title: "Reverse Engineering a Contract" description: How to understand a contract when you don't have the source code author: Ori Pomerantz lang: en tags: ["evm", "opcodes"] skill: advanced published: 2021-12-30

Introduction {#introduction}

There are no secrets on the blockchain, everything that happens is consistent, verifiable, and publicly available. Ideally, contracts should have their source code published and verified on Etherscan. However, that is not always the case. In this article you learn how to reverse engineer contracts by looking at a contract without source code, `0x2510c039cc3b061d79e564b38836da87e31b342f`.

There are reverse compilers, but they don't always produce usable results. In this article you learn how to manually reverse engineer and understand a contract from the opcodes, as well as how to interpret the results of a decompiler.

To be able to understand this article you should already know the basics of the EVM, and be at least somewhat familiar with EVM assembler. You can read about these topics here.

Prepare the Executable Code {#prepare-the-executable-code}

You can get the opcodes by going to Etherscan for the contract, clicking the **Contract** tab and then **Switch to Opcodes View**. You get a view that is one opcode per line.

To be able to understand jumps, however, you need to know where in the code each opcode is located. To do that, one way is to open a Google Spreadsheet and paste the opcodes in column C. You can skip the following steps by making a copy of this already prepared spreadsheet.

The next step is to get the correct code locations so we'll be able to understand jumps. We'll put the opcode size in column B, and the location (in hexadecimal) in column A. Type this function in cell `B1` and then copy and paste it for the rest of column B, until the end of the code. After you do this you can hide column B.

```
=1+IF(REGEXMATCH(C1,"PUSH"),REGEXEXTRACT(C1,"PUSH(\d+)"),0)
```

First this function adds one byte for the opcode itself, and then looks for `PUSH`. Push opcodes are special because they need to have additional bytes for the value being pushed. If the opcode is a `PUSH`, we extract the number of bytes and add that.

In `A1` put the first offset, zero. Then, in `A2`, put this function and again copy and paste it for the rest of column A:

```
=dec2hex(hex2dec(A1)+B1)
```

We need this function to give us the hexadecimal value because the values that are pushed prior to jumps (`JUMP` and `JUMPI`) are given to us in hexadecimal.

The Entry Point (0x00) {#the-entry-point-0x00}

Contracts are always executed from the first byte. This is the initial part of the code:

| Offset | Opcode | Stack (after the opcode) | | -----: | ------------ | ----------------------- | | 0 | PUSH1 0x80 | 0x80 | | 2 | PUSH1 0x40 | 0x40, 0x80 | | 4 | MSTORE | Empty | | 5 | PUSH1 0x04 | 0x04 | | 7 | CALLDATASIZE | CALLDATASIZE 0x04 | | 8 | LT | CALLDATASIZE<4 | | 9 | PUSH2 0x005e | 0x5E CALLDATASIZE<4 | | C | JUMPI | Empty |

This code does two things:

1. Write 0x80 as a 32 byte value to memory locations 0x40-0x5F (0x80 is stored in 0x5F, and 0x40-0x5E are all zeroes).
2. Read the calldata size. Normally the call data for an Ethereum contract follows the ABI (application binary interface), which at a minimum requires four bytes for the function selector. If the call data size is less than four, jump to 0x5E.

The Handler at 0x5E (for non-ABI call data) {#the-handler-at-0x5e-for-non-abi-call-data}

| Offset | Opcode | | -----: | ------------ | | 5E | JUMPDEST | | 5F | CALLDATASIZE | | 60 | PUSH2 0x007c | | 63 | JUMPI |

This snippet starts with a `JUMPDEST`. EVM (Ethereum virtual machine) programs throw an exception if you jump to an opcode that isn't `JUMPDEST`. Then it looks at the CALLDATASIZE, and if it is "true" (that is, not zero) jumps to 0x7C. We'll get to that below.

| Offset | Opcode | Stack (after opcode) | | -----: | ---------- | --- | | 64 | CALLVALUE | [Wei](#) provided by the call. Called `msg.value` in Solidity | | 65 | PUSH1 0x06 | 6 CALLVALUE | | 67 | PUSH1 0x00 | 0 6 CALLVALUE | | 69 | DUP3 | CALLVALUE 0 6 CALLVALUE | | 6A | DUP3 | 6 CALLVALUE 0 6 CALLVALUE | | 6B | SLOAD | Storage[6] CALLVALUE 0 6 CALLVALUE |

So when there is no call data we read the value of Storage[6]. We don't know what this value is yet, but we can look for transactions that the contract received with no call data. Transactions which just transfer ETH without any call data (and therefore no method) have in Etherscan the method `Transfer`. In fact, [the very first transaction the contract received](#) is a transfer.

If we look in that transaction and click **Click to see More**, we see that the call data, called input data, is indeed empty (`0x`). Notice also that the value is 1.559 ETH, that will be relevant later.

Next, click the **State** tab and expand the contract we're reverse engineering (0x2510...). You can see that `Storage[6]` did change during the transaction, and if you change Hex to **Number**, you see it became 1,559,000,000,000,000,000, the value transferred in wei (I added the commas for clarity), corresponding to the next contract value.

If we look in the state changes caused by [other `Transfer` transactions from the same period](#) we see that `Storage[6]` tracked the value of the contract for a while. For now we'll call it `Value*`. The asterisk (`*`) reminds us that we don't *know* what this variable does yet, but it can't be just to track the contract value because there's no need to use storage, which is very expensive, when you can get your accounts balance using `ADDRESS BALANCE`. The first opcode pushes the contract's own address. The second one reads the address at the top of the stack and replaces it with the balance of that address.

| Offset | Opcode | Stack | | -----: | ------------ | -- | | 6C | PUSH2 0x0075 | 0x75 Value* CALLVALUE 0 6 CALLVALUE | | 6F | SWAP2 | CALLVALUE Value* 0x75 0 6 CALLVALUE | | 70 | SWAP1 | Value* CALLVALUE 0x75 0 6 CALLVALUE | | 71 | PUSH2 0x01a7 | 0x01A7 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 74 | JUMP |

We'll continue to trace this code at the jump destination.

| Offset | Opcode | Stack | | -----: | ---------- | --- | | 1A7 | JUMPDEST | Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1A8 | PUSH1 0x00 | 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1AA | DUP3 | CALLVALUE 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1AB | NOT | 2^256-CALLVALUE-1 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE |

The `NOT` is bitwise, so it reverses the value of every bit in the call value.

| Offset | Opcode | Stack | | -----: | ------------ | -- | | 1AC | DUP3 | Value* 2^25 CALLVALUE-1 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1AD | GT | Value*>2^256-CALLVALUE-1 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1AE | ISZERO | Value*<=2^256-CALLVALUE-1 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1AF | PUSH2 0x01df | 0x01DF Value*<=2^256-CALLVALUE-1 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1B2 | JUMPI |

We jump if `Value*` is smaller than 2^256-CALLVALUE-1 or equal to it. This looks like logic to prevent overflow. And indeed, we see that after a few nonsense operations (writing to memory is about to get deleted, for example) at offset 0x01DE the contract reverts if the overflow is detected, which is normal behavior.

Note that such an overflow is extremely unlikely, because it would require the call value plus `Value*` to be comparable to 2^256 wei, about 10^59 ETH. [The total ETH supply, at writing, is less than two hundred million](#)

| Offset | Opcode | Stack | | -----: | -------- | -- | | 1DF | JUMPDEST | 0x00 Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1E0 | POP | Value* CALLVALUE 0x75 0 6 CALLVALUE | | 1E1 | ADD | Value*+CALLVALUE 0x75 0 6 CALLVALUE | | 1E2 | SWAP1 | 0x75 Value*+CALLVALUE 0 6 CALLVALUE | | 1E3 | JUMP |

If we got here, get `Value* + CALLVALUE` and jump to offset 0x75.

| Offset | Opcode | Stack | | -----: | -------- | ------------------------------ | | 75 | JUMPDEST | Value*+CALLVALUE 0 6 CALLVALUE | | 76 | SWAP1 | 0 Value*+CALLVALUE 6 CALLVALUE | | 77 | SWAP2 | 6 Value*+CALLVALUE 0 CALLVALUE | | 78 | SSTORE | 0 CALLVALUE |

If we get here (which requires the call data to be empty) we add to `Value*` the call value. This is consistent with what we say `Transfer` transactions do.

| Offset | Opcode | | -----: | ------ | | 79 | POP | | 7A | POP | | 7B | STOP |

Finally, clear the stack (which isn't necessary) and signal the successful end of the transaction.

To sum it all up, here's a flowchart for the initial code.

The Handler at 0x7C {#the-handler-at-0x7c}

I purposely did not put in the heading what this handler does. The point isn't to teach you how this specific contract works, but how to reverse engineer contracts. You will learn what it does the same way I did, by following the code.

We get here from several places:

- If there is call data of 1, 2, or 3 bytes (from offset 0x63)
- If the method signature is unknown (from offsets 0x42 and 0x5D)

| Offset | Opcode | Stack | | -----: | ------------ | ------------------- | | 7C | JUMPDEST | | 7D | PUSH1 0x00 | 0x00 | | 7F | PUSH2 0x009d | 0x9D 0x00 | | 82 | PUSH1 0x03 | 0x03 0x9D 0x00 | | 84 | SLOAD | Storage[3] 0x9D 0x00 |

This is another storage cell, one that I couldn't find in any transactions so it's harder to know what it means. The code below will make it clearer.

| Offset | Opcode | Stack | | -----: | -- | ------------------------------ | | 85 | PUSH20 0xffffffffffffffffffffffffffffffffffffffff | 0xff....ff Storage[3] 0x9D 0x00 | | 9A | AND | Storage[3]-as-address 0x9D 0x00 |

These opcodes truncate the value we read from Storage[3] to 160 bits, the length of an Ethereum address.

| Offset | Opcode | Stack | | -----: | ------ | ------------------------------ | | 9B | SWAP1 | 0x9D Storage[3]-as-address 0x00 | | 9C | JUMP | Storage[3]-as-address 0x00 |

This jump is superfluous, since we're going to the next opcode. This code isn't nearly as gas-efficient as it could be.

| Offset | Opcode | Stack | | -----: | ---------- | ------------------------------ | | 9D | JUMPDEST | Storage[3]-as-address 0x00 | | 9E | SWAP1 | 0x00 Storage[3]-as-address | | 9F | POP | Storage[3]-as-address | | A0 | PUSH1 0x40 | 0x40 Storage[3]-as-address | | A2 | MLOAD | Mem[0x40] Storage[3]-as-address |

In the very beginning of the code we set Mem[0x40] to 0x80. If we look for 0x40 later, we see that we don't change it - so we can assume it is 0x80.

| Offset | Opcode | Stack | | -----: | ------------ | -- | | A3 | CALLDATASIZE | CALLDATASIZE 0x80 Storage[3]-as-address | | A4 | PUSH1 0x00 | 0x00 CALLDATASIZE 0x80 Storage[3]-as-address | | A6 | DUP3 | 0x80 0x00 CALLDATASIZE 0x80 Storage[3]-as-address | | A7 | CALLDATACOPY | 0x80 Storage[3]-as-address |

Copy all the call data to memory, starting at 0x80.

| Offset | Opcode | Stack | | -----: | ------------- | -- | | A8 | PUSH1 0x00 | 0x00 0x80 Storage[3]-as-address | | AA | DUP1 | 0x00 0x00 0x80 Storage[3]-as-address | | AB | CALLDATASIZE | CALLDATASIZE 0x00 0x00 0x80 Storage[3]-as-address | | AC | DUP4 | 0x80 CALLDATASIZE 0x00 0x00 0x80 Storage[3]-as-address | | AD | DUP6 | Storage[3]-as-address 0x80 CALLDATASIZE 0x00 0x00 0x80 Storage[3]-as-address | | AE | GAS | GAS Storage[3]-as-address 0x80 CALLDATASIZE 0x00 0x00 0x80 Storage[3]-as-address | | AF | DELEGATE_CALL |

Now things are a lot clearer. This contract can act as a [proxy](), calling the address in Storage[3] to do the real work. `DELEGATE_CALL` calls a separate contract, but stays in the same storage. This means that the delegated contract, the one we are a proxy for, accesses the same storage space. The parameters for the call are:

- *Gas*: All the remaining gas
- *Called address*: Storage[3]-as-address
- *Call data*: The CALLDATASIZE bytes starting at 0x80, which is where we put the original call data
- *Return data*: None (0x00 - 0x00) We'll get the return data by other means (see below)

| Offset | Opcode | Stack | | -----: | ------------- | -- | | B0 | RETURNDATASIZE | RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | B1 | DUP1 | RETURNDATASIZE RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | B2 | PUSH1 0x00 | 0x00 RETURNDATASIZE RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | B4 | DUP5 | 0x80 0x00 RETURNDATASIZE RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | B5 | RETURNDATACOPY | RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address |

Here we copy all the return data to the memory buffer starting at 0x80.

| Offset | Opcode | Stack | | -----: | ------------ | -- | | B6 | DUP2 | (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | B7 | DUP1 | (((call success/failure))) (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | B8 | ISZERO | (((did the call fail))) (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | B9 | PUSH2 0x00c0 | 0xC0 (((did the call fail))) (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | BC | JUMPI | (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | BD | DUP2 | RETURNDATASIZE (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | BE | DUP5 | 0x80 RETURNDATASIZE (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | BF | RETURN | |

So after the call we copy the return data to the buffer 0x80 - 0x80+RETURNDATASIZE, and if the call is successful we then `RETURN` with exactly that buffer.

DELEGATECALL Failed {#delegatecall-failed}

If we get here, to 0xC0, it means that the contract we called reverted. As we are just a proxy for that contract, we want to return the same data and also revert.

| Offset | Opcode | Stack | | -----: | -------- | --- | | | JUMPDEST | (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | C1 | DUP2 | RETURNDATASIZE (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | C2 | DUP5 | 0x80 RETURNDATASIZE (((call success/failure))) RETURNDATASIZE (((call success/failure))) 0x80 Storage[3]-as-address | | C3 | REVERT |

So we `REVERT` with the same buffer we used for`RETURN` earlier: 0x80 - 0x80+RETURNDATASIZE

ABI calls {#abi-calls}

If the call data size is four bytes or more this might be a valid ABI call.

| Offset | Opcode | Stack | | -----: | ------------ | --- | | D | PUSH1 0x00 | 0x00 | | F | CALLDATALOAD | (((First word (256 bits) of the call data))) | | 10 | PUSH1 0xe0 | 0xE0 (((First word (256 bits) of the call data))) | | 12 | SHR | (((first 32 bits (4 bytes) of the call data))) |

Etherscan tells us that `1C` is an unknown opcode, because [it was added after Etherscan wrote this feature](#) and they haven't updated it. An [up to date opcode table](#) shows us that this is shift right

| Offset | Opcode | Stack | | -----: | --------------- | --- | | 1 DUP1 | (((first 32 bits (4 bytes) of the call data))) (((first 32 bits (4 bytes) of the call data))) | | 14 | PUSH4 0x3cd8045e | 0x3CD8045E (((first 32 bits (4 bytes) of the call data))) (((first 32 bits (4 bytes) of the call data))) | | 19 | GT | 0x3CD8045E>first-32-bits-of-the-call-data (((first 32 bits (4 bytes) of the call data))) | | 1A | PUSH2 0x0043 | 0x43 0x3CD8045E>first-32-bits-of-the-call-data (((first 32 bits (4 bytes) of the call data))) | | 1D | JUMPI | (((first 32 bits (4 bytes) of the call data))) |

By dividing the method signature matching tests in two like this saves half the tests on average. The code that immediately follows this and the code in 0x43 follow the same pattern: `DUP1` the first 32 bits of the call data, `PUSH4 (((method signature>`, run `EQ` to check for equality, and then `JUMPI` if the method signature matches. Here are the method signatures, their addresses, and if known [the corresponding method definition]:

Method	Method signature	Offset to jump into
[splitter()]	0x3cd8045e	0x0103
???	0x81e580d3	0x0138
[currentWindow()]	0xba0bafb4	0x0158
???	0x1f135823	0x00C4
[merkleRoot()]	0x2eb4a7ab	0x00ED

If no match is found, the code jumps to [the proxy handler at 0x7C], in the hope that the contract to which we are a proxy has a match.

splitter() {#splitter}

Offset	Opcode	Stack
103	JUMPDEST	
104	CALLVALUE	CALLVALUE
105	DUP1	CALLVALUE CALLVALUE
106	ISZERO	CALLVALUE==0 CALLVALUE
107	PUSH2 0x010f	0x010F CALLVALUE==0 CALLVALUE
10A	JUMPI	CALLVALUE
10B	PUSH1 0x00	0x00 CALLVALUE
10D	DUP1	0x00 0x00 CALLVALUE
10E	REVERT	

The first thing this function does is check that the call did not send any ETH. This function is not [`payable`]. If somebody sent us ETH that must be a mistake and we want to `REVERT` to avoid having that ETH where they can't get it back.

Offset	Opcode	Stack
10F	JUMPDEST	
110	POP	
111	PUSH1 0x03	0x03
113	SLOAD	(((Storage[3] a.k.a the contract for which we are a proxy)))
114	PUSH1 0x40	0x40 (((Storage[3] a.k.a the contract for which we are a proxy)))
116	MLOAD	0x80 (((Storage[3] a.k.a the contract for which we are a proxy)))
117	PUSH20 0xffffffffffffffffffffffffffffffffffffffff	0xFF...FF 0x80 (((Storage[3] a.k.a the contract for which we are a proxy)))
12C	SWAP1	0x80 0xFF...FF (((Storage[3] a.k.a the contract for which we are a proxy)))
12D	SWAP2	(((Storage[3] a.k.a the contract for which we are a proxy))) 0xFF...FF 0x80
12E	AND	ProxyAddr 0x80
12F	DUP2	0x80 ProxyAddr 0x80
130	MSTORE	0x80

And 0x80 now contains the proxy address

Offset	Opcode	Stack
131	PUSH1 0x20	0x20 0x80
133	ADD	0xA0
134	PUSH2 0x00e4	0xE4 0xA0
137	JUMP	0xA0

The E4 Code {#the-e4-code}

This is the first time we see these lines, but they are shared with other methods (see below). So we'll call the value in the stack X, and just remember that in `splitter()` the value of this X is 0xA0.

Offset	Opcode	Stack
E4	JUMPDEST	X
E5	PUSH1 0x40	0x40 X
E7	MLOAD	0x80 X
E8	DUP1	0x80 0x80 X
E9	SWAP2	X 0x80 0x80
EA	SUB	X-0x80 0x80
EB	SWAP1	0x80 X-0x80
EC	RETURN	

So this code receives a memory pointer in the stack (X), and causes the contract to `RETURN` with a buffer that is 0x80 - X.

In the case of `splitter()`, this returns the address for which we are a proxy. `RETURN` returns the buffer in 0x80-0x9F, which is where we wrote this data (offset 0x130 above).

currentWindow() {#currentwindow}

The code in offsets 0x158-0x163 is identical to what we saw in 0x103-0x10E in `splitter()` (other than the `JUMPI` destination), so we know `currentWindow()` is also not `payable`.

Offset	Opcode	Stack
164	JUMPDEST	
165	POP	
166	PUSH2 0x00da	0xDA
169	PUSH1 0x01	0x01 0xDA
16B	SLOAD	Storage[1] 0xDA
16C	DUP2	0xDA Storage[1] 0xDA
16D	JUMP	Storage[1] 0xDA

The DA code {#the-da-code}

This code is also shared with other methods. So we'll call the value in the stack Y, and just remember that in`currentWindow()` the value of this Y is Storage[1].

| Offset | Opcode | Stack | | -----: | ---------- | ---------------- | | DA | JUMPDEST | Y 0xDA | | DB | PUSH1 0x40 | 0x40 Y 0xDA | | DD | MLOAD | 0x80 Y 0xDA | | DE | SWAP1 | Y 0x80 0xDA | | DF | DUP2 | 0x80 Y 0x80 0xDA | | E0 | MSTORE | 0x80 0xDA |

Write Y to 0x80-0x9F.

| Offset | Opcode | Stack | | -----: | ---------- | -------------- | | E1 | PUSH1 0x20 | 0x20 0x80 0xDA | | E3 | ADD | 0xA0 0xDA |

And the rest is already explained[above](). So jumps to 0xDA write the stack top (Y) to 0x80-0x9F, and return that value. In the case of `currentWindow()`, it returns Storage[1].

merkleRoot() {#merkleroot}

The code in offsets 0xED-0xF8 is identical to what we saw in 0x103-0x10E in`splitter()` (other than the `JUMPI` destination), so we know `merkleRoot()` is also not`payable`.

| Offset | Opcode | Stack | | -----: | ------------ | -------------------- | | F9 | JUMPDEST | | FA | POP | | FB | PUSH2 0x00da | 0xDA | | FE | PUSH1 0x00 | 0x00 0xDA | | 100 | SLOAD | Storage[0] 0xDA | | 101 | DUP2 | 0xDA Storage[0] 0xDA | | 102 | JUMP | Storage[0] 0xDA |

What happens after the jump[we already figured out](). So `merkleRoot()` returns Storage[0].

0x81e580d3 {#0x81e580d3}

The code in offsets 0x138-0x143 is identical to what we saw in 0x103-0x10E in`splitter()` (other than the `JUMPI` destination), so we know this function is also not `payable`.

| Offset | Opcode | Stack | | -----: | ------------ | --- | | 144 | JUMPDEST | | 145 | POP | | 146 PUSH2 0x00da | 0xDA | | 149 | PUSH2 0x0153 | 0x0153 0xDA | | 14C | CALLDATASIZE | CALLDATASIZE 0x0153 0xDA | | 14D | PUSH1 0x04 | 0x04 CALLDATASIZE 0x0153 0xDA | | 14F | PUSH2 0x018f | 0x018F 0x04 CALLDATASIZE 0x0153 0xDA | | 152 | JUMP | 0x04 CALLDATASIZE 0x0153 0xDA | | 18F | JUMPDEST | 0x04 CALLDATASIZE 0x0153 0xDA | | 190 | PUSH1 0x00 | 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 192 | PUSH1 0x20 | 0x20 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 194 | DUP3 | 0x04 0x20 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 195 | DUP5 | CALLDATASIZE 0x04 0x20 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 196 | SUB | CALLDATASIZE-4 0x20 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 197 | SLT | CALLDATASIZE-4<32 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 198 | ISZERO | CALLDATASIZE-4>=32 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 199 | PUSH2 0x01a0 | 0x01A0 CALLDATASIZE-4>=32 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 19C | JUMPI | 0x00 0x04 CALLDATASIZE 0x0153 0xDA |

It looks like this function takes at least 32 bytes (one word) of call data.

| Offset | Opcode | Stack | | -----: | ------ | -- | | 19D | DUP1 | 0x00 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 19E | DUP2 | 0x00 0x00 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 19F | REVERT |

If it doesn't get the call data the transaction is reverted without any return data.

Let's see what happens if the function*does* get the call data it needs.

| Offset | Opcode | Stack | | -----: | ----------- | -- | | 1A0 | JUMPDEST | 0x00 0x04 CALLDATASIZE 0x0153 0xDA | | 1A1 | POP | 0x04 CALLDATASIZE 0x0153 0xDA | | 1A2 | CALLDATALOAD | calldataload(4) CALLDATASIZE 0x0153 0xDA |

`calldataload(4)` is the first word of the call data*after* the method signature

| Offset | Opcode | Stack | | -----: | ----------- | --- | | 1A3 | SWAP2 | 0x0153 CALLDATASIZE calldataload(4) 0xDA | | 1A4 | SWAP1 | CALLDATASIZE 0x0153 calldataload(4) 0xDA | | 1A5 | POP | 0x0153 calldataload(4) 0xDA | | 1A6 | JUMP | calldataload(4) 0xDA | | 153 | JUMPDEST | calldataload(4) 0xDA | | 154 | PUSH2 0x016e | 0x016E calldataload(4) 0xDA | | 157 | JUMP | calldataload(4) 0xDA | | 16E | JUMPDEST | calldataload(4) 0xDA | | 16F | PUSH1

0x04 | 0x04 calldataload(4) 0xDA | | 171 | DUP2 | calldataload(4) 0x04 calldataload(4) 0xDA | | 172 | DUP2 | 0x04 calldataload(4) 0x04 calldataload(4) 0xDA | | 173 | SLOAD | Storage[4] calldataload(4) 0x04 calldataload(4) 0xDA | | 174 | DUP2 | calldataload(4) Storage[4] calldataload(4) 0x04 calldataload(4) 0xDA | | 175 | LT | calldataload(4)<Storage[4] calldataload(4) 0x04 calldataload(4) 0xDA | | 176 | PUSH2 0x017e | 0x017EC calldataload(4)<Storage[4] calldataload(4) 0x04 calldataload(4) 0xDA | | 179 | JUMPI | calldataload(4) 0x04 calldataload(4) 0xDA |

If the first word is not less than Storage[4], the function fails. It reverts without any returned value:

| Offset | Opcode | Stack | | -----: | ---------- | ------------ | | 17A | PUSH1 0x00 | 0x00 ... | | 17C | DUP1 | 0x00 0x00 ... | | 17D | REVERT |

If the calldataload(4) is less than Storage[4], we get this code:

| Offset | Opcode | Stack | | -----: | ---------- | --- | | 17E | JUMPDEST | calldataload(4) 0x04 calldataload(4) 0xDA | | 17F | PUSH1 0x00 | 0x00 calldataload(4) 0x04 calldataload(4) 0xDA | | 181 | SWAP2 | 0x04 calldataload(4) 0x00 calldataload(4) 0xDA | | 182 | DUP3 | 0x00 0x04 calldataload(4) 0x00 calldataload(4) 0xDA | | 183 | MSTORE | calldataload(4) 0x00 calldataload(4) 0xDA |

And memory locations 0x00-0x1F now contain the data 0x04 (0x00-0x1E are all zeros, 0x1F is four)

| Offset | Opcode | Stack | | -----: | ---------- | -- | | 184 | PUSH1 0x20 | 0x20 calldataload(4) 0x00 calldataload(4) 0xDA | | 186 | SWAP1 | calldataload(4) 0x20 0x00 calldataload(4) 0xDA | | 187 | SWAP2 | 0x00 0x20 calldataload(4) calldataload(4) 0xDA | | 188 | SHA3 | (((SHA3 of 0x00-0x1F))) calldataload(4) calldataload(4) 0xDA | | 189 | ADD | (((SHA3 of 0x00-0x1F)))+calldataload(4) calldataload(4) 0xDA | | 18A | SLOAD | Storage[(((SHA3 of 0x00-0x1F))) + calldataload(4)] calldataload(4) 0xDA |

So there is a lookup table in storage, which starts at the SHA3 of 0x000...0004 and has an entry for every legitimate call data value (value below Storage[4]).

| Offset | Opcode | Stack | | -----: | ------ | --- | | 18B | SWAP1 | calldataload(4) Storage[(((SHA3 of 0x00-0x1F))) + calldataload(4)] 0xDA | | 18C | POP | Storage[(((SHA3 of 0x00-0x1F))) + calldataload(4)] 0xDA | | 18D | DUP2 | 0xDA Storage[(((SHA3 of 0x00-0x1F))) + calldataload(4)] 0xDA | | 18E | JUMP | Storage[(((SHA3 of 0x00-0x1F))) + calldataload(4)] 0xDA |

We already know what the code at offset 0xDA does, it returns the stack top value to the caller. So this function returns the value from the lookup table to the caller.

0x1f135823 {#0x1f135823}

The code in offsets 0xC4-0xCF is identical to what we saw in 0x103-0x10E in `splitter()` (other than the `JUMPI` destination), so we know this function is also not `payable`.

| Offset | Opcode | Stack | | -----: | ----------- | ----------------- | | D0 | JUMPDEST | | | D1 | POP | | | D2 | PUSH2 0x00da | 0xDA | | D5 | PUSH1 0x06 | 0x06 0xDA | | D7 | SLOAD | Value* 0xDA | | D8 | DUP2 | 0xDA Value* 0xDA | | D9 | JUMP | Value* 0xDA |

We already know what the code at offset 0xDA does, it returns the stack top value to the caller. So this function returns `Value*`.

Method Summary {#method-summary}

Do you feel you understand the contract at this point? I don't. So far we have these methods:

| Method | Meaning | | --------------------------------- | --- | | Transfer | Acce the value provided by the call and increase `Value*` by that amount | | splitter() | Return Storage[3], the proxy address | | currentWindow() | Return Storage[1] | | merkleRoot() | Return Storage[0] | | 0x81e580d3 | Return the value from a lookup table, provided the parameter is less than Storage[4] | | 0x1f135823 | Return Storage[6], a.k.a. Value* |

But we know any other functionality is provided by the contract in Storage[3]. Maybe if we knew what that contract is it'll give us a clue. Thankfully, this is the blockchain and everything is known, at least in theory. We didn't see any methods that set Storage[3], so it must have been set by the constructor.

The Constructor {#the-constructor}

When we look at a contract we can also see the transaction that created it.

If we click that transaction, and then the **State** tab, we can see the initial values of the parameters. Specifically, we can see that Storage[3] contains 0x2f81e57ff4f4d83b40a9f719fd892d8e806e0761. That contract must contain the missing functionality. We can understand it using the same tools we used for the contract we are investigating.

The Proxy Contract {#the-proxy-contract}

Using the same techniques we used for the original contract above we can see that the contract reverts if:

- There is any ETH attached to the call (0x05-0x0F)
- The call data size is less than four (0x10-0x19 and 0xBE-0xC2)

And that the methods it supports are:

Method	Method signature	Offset to jump into
scaleAmountByPercentage(uint256,uint256)	0x8ffb5c97	0x0135
isClaimed(uint256,address)	0xd2ef0795	0x0151
claim(uint256,address,uint256,bytes32[])	0x2e7ba6ef	0x00F4
incrementWindow()	0x338b1d31	0x0110
???	0x3f26479e	0x0118
???	0x1e7df9d3	0x00C3
currentWindow()	0xba0bafb4	0x0148
merkleRoot()	0x2eb4a7ab	0x0107
???	0x81e580d3	0x0122
???	0x1f135823	0x00D8

We can ignore the bottom four methods because we will never get to them. Their signatures are such that our original contract takes care of them by itself (you can click the signatures to see the details above), so they must be methods that are overridden.

One of the remaining methods is `claim(<params>)`, and another is `isClaimed(<params>)`, so it looks like an airdrop contract. Instead of going through the rest opcode by opcode, we can try the decompiler, which produces usable results for three functions from this contract. Reverse engineering the other ones is left as an exercise to the reader.

scaleAmountByPercentage {#scaleamountbypercentage}

This is what the decompiler gives us for this function:

```
python def unknown8ffb5c97(uint256 _param1, uint256 _param2) payable: require calldata.size - 4 >=' 64 if _param1 and _param2 > -1 / _param1: revert with 0, 17 return (_param1 * _param2 / 100 * 10^6)
```

The first `require` tests that the call data has, in addition to the four bytes of the function signature, at least 64 bytes, enough for the two parameters. If not then there is obviously something wrong.

The `if` statement seems to check that `_param1` is not zero, and that `_param1 * _param2` is not negative. It is probably to prevent cases of wrap around.

Finally, the function returns a scaled value.

claim {#claim}

The code the decompiler creates is complex, and not all of it is relevant for us. I am going to skip some of it to focus on the lines that I believe provide useful information

```
python def unknown2e7ba6ef(uint256 _param1, uint256 _param2, uint256 _param3, array _param4) payable: ... require _param2 == addr(_param2) ... if currentWindow <= _param1: revert with 0, 'cannot claim for a future window'
```

We see here two important things:

- `_param2`, while it is declared as a `uint256`, is actually an address
- `_param1` is the window being claimed, which has to be `currentWindow` or earlier.

```
python ... if stor5[_claimWindow][addr(_claimFor)]: revert with 0, 'Account already claimed the given window'
```

So now we know that Storage[5] is an array of windows and addresses, and whether the address claimed the reward for that window.

```
python ... idx = 0 s = 0 while idx < _param4.length: ... if s + sha3(mem[(32 * _param4.length) + 328 len mem[(32 * _param4.length) + 296]]) > mem[(32 * idx) + 296]: mem[mem[64] + 32] = mem[(32 * idx) + 296] ... s = sha3(mem[_62 + 32 len mem[_62]]) continue ... s = sha3(mem[_66 + 32 len mem[_66]]) continue if unknown2eb4a7ab != s: revert with 0, 'Invalid proof'
```

We know that `unknown2eb4a7ab` is actually the function `merkleRoot()`, so this code looks like it is verifying a merkle proof. This means that `_param4` is a merkle proof.

```
python call addr(_param2) with: value unknown81e580d3[_param1] * _param3 / 100 * 10^6 wei gas 30000 wei
```

This is how a contract transfers its own ETH to another address (contract or externally owned). It calls it with a value that is the amount to be transferred. So it looks like this is an airdrop of ETH.

```
python if not return_data.size: if not ext_call.success: require ext_code.size(stor2) call stor2.deposit() with: value unknown81e580d3[_param1] * _param3 / 100 * 10^6 wei
```

The bottom two lines tell us that Storage[2] is also a contract that we call. If we look at the constructor transaction we see that this contract is 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2, a Wrapped Ether contract whose source code has been uploaded to Etherscan.

So it looks like the contracts attempts to send ETH to `_param2`. If it can do it, great. If not, it attempts to send WETH. If `_param2` is an externally owned account (EOA) then it can always receive ETH, but contracts can refuse to receive ETH. However, WETH is ERC-20 and contracts can't refuse to accept that.

```
python ... log 0xdbd5389f: addr(_param2), unknown81e580d3[_param1] * _param3 / 100 * 10^6, bool(ext_call.success)
```

At the end of the function we see a log entry being generated. Look at the generated log entries and filter on the topic that starts with `0xdbd5...`. If we click one of the transactions that generated such an entry we see that indeed it looks like a claim - the account sent a message to the contract we're reverse engineering, and in return got ETH.

1e7df9d3 {#1e7df9d3}

This function is very similar to `claim` above. It also checks a merkle proof, attempts to transfer ETH to the first, and produces the same type of log entry.

```
python def unknown1e7df9d3(uint256 _param1, uint256 _param2, array _param3) payable: ... idx = 0 s = 0 while idx < _param3.length: if idx >= mem[96]: revert with 0, 50 _55 = mem[(32 * idx) + 128] if s + sha3(mem[(32 * _param3.length) + 160 len mem[(32 * _param3.length) + 128]]) > mem[(32 * idx) + 128]: ... s = sha3(mem[_58 + 32 len mem[_58]]) continue mem[mem[64] + 32] = s + sha3(mem[(32 * _param3.length) + 160 len mem[(32 * _param3.length) + 128]]) ... if unknown2eb4a7ab != s: revert with 0, 'Invalid proof' ... call addr(_param1) with: value s wei gas 30000 wei if not return_data.size: if not ext_call.success: require ext_code.size(stor2) call stor2.deposit() with: value s wei gas gas_remaining wei ... log 0xdbd5389f: addr(_param1), s, bool(ext_call.success)
```

The main difference is that the first parameter, the window to withdraw, isn't there. Instead, there is a loop over all the windows that could be claimed.

```
python idx = 0 s = 0 while idx < currentWindow: ... if stor5[mem[0]]: if idx == -1: revert with 0, 17 idx = idx + 1 s = s continue ... stor5[idx][addr(_param1)] = 1 if idx >= unknown81e580d3.length: revert with 0, 50 mem[0] = 4 if unknown81e580d3[idx] and _param2 > -1 / unknown81e580d3[idx]: revert with 0, 17 if s > !(unknown81e580d3[idx] * _param2 / 100 * 10^6): revert with 0, 17 if idx == -1: revert with 0, 17 idx = idx + 1 s = s + (unknown81e580d3[idx] * _param2 / 100 * 10^6) continue
```

So it looks like a `claim` variant that claims all the windows.

Conclusion {#conclusion}

By now you should know how to understand contracts whose source code is not available, using either the opcodes or (when it works) the decompiler. As is evident from the length of this article, reverse engineering a contract is not trivial, but in a system where security is essential it is an important skill to be able to verify contracts work as promised.