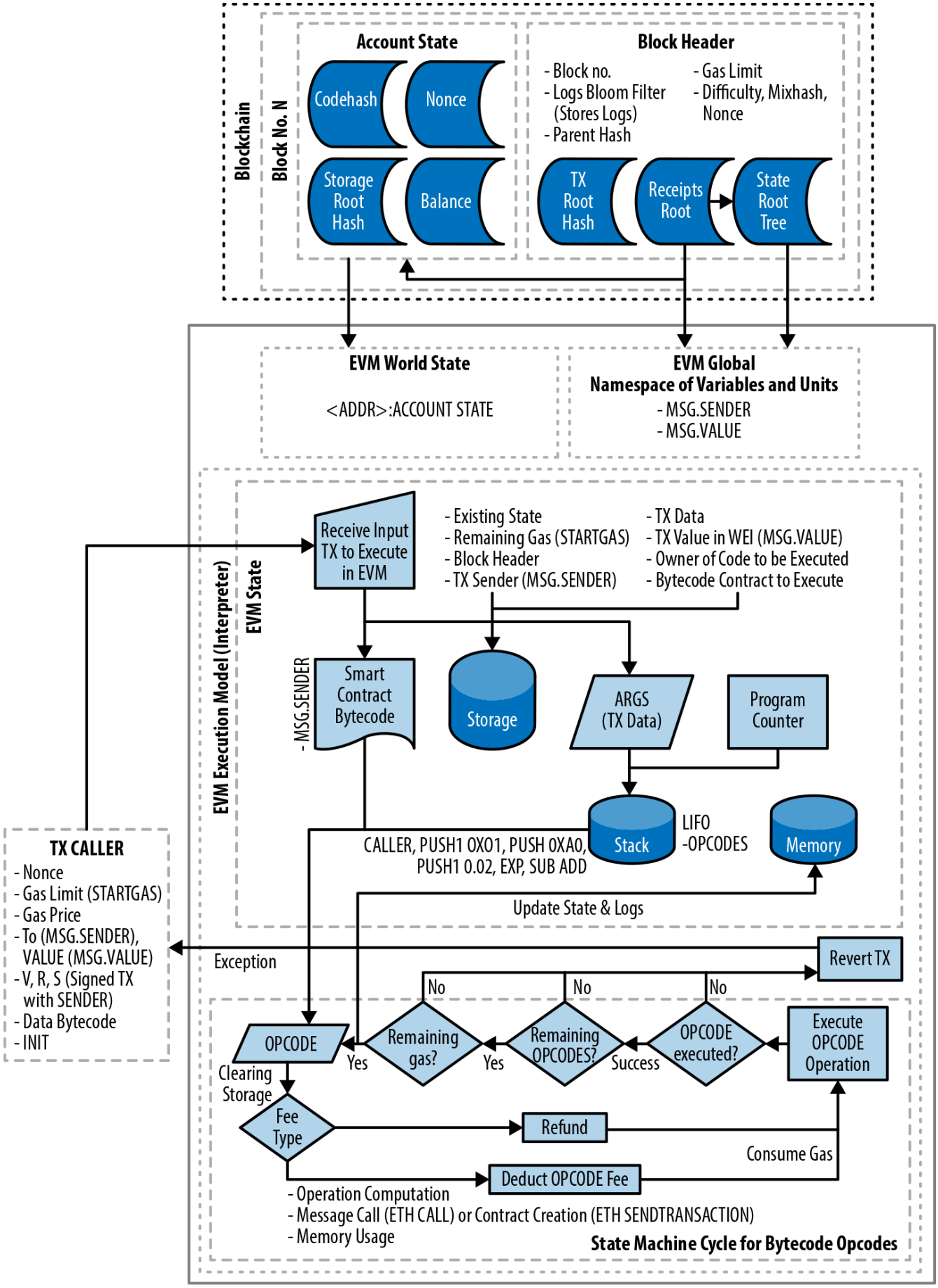

Figure 1. The Ethereum Virtual Machine (EVM) Architecture and Execution Context

Comparison with Existing Technology

The term "virtual machine" is often applied to the virtualization of a real computer, typically by a "hypervisor" such as VirtualBox or QEMU, or of an entire operating system instance, such as Linux's KVM. These must provide a software abstraction, respectively, of actual hardware, and of system calls and other kernel functionality.

The EVM operates in a much more limited domain: it is just a computation engine, and as such provides an abstraction of just computation and storage, similar to the Java Virtual Machine (JVM) specification, for example. From a high-level viewpoint, the JVM is designed to provide a runtime environment that is agnostic of the underlying host OS or hardware, enabling compatibility across a wide variety of systems. High-level programming languages such as Java or Scala (which use the JVM) or C# (which uses .NET) are compiled into the bytecode instruction set of their respective virtual machine. In the same way, the EVM executes its own bytecode instruction set (described in the next section), which higher-level smart contract programming languages such as LLL, Serpent, Mutan, or Solidity are compiled into.

The EVM, therefore, has no scheduling capability, because execution ordering is organized externally to it—Ethereum clients run through verified block transactions to determine which smart contracts need executing and in which order. In this sense, the Ethereum world computer is single-threaded, like JavaScript. Neither does the EVM have any "system interface" handling or "hardware support"—there is no physical machine to interface with. The Ethereum world computer is completely virtual.

The EVM Instruction Set (Bytecode Operations)

The EVM instruction set offers most of the operations you might expect, including:

- Arithmetic and bitwise logic operations
- Execution context inquiries
- Stack, memory, and storage access
- Control flow operations
- Logging, calling, and other operators

In addition to the typical bytecode operations, the EVM also has access to account information (e.g., address and balance) and block information (e.g., block number and current gas price).

Let's start our exploration of the EVM in more detail by looking at the available opcodes and what they do. As you might expect, all operands are taken from the stack, and the result (where applicable) is often put back on the top of the stack.

NOTE	A complete list of opcodes and their corresponding gas cost can be found in [evm_opcodes].

The available opcodes can be divided into the following categories:

Arithmetic operations

Arithmetic opcode instructions:

```
ADD        //Add the top two stack items
MUL        //Multiply the top two stack items
SUB        //Subtract the top two stack items
DIV        //Integer division
SDIV       //Signed integer division
MOD        //Modulo (remainder) operation
SMOD       //Signed modulo operation
ADDMOD     //Addition modulo any number
MULMOD     //Multiplication modulo any number
EXP        //Exponential operation
SIGNEXTEND //Extend the length of a two's complement signed integer
SHA3       //Compute the Keccak-256 hash of a block of memory
```

Note that all arithmetic is performed modulo 2^{256} (unless otherwise noted), and that the zeroth power of zero, 0^0, is taken to be 1.

Stack operations

Stack, memory, and storage management instructions:

```
POP     //Remove the top item from the stack
MLOAD   //Load a word from memory
MSTORE  //Save a word to memory
MSTORE8 //Save a byte to memory
SLOAD   //Load a word from storage
SSTORE  //Save a word to storage
MSIZE   //Get the size of the active memory in bytes
PUSHx   //Place x byte item on the stack, where x can be any integer from
        // 1 to 32 (full word) inclusive
DUPx    //Duplicate the x-th stack item, where x can be any integer from
        // 1 to 16 inclusive
SWAPx   //Exchange 1st and (x+1)-th stack items, where x can be any
        // integer from 1 to 16 inclusive
```

Process flow operations

Instructions for control flow:

```
STOP      //Halt execution
JUMP      //Set the program counter to any value
JUMPI     //Conditionally alter the program counter
PC        //Get the value of the program counter (prior to the increment
          //corresponding to this instruction)
JUMPDEST  //Mark a valid destination for jumps
```

System operations

Opcodes for the system executing the program:

```
LOGx          //Append a log record with x topics, where x is any integer
              //from 0 to 4 inclusive
CREATE        //Create a new account with associated code
CALL          //Message-call into another account, i.e. run another
              //account's code
CALLCODE      //Message-call into this account with another
              //account's code
RETURN        //Halt execution and return output data
DELEGATECALL  //Message-call into this account with an alternative
              //account's code, but persisting the current values for
              //sender and value
STATICCALL    //Static message-call into an account
REVERT        //Halt execution, reverting state changes but returning
              //data and remaining gas
INVALID       //The designated invalid instruction
SELFDESTRUCT  //Halt execution and register account for deletion
```

Logic operations

Opcodes for comparisons and bitwise logic:

```
LT     //Less-than comparison
GT     //Greater-than comparison
SLT    //Signed less-than comparison
SGT    //Signed greater-than comparison
EQ     //Equality comparison
ISZERO //Simple NOT operator
AND    //Bitwise AND operation
OR     //Bitwise OR operation
XOR    //Bitwise XOR operation
NOT    //Bitwise NOT operation
BYTE   //Retrieve a single byte from a full-width 256-bit word
```

Environmental operations

Opcodes dealing with execution environment information:

```
GAS           //Get the amount of available gas (after the reduction for
              //this instruction)
ADDRESS       //Get the address of the currently executing account
BALANCE       //Get the account balance of any given account
ORIGIN        //Get the address of the EOA that initiated this EVM
              //execution
CALLER        //Get the address of the caller immediately responsible
              //for this execution
CALLVALUE     //Get the ether amount deposited by the caller responsible
              //for this execution
CALLDATALOAD  //Get the input data sent by the caller responsible for
              //this execution
```

```
CALLDATASIZE   //Get the size of the input data
CALLDATACOPY   //Copy the input data to memory
CODESIZE       //Get the size of code running in the current environment
CODECOPY       //Copy the code running in the current environment to
               //memory
GASPRICE       //Get the gas price specified by the originating
               //transaction
EXTCODESIZE    //Get the size of any account's code
EXTCODECOPY    //Copy any account's code to memory
RETURNDATASIZE //Get the size of the output data from the previous call
               //in the current environment
RETURNDATACOPY //Copy data output from the previous call to memory
```

Block operations

Opcodes for accessing information on the current block:

```
BLOCKHASH  //Get the hash of one of the 256 most recently completed
           //blocks
COINBASE   //Get the block's beneficiary address for the block reward
TIMESTAMP  //Get the block's timestamp
NUMBER     //Get the block's number
DIFFICULTY //Get the block's difficulty
GASLIMIT   //Get the block's gas limit
```

Ethereum State

The job of the EVM is to update the Ethereum state by computing valid state transitions as a result of smart contract code execution, as defined by the Ethereum protocol. This aspect leads to the description of Ethereum as a *transaction-based state machine*, which reflects the fact that external actors (i.e., account holders and miners) initiate state transitions by creating, accepting, and ordering transactions. It is useful at this point to consider what constitutes the Ethereum state.

At the top level, we have the Ethereum *world state*. The world state is a mapping of Ethereum addresses (160-bit values) to*accounts*.At the lower level, each Ethereum address represents an account comprising an ether *balance* (stored as the number of wei owned by the account), a *nonce* (representing the number of transactions successfully sent from this account if it is an EOA, or the number of contracts created by it if it is a contract account), the account's *storage* (which is a permanent data store, only used by smart contracts), and the account's *program code* (again, only if the account is a smart contract account). An EOA will always have no code and an empty storage.

When a transaction results in smart contract code execution, an EVM is instantiated with all the information required in relation to the current block being created and the specific transaction being processed. In particular, the EVM's program code ROM is loaded with the code of the contract account being called, the program counter is set to zero, the storage is loaded from the contract account's storage, the memory is set to all zeros, and all the block and environment variables are set. A key variable is the gas supply for this execution, which is set to the amount of gas paid for by the sender at the start of the transaction (see Gas for more details). As code execution progresses, the gas supply is reduced according to the gas cost of the operations executed. If at any point the gas

supply is reduced to zero we get an "Out of Gas" (OOG) exception; execution immediately halts and the transaction is abandoned. No changes to the Ethereum state are applied, except for the sender's nonce being incremented and their ether balance going down to pay the block's beneficiary for the resources used to execute the code to the halting point. At this point, you can think of the EVM running on a sandboxed copy of the Ethereum world state, with this sandboxed version being discarded completely if execution cannot complete for whatever reason. However, if execution does complete successfully, then the real-world state is updated to match the sandboxed version, including any changes to the called contract's storage data, any new contracts created, and any ether balance transfers that were initiated.

Note that because a smart contract can itself effectively initiate transactions, code execution is a recursive process. A contract can call other contracts, with each call resulting in another EVM being instantiated around the new target of the call. Each instantiation has its sandbox world state initialized from the sandbox of the EVM at the level above. Each instantiation is also given a specified amount of gas for its gas supply (not exceeding the amount of gas remaining in the level above, of course), and so may itself halt with an exception due to being given too little gas to complete its execution. Again, in such cases, the sandbox state is discarded, and execution returns to the EVM at the level above.

Compiling Solidity to EVM Bytecode

Compiling a Solidity source file to EVM bytecode can be accomplished via several methods. In [intro_chapter] we used the online Remix compiler. In this chapter, we will use the solc executable at the command line. For a list of options, run the following command:

```
<pre data-type="programlisting">
$ <strong>solc --help</strong>
</pre>
```

Generating the raw opcode stream of a Solidity source file is easily achieved with the --opcodes command-line option. This opcode stream leaves out some information (the --asm option produces the full information), but it is sufficient for this discussion. For example, compiling an example Solidity file, *Example.sol*, and sending the opcode output into a directory named *BytecodeDir* is accomplished with the following command:

```
<pre data-type="programlisting">
$ <strong>solc -o BytecodeDir --opcodes Example.sol</strong>
</pre>
```

or:

```
<pre data-type="programlisting">
$ <strong>solc -o BytecodeDir --asm Example.sol</strong>
</pre>
```

The following command will produce the bytecode binary for our example program:

```
<pre data-type="programlisting">
$ <strong>solc -o BytecodeDir --bin Example.sol</strong>
</pre>
```

The output opcode files generated will depend on the specific contracts contained within the Solidity source file. Our simple Solidity file *Example.sol* has only one contract, named example:

```
pragma solidity ^0.4.19;

contract example {

  address contractOwner;

  function example() {
    contractOwner = msg.sender;
  }
}
```

As you can see, all this contract does is hold one persistent state variable, which is set as the address of the last account to run this contract.

If you look in the *BytecodeDir* directory you will see the opcode file *example.opcode*, which contains the EVM opcode instructions of the example contract. Opening the *example.opcode* file in a text editor will show the following:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH1 0xE JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH1 0x0 DUP1 PUSH2 0x100 EXP DUP2 SLOAD DUP2 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF MUL NOT AND SWAP1 DUP4 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND MUL OR SWAP1 SSTORE POP PUSH1
0x35 DUP1 PUSH1 0x5B PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60 PUSH1
0x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 JUMP
0xb9 SWAP14 0xcb 0x1e 0xdd RETURNDATACOPY 0xec 0xe0 0x1f 0x27 0xc9 PUSH5
0x9C5ABCC14A NUMBER 0x5e INVALID EXTCODESIZE 0xdb 0xcf EXTCODESIZE 0x27
EXTCODESIZE 0xe2 0xb8 SWAP10 0xed 0x
```

Compiling the example with the --asm option produces a file named *example.evm* in our *BytecodeDir* directory. This contains a slightly higher-level description of the EVM bytecode instructions, together with some helpful annotations:

```
/* "Example.sol":26:132  contract example {... */
  mstore(0x40, 0x60)
    /* "Example.sol":74:130  function example() {... */
  jumpi(tag_1, iszero(callvalue))
  0x0
  dup1
  revert
tag_1:
    /* "Example.sol":115:125  msg.sender */
  caller
    /* "Example.sol":99:112  contractOwner */
  0x0
  dup1
```

```
    /* "Example.sol":99:125  contractOwner = msg.sender */
  0x100
  exp
  dup2
  sload
  dup2
  0xffffffffffffffffffffffffffffffffffffffff
  mul
  not
  and
  swap1
  dup4
  0xffffffffffffffffffffffffffffffffffffffff
  and
  mul
  or
  swap1
  sstore
  pop
    /* "Example.sol":26:132  contract example {... */
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x0
  codecopy
  0x0
  return
stop

sub_0: assembly {
        /* "Example.sol":26:132  contract example {... */
      mstore(0x40, 0x60)
      0x0
      dup1
      revert

    auxdata: 0xa165627a7a7230582056b99dcb1edd3eece01f27c9649c5abcc14a435efe3b...
}
```

The --bin-runtime option produces the machine-readable hexadecimal bytecode:

```
60606040523415600e57600080fd5b336000806101000a81548173
ffffffffffffffffffffffffffffffffffffffff
021916908373
ffffffffffffffffffffffffffffffffffffffff
160217905550603580605b6000396000f3006060604052600080fd00a165627a7a7230582056b...
```

You can investigate what's going on here in detail using the opcode list given in The EVM Instruction Set (Bytecode Operations). However, that's quite a task, so let's just start by examining

the first four instructions:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
```

Here we have PUSH1 followed by a raw byte of value 0x60. This EVM instruction takes the single byte following the opcode in the program code (as a literal value) and pushes it onto the stack. It is possible to push values of size up to 32 bytes onto the stack, as in:

```
PUSH32 0x436f6e67726174756c6174696f6e732120536f6f6e20746f206d617374657221
```

The second PUSH1 opcode from *example.opcode* stores 0x40 onto the top of the stack (pushing the 0x60 already present there down one slot).

Next is MSTORE, which is a memory store operation that saves a value to the EVM's memory. It takes two arguments and, like most EVM operations, obtains them from the stack. For each argument the stack is "popped"; i.e., the top value on the stack is taken off and all the other values on the stack are shifted up one position. The first argument for MSTORE is the address of the word in memory where the value to be saved will be put. For this program we have 0x40 at the top of the stack, so that is removed from the stack and used as the memory address. The second argument is the value to be saved, which is 0x60 here. After the MSTORE operation is executed our stack is empty again, but we have the value 0x60 (96 in decimal) at the memory location 0x40.

The next opcode is CALLVALUE, which is an environmental opcode that pushes onto the top of the stack the amount of ether (measured in wei) sent with the message call that initiated this execution.

We could continue to step through this program in this way until we had a full understanding of the low-level state changes that this code effects, but it wouldn't help us at this stage. We'll come back to it later in the chapter.

Contract Deployment Code

There is an important but subtle difference between the code used when creating and deploying a new contract on the Ethereum platform and the code of the contract itself. In order to create a new contract, a special transaction is needed that has its to field set to the special 0x0 address and its data field set to the contract's *initiation code*. When such a contract creation transaction is processed, the code for the new contract account is *not* the code in the data field of the transaction. Instead, an EVM is instantiated with the code in the data field of the transaction loaded into its program code ROM, and then the output of the execution of that deployment code is taken as the code for the new contract account. This is so that new contracts can be programmatically initialized using the Ethereum world state at the time of deployment, setting values in the contract's storage and even sending ether or creating further new contracts.

When compiling a contract offline, e.g., using solc on the command line, you can either get the *deployment bytecode* or the *runtime bytecode*.

The deployment bytecode is used for every aspect of the initialization of a new contract account, including the bytecode that will actually end up being executed when transactions call this new

contract (i.e., the runtime bytecode) and the code to initialize everything based on the contract's constructor.

The runtime bytecode, on the other hand, is exactly the bytecode that ends up being executed when the new contract is called, and nothing more; it does not include the bytecode needed to initialize the contract during deployment.

Let's take the simple *Faucet.sol* contract we created earlier as an example:

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;

// Our first contract is a faucet!
contract Faucet {

  // Give out ether to anyone who asks
  function withdraw(uint withdraw_amount) public {

      // Limit withdrawal amount
      require(withdraw_amount <= 100000000000000000);

      // Send the amount to the address that requested it
      msg.sender.transfer(withdraw_amount);
    }

  // Accept any incoming amount
  function () external payable {}

}
```

To get the deployment bytecode, we would run `solc --bin Faucet.sol`. If we instead wanted just the runtime bytecode, we would run `solc --bin-runtime Faucet.sol`.

If you compare the output of these commands, you will see that the runtime bytecode is a subset of the deployment bytecode. In other words, the runtime bytecode is entirely contained within the deployment bytecode.

Disassembling the Bytecode

Disassembling EVM bytecode is a great way to understand how high-level Solidity acts in the EVM. There are a few disassemblers you can use to do this:

- *Porosity* is a popular open source decompiler.
- *Ethersplay* is an EVM plug-in for Binary Ninja, a disassembler.
- *IDA-Evm* is an EVM plugin for IDA, another disassembler.

In this section, we will be using the Ethersplay plug-in for Binary Ninja and to start Disassembling the Faucet runtime bytecode. After getting the runtime bytecode of *Faucet.sol*, we can feed it into Binary Ninja (after loading the Ethersplay plug-in) to see what the EVM instructions look like.

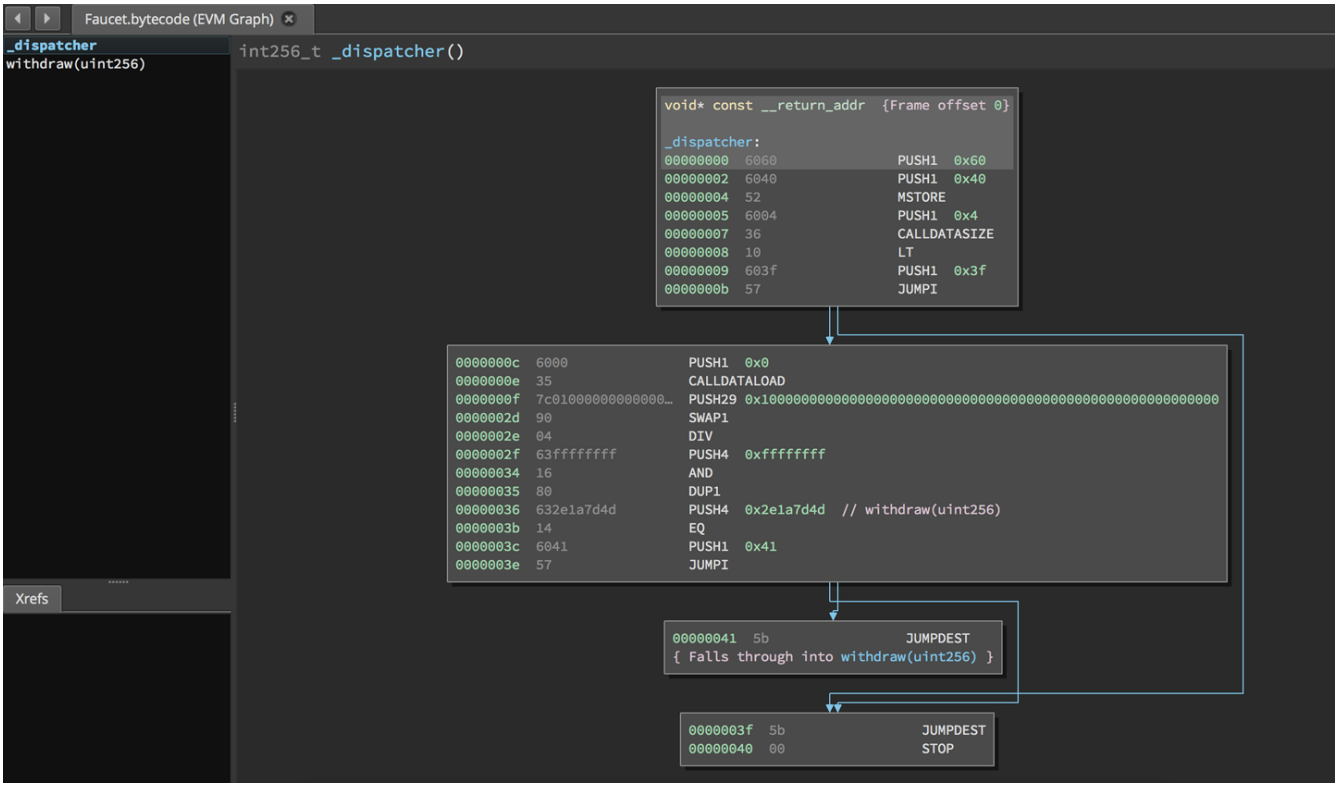

Figure 2. Disassembling the Faucet runtime bytecode

When you send a transaction to an ABI-compatible smart contract (which you can assume all contracts are), the transaction first interacts with that smart contract's *dispatcher*. The dispatcher reads in the data field of the transaction and sends the relevant part to the appropriate function. We can see an example of a dispatcher at the beginning of our disassembled *Faucet.sol* runtime bytecode. After the familiar MSTORE instruction, we see the following instructions:

```
PUSH1 0x4
CALLDATASIZE
LT
PUSH1 0x3f
JUMPI
```

As we have seen, PUSH1 0x4 places 0x4 onto the top of the stack, which is otherwise empty. CALLDATASIZE gets the size in bytes of the data sent with the transaction (known as the *calldata*) and pushes that number onto the stack. After these operations have been executed, the stack looks like this:

Stack
<length of calldata from tx>
0x4

This next instruction is LT, short for "less than." The LT instruction checks whether the top item on the stack is less than the next item on the stack. In our case, it checks to see if the result of CALLDATASIZE is less than 4 bytes.

Why does the EVM check to see that the calldata of the transaction is at least 4 bytes? Because of

how function identifiers work. Each function is identified by the first 4 bytes of its Keccak-256 hash. By placing the function's name and what arguments it takes into a keccak256 hash function, we can deduce its function identifier. In our case, we have:

```
keccak256("withdraw(uint256)") = 0x2e1a7d4d...
```

Thus, the function identifier for the withdraw(uint256) function is 0x2e1a7d4d, since these are the first 4 bytes of the resulting hash. A function identifier is always 4 bytes long, so if the entire data field of the transaction sent to the contract is less than 4 bytes, then there's no function with which the transaction could possibly be communicating, unless a *fallback function* is defined. Because we implemented such a fallback function in *Faucet.sol*, the EVM jumps to this function when the calldata's length is less than 4 bytes.

LT pops the top two values off the stack and, if the transaction's data field is less than 4 bytes, pushes 1 onto it. Otherwise, it pushes 0. In our example, let's assume the data field of the transaction sent to our contract *was* less than 4 bytes.

The PUSH1 0x3f instruction pushes the byte 0x3f onto the stack. After this instruction, the stack looks like this:

Stack
0x3f
1

The next instruction is JUMPI, which stands for "jump if." It works like so:

```
jumpi(label, cond) // Jump to "label" if "cond" is true
```

In our case, label is 0x3f, which is where our fallback function lives in our smart contract. The cond argument is 1, which was the result of the LT instruction earlier. To put this entire sequence into words, the contract jumps to the fallback function if the transaction data is less than 4 bytes.

At 0x3f, only a STOP instruction follows, because although we declared a fallback function, we kept it empty. As you can see in JUMPI instruction leading to fallback function, had we not implemented a fallback function, the contract would throw an exception instead.

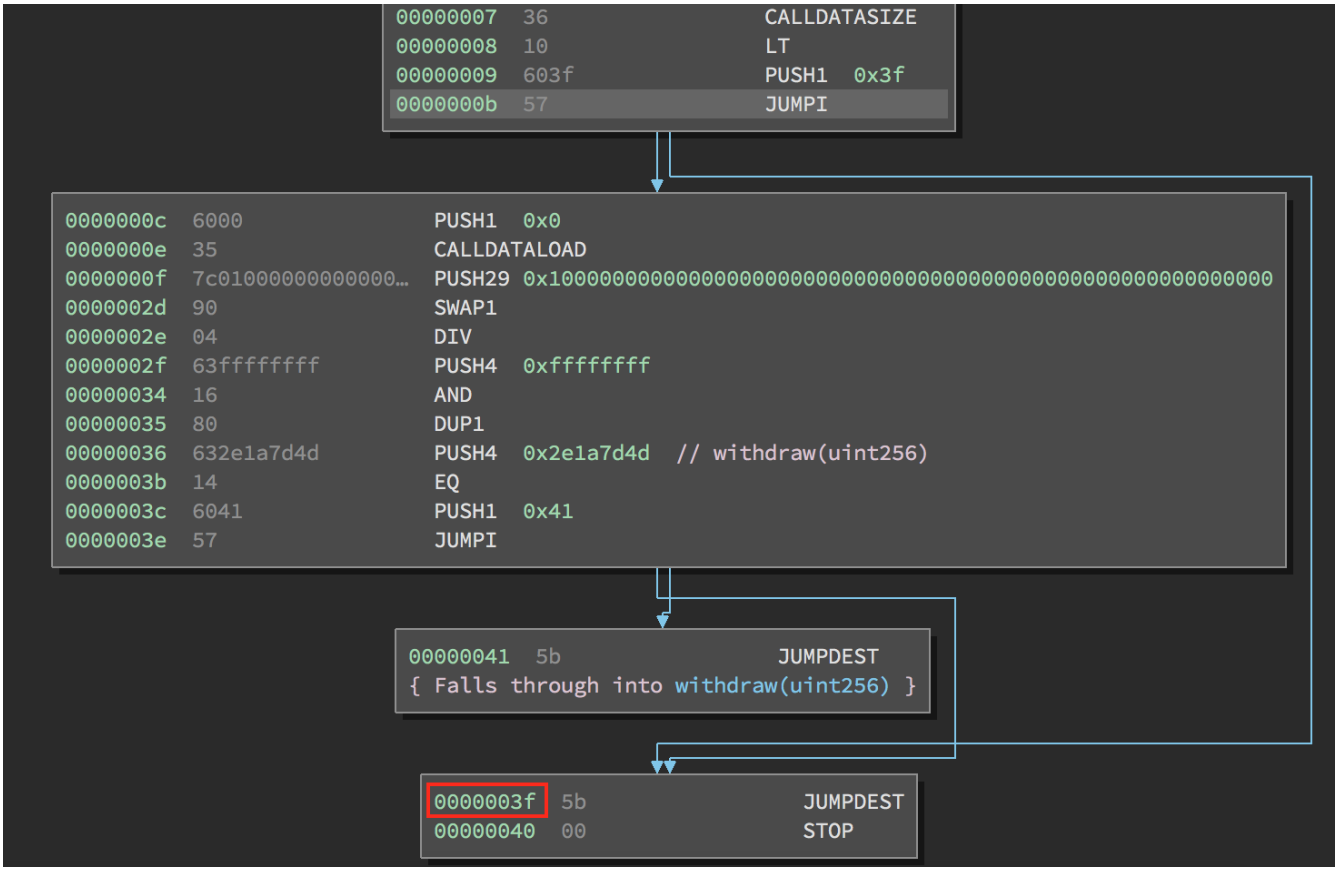

Figure 3. JUMPI instruction leading to fallback function

Let's examine the central block of the dispatcher. Assuming we received calldata that was *greater* than 4 bytes in length, the JUMPI instruction would not jump to the fallback function. Instead, code execution would proceed to the following instructions:

```
PUSH1 0x0
CALLDATALOAD
PUSH29 0x1000000...
SWAP1
DIV
PUSH4 0xffffffff
AND
DUP1
PUSH4 0x2e1a7d4d
EQ
PUSH1 0x41
JUMPI
```

PUSH1 0x0 pushes 0 onto the stack, which is now otherwise empty again. CALLDATALOAD accepts as an argument an index within the calldata sent to the smart contract and reads 32 bytes from that index, like so:

```
calldataload(p) //load 32 bytes of calldata starting from byte position p
```

Since 0 was the index passed to it from the PUSH1 0x0 command, CALLDATALOAD reads 32 bytes of

calldata starting at byte 0, and then pushes it to the top of the stack (after popping the original 0x0). After the PUSH29 0x1000000... instruction, the stack is then:

Stack
0x1000000... (29 bytes in length)
<32 bytes of calldata starting at byte 0>

SWAP1 switches the top element on the stack with the *i*-th element after it. In this case, it swaps 0x1000000... with the calldata. The new stack is:

Stack
<32 bytes of calldata starting at byte 0>
0x1000000... (29 bytes in length)

The next instruction is DIV, which works as follows:

```
div(x, y) // integer division x / y
```

In this case, x = 32 bytes of calldata starting at byte 0, and y = 0x100000000... (29 bytes total). Can you think of why the dispatcher is doing the division? Here's a hint: we read 32 bytes from calldata earlier, starting at index 0. The first 4 bytes of that calldata is the function identifier.

The 0x100000000... we pushed earlier is 29 bytes long, consisting of a 1 at the beginning, followed by all 0s. Dividing our 32 bytes of calldata by this value will leave us only the *topmost 4 bytes* of our calldata load, starting at index 0. These 4 bytes—the first 4 bytes in the calldata starting at index 0—are the function identifier, and this is how the EVM extracts that field.

If this part isn't clear to you, think of it like this: in base 10, 1234000 / 1000 = 1234. In base 16, this is no different. Instead of every place being a multiple of 10, it is a multiple of 16. Just as dividing by 10^3 (1000) in our smaller example kept only the topmost digits, dividing our 32-byte base 16 value by 16^{29} does the same.

The result of the DIV (the function identifier) gets pushed onto the stack, and our stack is now:

Stack
<function identifier sent in data>

Since the PUSH4 0xffffffff and AND instructions are redundant, we can ignore them entirely, as the stack will remain the same after they are done. The DUP1 instruction duplicates the first item on the stack, which is the function identifier. The next instruction, PUSH4 0x2e1a7d4d, pushes the precalculated function identifier of the `withdraw(uint256)` function onto the stack. The stack is now:

Stack
0x2e1a7d4d

Stack
<function identifier sent in data>
<function identifier sent in data>

The next instruction, EQ, pops off the top two items of the stack and compares them. This is where the dispatcher does its main job: it compares whether the function identifier sent in the msg.data field of the transaction matches that of `withdraw(uint256)`. If they are equal, EQ pushes 1 onto the stack, which will ultimately be used to jump to the withdraw function. Otherwise, EQ pushes 0 onto the stack.

Assuming the transaction sent to our contract indeed began with the function identifier for withdraw(uint256), our stack has become:

Stack
1
<function identifier sent in data> (now known to be 0x2e1a7d4d)

Next, we have PUSH1 0x41, which is the address at which the withdraw(uint256) function lives in the contract. After this instruction, the stack looks like this:

Stack
0x41
1
function identifier sent in msg.data

The JUMPI instruction is next, and it once again accepts the top two elements on the stack as arguments. In this case, we have jumpi(0x41, 1), which tells the EVM to execute the jump to the location of the withdraw(uint256) function, and the execution of that function's code can proceed.

Turing Completeness and Gas

As we have already touched on, in simple terms, a system or programming language is *Turing complete* if it can run any program. This capability, however, comes with a very important caveat: some programs take forever to run. An important aspect of this is that we can't tell, just by looking at a program, whether it will take forever or not to execute. We have to actually go through with the execution of the program and wait for it to finish to find out.Of course, if it is going to take forever to execute, we will have to wait forever to find out. This is called the *halting problem* and would be a huge problem for Ethereum if it were not addressed.

Because of the halting problem, the Ethereum world computer is at risk of being asked to execute a program that never stops. This could be by accident or malice. We have discussed that Ethereum acts like a single-threaded machine, without any scheduler, and so if it became stuck in an infinite loop this would mean it would become unusable.

However, with gas, there is a solution: if after a prespecified maximum amount of computation has been performed, the execution hasn't ended, the execution of the program is halted by the EVM. This makes the EVM a *quasi*–Turing-complete machine: it can run any program you feed into it, but only if the program terminates within a particular amount of computation. That limit isn't fixed in Ethereum—you can pay to increase it up to a maximum (called the "block gas limit"), and everyone can agree to increase that maximum over time. Nevertheless, at any one time, there is a limit in place, and transactions that consume too much gas while executing are halted.

In the following sections, we will look at gas and examine how it works in detail.

Gas

Gas is Ethereum's unit for measuring the computational and storage resources required to perform actions on the Ethereum blockchain. In contrast to Bitcoin, whose transaction fees only take into account the size of a transaction in kilobytes, Ethereum must account for every computational step performed by transactions and smart contract code execution.

Each operation performed by a transaction or contract costs a fixed amount of gas. Some examples, from the Ethereum Yellow Paper:

- Adding two numbers costs 3 gas
- Calculating a Keccak-256 hash costs 30 gas + 6 gas for each 256 bits of data being hashed
- Sending a transaction costs 21,000 gas

Gas is a crucial component of Ethereum, and serves a dual role: as a buffer between the (volatile) price of Ethereum and the reward to miners for the work they do, and as a defense against denial-of-service attacks. To prevent accidental or malicious infinite loops or other computational wastage in the network, the initiator of each transaction is required to set a limit to the amount of computation they are willing to pay for. The gas system thereby disincentivizes attackers from sending "spam" transactions, as they must pay proportionately for the computational, bandwidth, and storage resources that they consume.

Gas Accounting During Execution

When an EVM is needed to complete a transaction, in the first instance it is given a gas supply equal to the amount specified by the gas limit in the transaction. Every opcode that is executed has a cost in gas, and so the EVM's gas supply is reduced as the EVM steps through the program. Before each operation, the EVM checks that there is enough gas to pay for the operation's execution. If there isn't enough gas, execution is halted and the transaction is reverted.

If the EVM reaches the end of execution successfully, without running out of gas, the gas cost used is paid to the miner as a transaction fee, converted to ether based on the gas price specified in the transaction:

```
miner fee = gas cost * gas price
```

The gas remaining in the gas supply is refunded to the sender, again converted to ether based on

the gas price specified in the transaction:

```
remaining gas = gas limit - gas cost
refunded ether = remaining gas * gas price
```

If the transaction "runs out of gas" during execution, the operation is immediately terminated, raising an "out of gas" exception. The transaction is reverted and all changes to the state are rolled back.

Although the transaction was unsuccessful, the sender will be charged a transaction fee, as miners have already performed the computational work up to that point and must be compensated for doing so.

Gas Accounting Considerations

The relative gas costs of the various operations that can be performed by the EVM have been carefully chosen to best protect the Ethereum blockchain from attack. You can see a detailed table of gas costs for different EVM opcodes in [evm_opcodes_table].

More computationally intensive operations cost more gas. For example, executing the SHA3 function is 10 times more expensive (30 gas) than the ADD operation (3 gas). More importantly, some operations, such as EXP, require an additional payment based on the size of the operand. There is also a gas cost to using EVM memory and for storing data in a contract's on-chain storage.

The importance of matching gas cost to the real-world cost of resources was demonstrated in 2016 when an attacker found and exploited a mismatch in costs. The attack generated transactions that were very computationally expensive, and made the Ethereum mainnet almost grind to a halt. This mismatch was resolved by a hard fork (codenamed "Tangerine Whistle") that tweaked the relative gas costs.

Gas Cost Versus Gas Price

While the gas *cost* is a measure of computation and storage used in the EVM, the gas itself also has a *price* measured in ether. When performing a transaction, the sender specifies the gas price they are willing to pay (in ether) for each unit of gas, allowing the market to decide the relationship between the price of ether and the cost of computing operations (as measured in gas):

```
transaction fee = total gas used * gas price paid  (in ether)
```

When constructing a new block, miners on the Ethereum network can choose among pending transactions by selecting those that offer to pay a higher gas price. Offering a higher gas price will therefore incentivize miners to include your transaction and get it confirmed faster.

In practice, the sender of a transaction will set a gas limit that is higher than or equal to the amount of gas expected to be used. If the gas limit is set higher than the amount of gas consumed, the sender will receive a refund of the excess amount, as miners are only compensated for the work they actually perform.

It is important to be clear about the distinction between the *gas cost* and the *gas price*. To recap:

- Gas cost is the number of units of gas required to perform a particular operation.
- Gas price is the amount of ether you are willing to pay per unit of gas when you send your transaction to the Ethereum network.

> **TIP** While gas has a price, it cannot be "owned" nor "spent." Gas exists only inside the EVM, as a count of how much computational work is being performed. The sender is charged a transaction fee in ether, which is then converted to gas for EVM accounting and then back to ether as a transaction fee paid to the miners.

Negative gas costs

Ethereum encourages the deletion of used storage variables and accounts by refunding some of the gas used during contract execution.

There are two operations in the EVM with negative gas costs:

- Deleting a contract (SELFDESTRUCT) is worth a refund of 24,000 gas.
- Changing a storage address from a nonzero value to zero (SSTORE[x] = 0) is worth a refund of 15,000 gas.

To avoid exploitation of the refund mechanism, the maximum refund for a transaction is set to half the total amount of gas used (rounded down).

Block Gas Limit

The block gas limit is the maximum amount of gas that may be consumed by all the transactions in a block, and constrains how many transactions can fit into a block.

For example, let's say we have 5 transactions whose gas limits have been set to 30,000, 30,000, 40,000, 50,000, and 50,000. If the block gas limit is 180,000, then any four of those transactions can fit in a block, while the fifth will have to wait for a future block. As previously discussed, miners decide which transactions to include in a block. Different miners are likely to select different combinations, mainly because they receive transactions from the network in a different order.

If a miner tries to include a transaction that requires more gas than the current block gas limit, the block will be rejected by the network. Most Ethereum clients will stop you from issuing such a transaction by giving a warning along the lines of "transaction exceeds block gas limit." The block gas limit on the Ethereum mainnet is 8 million gas at the time of writing according to https://etherscan.io, meaning that around 380 basic transactions (each consuming 21,000 gas) could fit into a block.

Who decides what the block gas limit is?

The miners on the network collectively decide the block gas limit. Individuals who want to mine on the Ethereum network use a mining program, such as Ethminer, which connects to a Geth or Parity Ethereum client. The Ethereum protocol has a built-in mechanism where miners can vote on the gas limit so capacity can be increased or decreased in subsequent blocks. The miner of a block can

vote to adjust the block gas limit by a factor of 1/1,024 (0.0976%) in either direction. The result of this is an adjustable block size based on the needs of the network at the time. This mechanism is coupled with a default mining strategy where miners vote on a gas limit that is at least 4.7 million gas, but which targets a value of 150% of the average of recent total gas usage per block (using a 1,024-block exponential moving average).

Conclusions

In this chapter we have explored the Ethereum Virtual Machine, tracing the execution of various smart contracts and looking at how the EVM executes bytecode. We also looked at gas, the EVM's accounting mechanism, and saw how it solves the halting problem and protects Ethereum from denial-of-service attacks. Next, in [consensus], we will look at the mechanism used by Ethereum to achieve decentralized consensus.