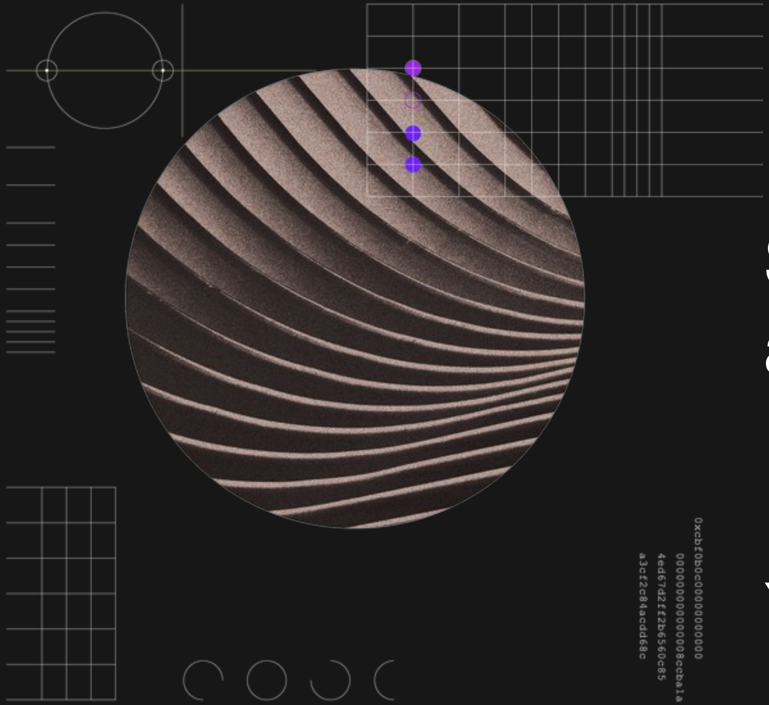
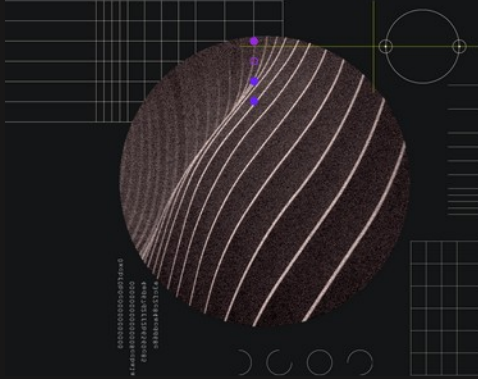


# So, You Think You Can Write an EVM Decompiler?

Yannis Smaragdakis ( DEDAUB , U.Athens)

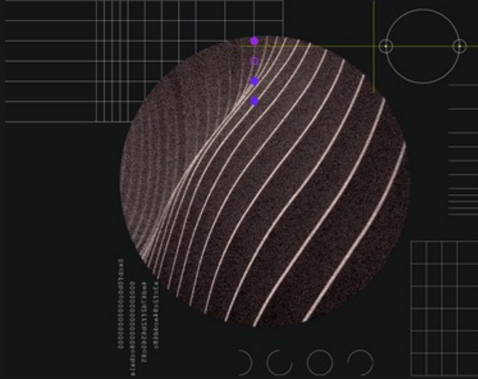
# Background + Context



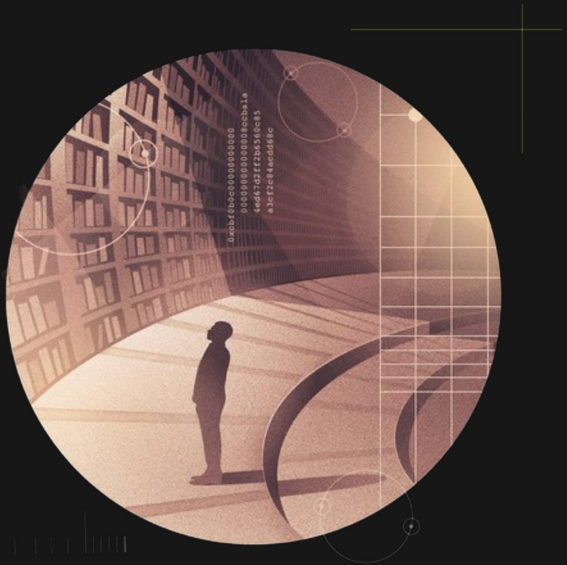
- We (Dedaub) have a long-running public decompiler
  - *Gigahorse*
  - **`app.dedaub.com`**
  - used by many hundreds every day
  - 10,000 registered users
  - algorithms documented in several research publications

    *[ICSE'19, OOPSLA'20, OOPSLA'22, ISSTA'25, ICSE'26]*
  - open source for the most part, source-level proprietary

# Point of the Talk

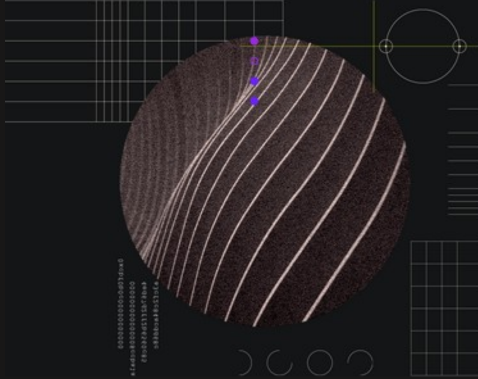


- What is hard about writing an EVM decompiler?
  - how do we do it? Is this the only way?
  - If you already know this stuff:
    - control-flow recovery
    - function recognition
    - memory modeling
    - storage modeling
- Good news: you won't need to understand algorithms!
  - you can do it!
- Bad news: at key points, I need your full attention, so you can understand the problem, from examples



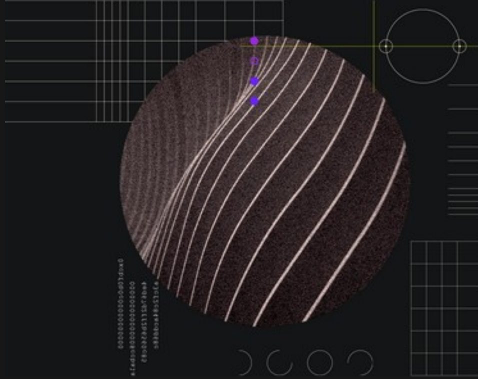
# Decompilation

# The Ethereum Virtual Machine (EVM)



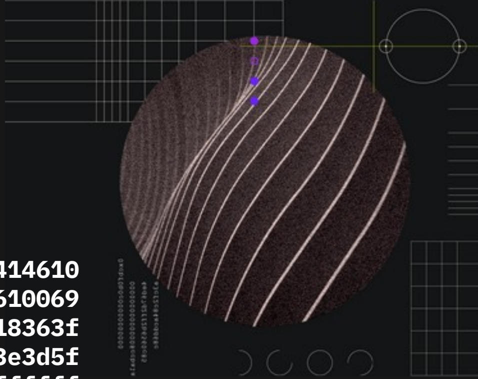
- Stack based:
  - No concept of variables
- Operations on 256 bit integers
  - No type information
- No abstractions of public functions/entry points or private methods
- JUMP targets can be "dynamic"
- Compilers optimize code in order to reduce code size:
  - Reuse of low-level code blocks when possible

# Simple Smart Contract (Solidity)



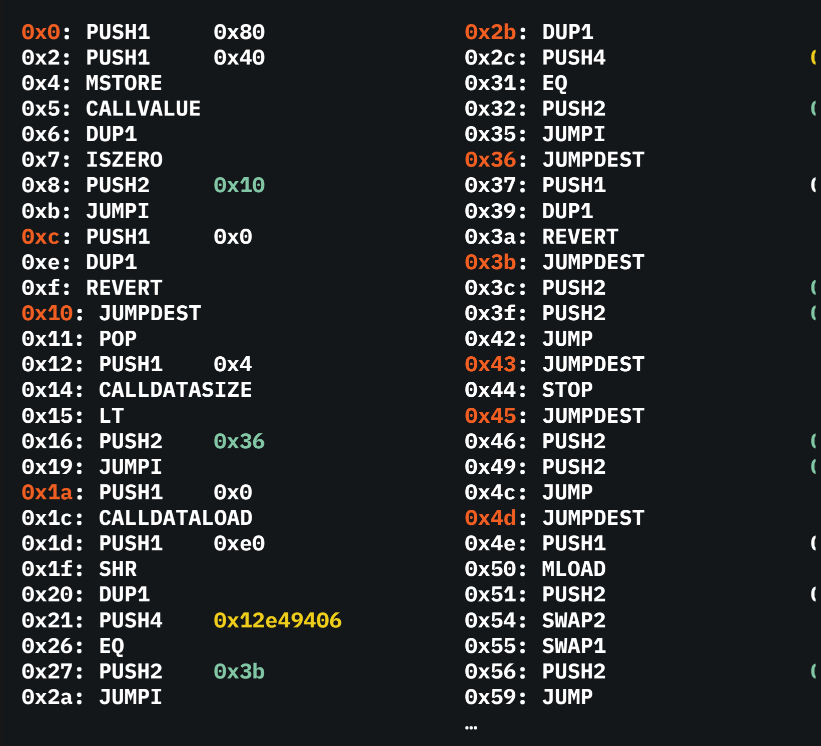
```
interface IERC20
{ function transfer(address to, uint256 value) external returns (bool); }
contract Contract {
  uint256 defaultFee;

  function simpleTransfer(address tok, address to, uint256 amt) external
  { IERC20(tok).transfer(to, amt); }

  function transWFee(address tok, address to, uint256 amt,
                     uint256 feeA, uint256 feeB) external
  { IERC20(tok).transfer(to, amt - defaultFee - feeA - feeB); }
  /* makes 3 private function calls! */
}
```

# Simple Smart Contract (compiled)



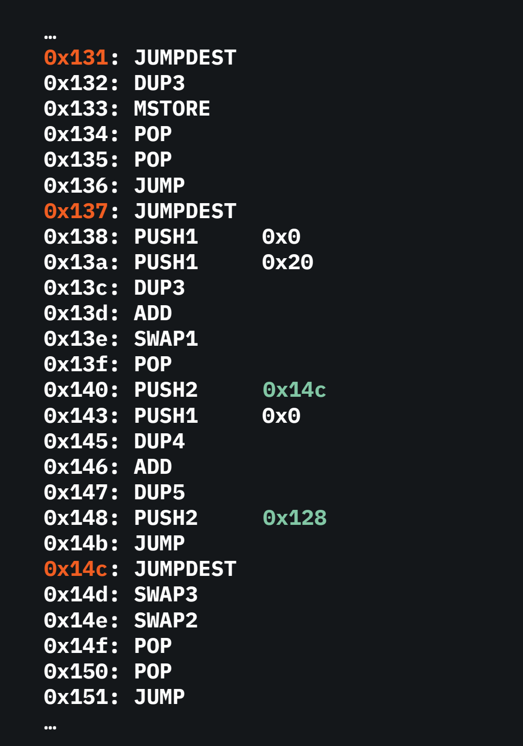
```
608060405234801561000f575f80fd5b5060043610610034575f3560e01c806312e494061461003857806387d7a5f414610
054575b5f80fd5b610052600480360381019061004d9190610227565b610070565b005b61006e600480360381019061006 9
9190610277565b6100f1565b005b8273ffffffffffffffffffffffffffffffffffffffff1663a9059cbb83836040518363f
fffffff1660e01b81526004016100ab92919061030c565b6020604051808303815f875af11580156100c7573d5f803e3d5f
fd5b505050506040513d601f19601f820116820180604052508101906100eb9190610368565b50505050565b8473fffffff
fffffffffffffffffffffffffffffffff1663a9059cbb8583855f548861011d91906103c0565b61012791906103c0565b61
013191906103c0565b6040518363ffffffff1660e01b815260040161014e92919061030c565b6020604051808303815f875
af115801561016a573d5f803e3d5ffd5b505050506040513d601f19601f82011682018060405250810190610 18e91906103
68565b505050505050565b5f80fd5b5f73ffffffffffffffffffffffffffffffffffffffff82169050919050565b5f6101c
38261019a565b9050919050565b6101d3816101b9565b81146101dd575f80fd5b50565b5f813590506101ee816101ca565b
92915050565b5f819050919050565b610206816101f4565b8114610210575f80fd5b50565b5f8135905061022181 6101fd5
65b92915050565b5f805f6060848603121561023e5761023d610196565b5b5f61024b868287016101e0565b935050602061
025c868287016101e0565b925050604061026d86828701610213565b9150509250925092565b5f805f805f60a0868803121
56102905761028f610196565b5b5f61029d888289016101e0565b95505060206102ae888289016101e0565b945050604061
02bf88828901610213565b93505060606102d088828901610213565b92505060806102e188828901610213565b915050929
5509295909350565b6102f7816101b9565b82525050565b610306816101f4565b82525050565b5f60408201905061031f5f
8301856102ee565b61032c60208301846102fd565b9392505050565b5f8115159050919050565b61034781610333565b811
4610351575f80fd5b50565b5f815190506103628161033e565b92915050565b5f6020828403121561037d5761037c610196
565b5b5f61038a848285016103545 65b9150509291505056 5b7f4e487b71000000000000000000000000000000000000000
000000000000000005f52601160045260245ffd5b5f6103ca826101f4565b91506103d5836101f4565b9250828203905081
8111156103ed576103ec610393565b5b9291505056fea2646970667358221220e987f5dd559089603fe79b9c0506b8440a1
a70c32d013a6a038c780a82de25d064736f6c63430008190033
```

# Simple EVM program (Disassembled): 710 loc



```
0x0: PUSH1     0x80
0x2: PUSH1     0x40
0x4: MSTORE
0x5: CALLVALUE
0x6: DUP1
0x7: ISZERO
0x8: PUSH2     0x10
0xb: JUMPI
0xc: PUSH1     0x0
0xe: DUP1
0xf: REVERT
0x10: JUMPDEST
0x11: POP
0x12: PUSH1     0x4
0x14: CALLDATASIZE
0x15: LT
0x16: PUSH2     0x36
0x19: JUMPI
0x1a: PUSH1     0x0
0x1c: CALLDATALOAD
0x1d: PUSH1     0xe0
0x1f: SHR
0x20: DUP1
0x21: PUSH4     0x12e49406
0x26: EQ
0x27: PUSH2     0x3b
0x2a: JUMPI
```

```
0x2b: DUP1
0x2c: PUSH4                 (
0x31: EQ
0x32: PUSH2                 (
0x35: JUMPI
0x36: JUMPDEST
0x37: PUSH1                 (
0x39: DUP1
0x3a: REVERT
0x3b: JUMPDEST
0x3c: PUSH2                 (
0x3f: PUSH2                 (
0x42: JUMP
0x43: JUMPDEST
0x44: STOP
0x45: JUMPDEST
0x46: PUSH2                 (
0x49: PUSH2                 (
0x4c: JUMP
0x4d: JUMPDEST
0x4e: PUSH1                 (
0x50: MLOAD
0x51: PUSH2                 (
0x54: SWAP2
0x55: SWAP1
0x56: PUSH2                 (
0x59: JUMP
…
```



```
…
0x131: JUMPDEST
0x132: DUP3
0x133: MSTORE
0x134: POP
0x135: POP
0x136: JUMP
0x137: JUMPDEST
0x138: PUSH1     0x0
0x13a: PUSH1     0x20
0x13c: DUP3
0x13d: ADD
0x13e: SWAP1
0x13f: POP
0x140: PUSH2     0x14c
0x143: PUSH1     0x0
0x145: DUP4
0x146: ADD
0x147: DUP5
0x148: PUSH2     0x128
0x14b: JUMP
0x14c: JUMPDEST
0x14d: SWAP3
0x14e: SWAP2
0x14f: POP
0x150: POP
0x151: JUMP
…
```

# Simple EVM program (Disassembled): 710 loc

contract entry-point

```
0x0: PUSH1      0x80
0x2: PUSH1      0x40
0x4: MSTORE
0x5: CALLVALUE
0x6: DUP1
0x7: ISZERO
0x8: PUSH2      0x10
0xb: JUMPI
0xc: PUSH1      0x0
0xe: DUP1
0xf: REVERT
0x10: JUMPDEST
0x11: POP
0x12: PUSH1     0x4
0x14: CALLDATASIZE
0x15: LT
0x16: PUSH2     0x36
0x19: JUMPI
0x1a: PUSH1     0x0
0x1c: CALLDATALOAD
0x1d: PUSH1     0xe0
0x1f: SHR
0x20: DUP1
0x21: PUSH4     0x12e49406
0x26: EQ
0x27: PUSH2     0x3b
0x2a: JUMPI
```

```
0x2b: DUP1
0x2c: PUSH4
0x31: EQ
0x32: PUSH2
0x35: JUMPI
0x36: JUMPDEST
0x37: PUSH1
0x39: DUP1
0x3a: REVERT
0x3b: JUMPDEST
0x3c: PUSH2
0x3f: PUSH2
0x42: JUMP
0x43: JUMPDEST
0x44: STOP
0x45: JUMPDEST
0x46: PUSH2
0x49: PUSH2
0x4c: JUMP
0x4d: JUMPDEST
0x4e: PUSH1
0x50: MLOAD
0x51: PUSH2
0x54: SWAP2
0x55: SWAP1
0x56: PUSH2
0x59: JUMP
…
```

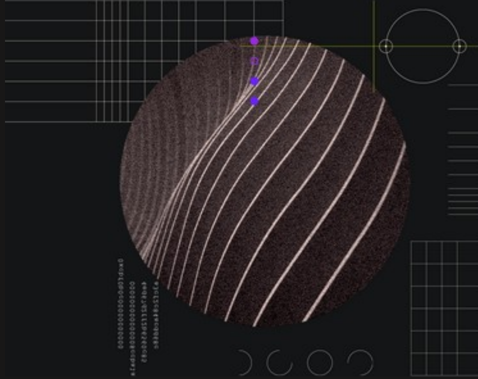
```
…
0x131: JUMPDEST
0x132: DUP3
0x133: MSTORE
0x134: POP
0x135: POP
0x136: JUMP
0x137: JUMPDEST
0x138: PUSH1      0x0
0x13a: PUSH1      0x20
0x13c: DUP3
0x13d: ADD
0x13e: SWAP1
0x13f: POP
0x140: PUSH2      0x14c
0x143: PUSH1      0x0
0x145: DUP4
0x146: ADD
0x147: DUP5
0x148: PUSH2      0x128
0x14b: JUMP
0x14c: JUMPDEST
0x14d: SWAP3
0x14e: SWAP2
0x14f: POP
0x150: POP
0x151: JUMP
…
```

```
0x1a: PUSH1         0x0
0x1c: CALLDATALOAD
0x1d: PUSH1         0xe0
0x1f: SHR
0x20: DUP1
0x21: PUSH4   0x12e49406
0x26: EQ
0x27: PUSH2         0x3b
0x2a: JUMPI
```

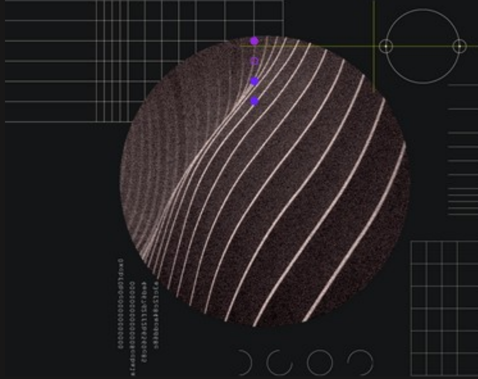
```
0x45: JUMPDEST
0x46: PUSH2  0x4d
0x49: PUSH2 0x104
0x4c: JUMP
```

```
0x14d: SWAP3

0x14e: SWAP2

0x14f: POP

0x150: POP

0x151: JUMP
```

# Decompilation Definition



The recovery of a structured intermediate representation (three-address code) from very low-level bytecode.

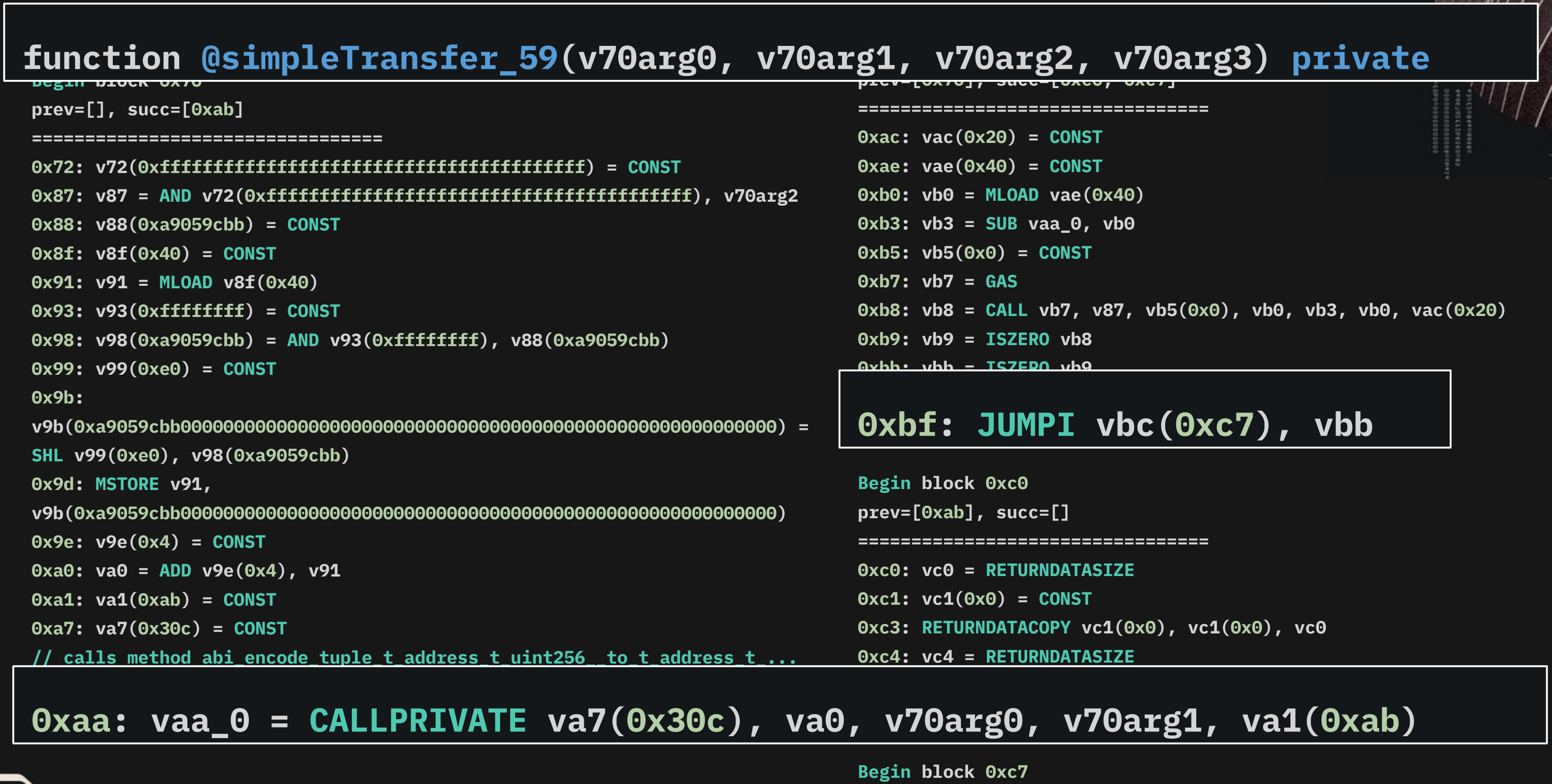


# Gigahorse Three-Address-Code (TAC)

```
function @simpleTransfer_59(v70arg0, v70arg1, v70arg2, v70arg3) private {
Begin block 0x70
prev=[], succ=[0xab]
=================================
0x72: v72(0xffffffffffffffffffffffffffffffffffffffff) = CONST
0x87: v87 = AND v72(0xffffffffffffffffffffffffffffffffffffffff), v70arg2
0x88: v88(0xa9059cbb) = CONST
0x8f: v8f(0x40) = CONST
0x91: v91 = MLOAD v8f(0x40)
0x93: v93(0xffffffff) = CONST
0x98: v98(0xa9059cbb) = AND v93(0xffffffff), v88(0xa9059cbb)
0x99: v99(0xe0) = CONST
0x9b:
v9b(0xa9059cbb00000000000000000000000000000000000000000000000000000000) =
SHL v99(0xe0), v98(0xa9059cbb)
0x9d: MSTORE v91,
v9b(0xa9059cbb00000000000000000000000000000000000000000000000000000000)
0x9e: v9e(0x4) = CONST
0xa0: va0 = ADD v9e(0x4), v91
0xa1: va1(0xab) = CONST
0xa7: va7(0x30c) = CONST
// calls method abi_encode_tuple_t_address_t_uint256__to_t_address_t_...
0xaa: vaa_0 = CALLPRIVATE va7(0x30c), va0, v70arg0, v70arg1, va1(0xab)
```

```
Begin block 0xab
prev=[0x70], succ=[0xc0, 0xc7]
=================================
0xac: vac(0x20) = CONST
0xae: vae(0x40) = CONST
0xb0: vb0 = MLOAD vae(0x40)
0xb3: vb3 = SUB vaa_0, vb0
0xb5: vb5(0x0) = CONST
0xb7: vb7 = GAS
0xb8: vb8 = CALL vb7, v87, vb5(0x0), vb0, vb3, vb0, vac(0x20)
0xb9: vb9 = ISZERO vb8
0xbb: vbb = ISZERO vb9
0xbc: vbc(0xc7) = CONST
0xbf: JUMPI vbc(0xc7), vbb

Begin block 0xc0
prev=[0xab], succ=[]
=================================
0xc0: vc0 = RETURNDATASIZE
0xc1: vc1(0x0) = CONST
0xc3: RETURNDATACOPY vc1(0x0), vc1(0x0), vc0
0xc4: vc4 = RETURNDATASIZE
0xc5: vc5(0x0) = CONST
0xc6: REVERT vc5(0x0), vc4

Begin block 0xc7
```

# Gigahorse Three-Address-Code (TAC)



```
function @simpleTransfer_59(v70arg0, v70arg1, v70arg2, v70arg3) private
```

```
Begin block 0x70
prev=[], succ=[0xab]
=================================
0x72: v72(0xffffffffffffffffffffffffffffffffffffffff) = CONST
0x87: v87 = AND v72(0xffffffffffffffffffffffffffffffffffffffff), v70arg2
0x88: v88(0xa9059cbb) = CONST
0x8f: v8f(0x40) = CONST
0x91: v91 = MLOAD v8f(0x40)
0x93: v93(0xffffffff) = CONST
0x98: v98(0xa9059cbb) = AND v93(0xffffffff), v88(0xa9059cbb)
0x99: v99(0xe0) = CONST
0x9b:
v9b(0xa9059cbb00000000000000000000000000000000000000000000000000000000) =
SHL v99(0xe0), v98(0xa9059cbb)
0x9d: MSTORE v91,
v9b(0xa9059cbb00000000000000000000000000000000000000000000000000000000)
0x9e: v9e(0x4) = CONST
0xa0: va0 = ADD v9e(0x4), v91
0xa1: va1(0xab) = CONST
0xa7: va7(0x30c) = CONST
// calls method abi_encode_tuple_t_address_t_uint256__to_t_address_t_...
```

```
0xaa: vaa_0 = CALLPRIVATE va7(0x30c), va0, v70arg0, v70arg1, va1(0xab)
```

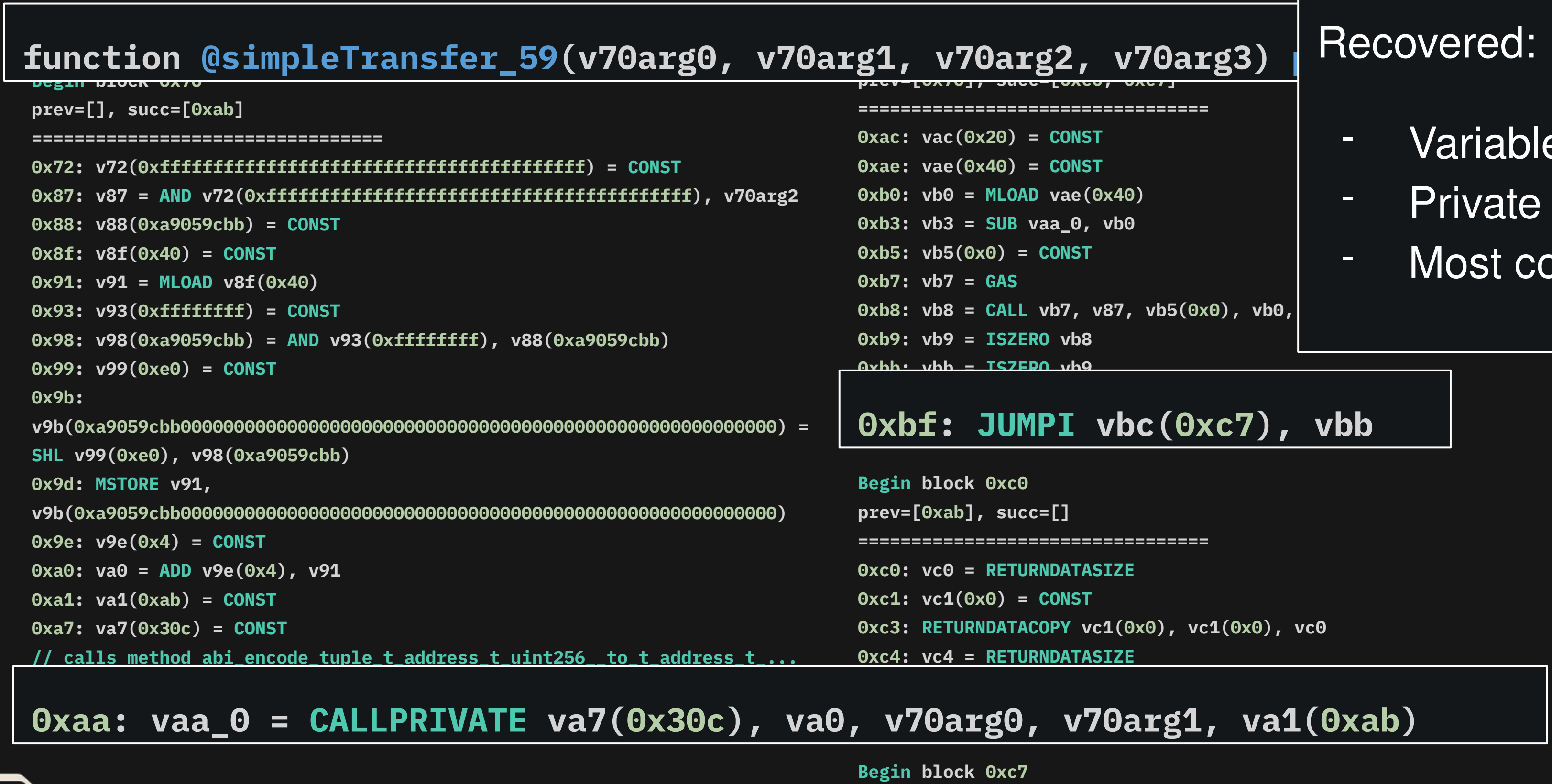
```
prev=[0x70], succ=[0xc0, 0xc7]
=================================
0xac: vac(0x20) = CONST
0xae: vae(0x40) = CONST
0xb0: vb0 = MLOAD vae(0x40)
0xb3: vb3 = SUB vaa_0, vb0
0xb5: vb5(0x0) = CONST
0xb7: vb7 = GAS
0xb8: vb8 = CALL vb7, v87, vb5(0x0), vb0, vb3, vb0, vac(0x20)
0xb9: vb9 = ISZERO vb8
0xbb: vbb = ISZERO vb9
```

```
0xbf: JUMPI vbc(0xc7), vbb
```

```
Begin block 0xc0
prev=[0xab], succ=[]
=================================
0xc0: vc0 = RETURNDATASIZE
0xc1: vc1(0x0) = CONST
0xc3: RETURNDATACOPY vc1(0x0), vc1(0x0), vc0
0xc4: vc4 = RETURNDATASIZE
```

```
Begin block 0xc7
```

# Gigahorse Three-Address-Code (TAC)



```
function @simpleTransfer_59(v70arg0, v70arg1, v70arg2, v70arg3)
```

```
Begin block 0x70
prev=[], succ=[0xab]
=================================
0x72: v72(0xffffffffffffffffffffffffffffffffffffffff) = CONST
0x87: v87 = AND v72(0xffffffffffffffffffffffffffffffffffffffff), v70arg2
0x88: v88(0xa9059cbb) = CONST
0x8f: v8f(0x40) = CONST
0x91: v91 = MLOAD v8f(0x40)
0x93: v93(0xffffffff) = CONST
0x98: v98(0xa9059cbb) = AND v93(0xffffffff), v88(0xa9059cbb)
0x99: v99(0xe0) = CONST
0x9b:
v9b(0xa9059cbb00000000000000000000000000000000000000000000000000000000) =
SHL v99(0xe0), v98(0xa9059cbb)
0x9d: MSTORE v91,
v9b(0xa9059cbb00000000000000000000000000000000000000000000000000000000)
0x9e: v9e(0x4) = CONST
0xa0: va0 = ADD v9e(0x4), v91
0xa1: va1(0xab) = CONST
0xa7: va7(0x30c) = CONST
// calls method abi_encode_tuple_t_address_t_uint256__to_t_address_t_...
```

```
0xaa: vaa_0 = CALLPRIVATE va7(0x30c), va0, v70arg0, v70arg1, va1(0xab)
```

```
prev=[0x70], succ=[0xc0, 0xc7]
=================================
0xac: vac(0x20) = CONST
0xae: vae(0x40) = CONST
0xb0: vb0 = MLOAD vae(0x40)
0xb3: vb3 = SUB vaa_0, vb0
0xb5: vb5(0x0) = CONST
0xb7: vb7 = GAS
0xb8: vb8 = CALL vb7, v87, vb5(0x0), vb0,
0xb9: vb9 = ISZERO vb8
0xbb: vbb = ISZERO vb9
```

```
0xbf: JUMPI vbc(0xc7), vbb
```

```
Begin block 0xc0
prev=[0xab], succ=[]
=================================
0xc0: vc0 = RETURNDATASIZE
0xc1: vc1(0x0) = CONST
0xc3: RETURNDATACOPY vc1(0x0), vc1(0x0), vc0
0xc4: vc4 = RETURNDATASIZE
```

```
Begin block 0xc7
```

Recovered:

- Variables
- Private functions
- Most control flow

# Difficulties #1, #2: Control Flow + Functions

# Simple EVM program (Disassembled): 710 loc

contract entry-point

```
0x0: PUSH1     0x80
0x2: PUSH1     0x40
0x4: MSTORE
0x5: CALLVALUE
0x6: DUP1
0x7: ISZERO
0x8: PUSH2     0x10
0xb: JUMPI
0xc: PUSH1     0x0
0xe: DUP1
0xf: REVERT
0x10: JUMPDEST
0x11: POP
0x12: PUSH1     0x4
0x14: CALLDATASIZE
0x15: LT
0x16: PUSH2     0x36
0x19: JUMPI
0x1a: PUSH1     0x0
0x1c: CALLDATALOAD
0x1d: PUSH1     0xe0
0x1f: SHR
0x20: DUP1
0x21: PUSH4     0x12e49406
0x26: EQ
0x27: PUSH2     0x3b
0x2a: JUMPI
```

```
0x2b: DUP1
0x2c: PUSH4
0x31: EQ
0x32: PUSH2
0x35: JUMPI
0x36: JUMPDEST
0x37: PUSH1
0x39: DUP1
0x3a: REVERT
0x3b: JUMPDEST
0x3c: PUSH2
0x3f: PUSH2
0x42: JUMP
0x43: JUMPDEST
0x44: STOP
0x45: JUMPDEST
0x46: PUSH2
0x49: PUSH2
0x4c: JUMP
0x4d: JUMPDEST
0x4e: PUSH1
0x50: MLOAD
0x51: PUSH2
0x54: SWAP2
0x55: SWAP1
0x56: PUSH2
0x59: JUMP
…
```

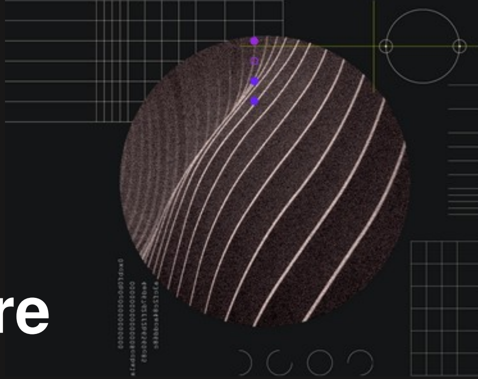
```
…
0x131: JUMPDEST
0x132: DUP3
0x133: MSTORE
0x134: POP
0x135: POP
0x136: JUMP
0x137: JUMPDEST
0x138: PUSH1     0x0
0x13a: PUSH1     0x20
0x13c: DUP3
0x13d: ADD
0x13e: SWAP1
0x13f: POP
0x140: PUSH2     0x14c
0x143: PUSH1     0x0
0x145: DUP4
0x146: ADD
0x147: DUP5
0x148: PUSH2     0x128
0x14b: JUMP
0x14c: JUMPDEST
0x14d: SWAP3
0x14e: SWAP2
0x14f: POP
0x150: POP
0x151: JUMP
…
```

```
0x1a:  PUSH1          0x0
0x1c:  CALLDATALOAD
0x1d:  PUSH1          0xe0
0x1f:  SHR
0x20:  DUP1
0x21:  PUSH4   0x12e49406
0x26:  EQ
0x27:  PUSH2          0x3b
0x2a:  JUMPI
```

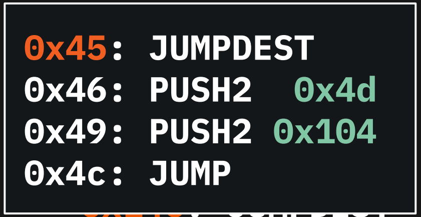
```
0x45:  JUMPDEST
0x46:  PUSH2   0x4d
0x49:  PUSH2  0x104
0x4c:  JUMP
```

```
0x14d:  SWAP3

0x14e:  SWAP2

0x14f:  POP

0x150:  POP

0x151:  JUMP
```

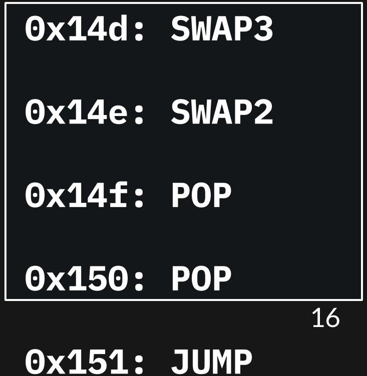
# Heart of the Problem



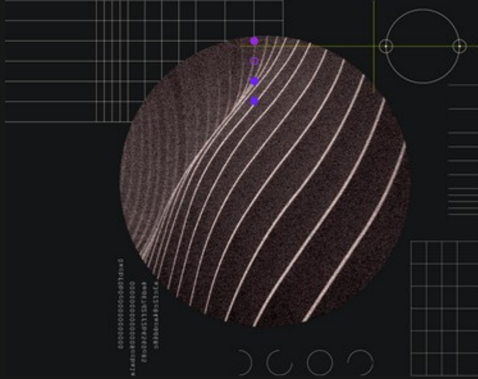
- **Execution of the code determines structure of the code!**
- Need work to tell if a **`JUMP`** is:
  - **`if`**
  - **`for`**
  - function call
  - return
- Secondarily: *does the top of the stack at this instruction contain a jump label, or a run-time value?*



```
0x45: JUMPDEST
0x46: PUSH2  0x4d
0x49: PUSH2 0x104
0x4c: JUMP
```

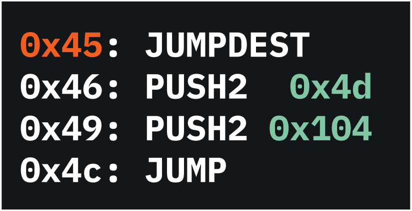


```
0x14d: SWAP3

0x14e: SWAP2

0x14f: POP

0x150: POP

0x151: JUMP
```

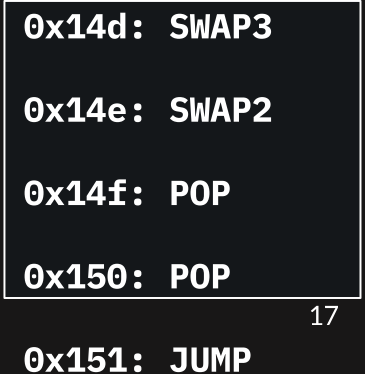


# "Obvious" solution

- Execution of the code determines structure of the code?
- **Just execute the code to discover its structure!**
  - several decompilers have tried to do just that: *symbolic execution*
    - Porosity
    - Panoramix
    - Heimdall-rs
  - *that's not what we do*

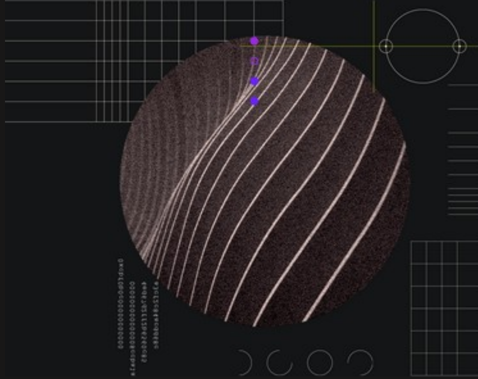


```
0x45: JUMPDEST
0x46: PUSH2  0x4d
0x49: PUSH2 0x104
0x4c: JUMP
```



```
0x14d: SWAP3

0x14e: SWAP2

0x14f: POP

0x150: POP

0x151: JUMP
```

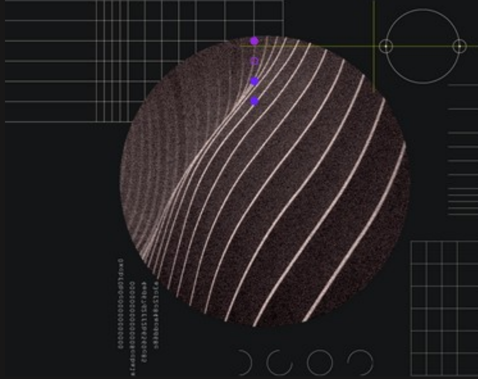
# Comparison to Heimdall-rs: Completeness



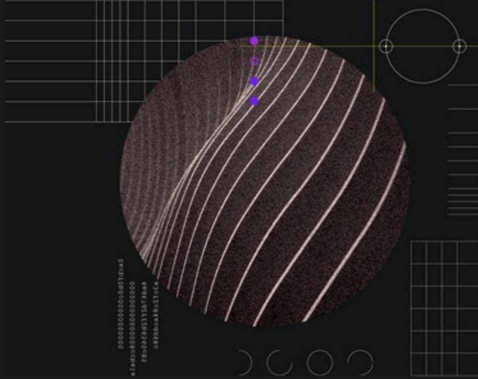
Yul Dataset

| | Unique External Calls | Unique Events |
|---|---|---|
| Gigahorse | 13600 ➕ | 13661 ➕ |
| Heimdall-rs | 9841 ➖ | 9505 ➖ |

# Problem with "Just Execute the Code"



- Code has exponential (or infinite) number of paths
    - how can you cover all?
    - only need to simulate the parts of the code that involve jump labels, not all values!
- Compiler has performed very complex optimizations, merged code from far-away source positions



# Problem with "Just Execute the Code"

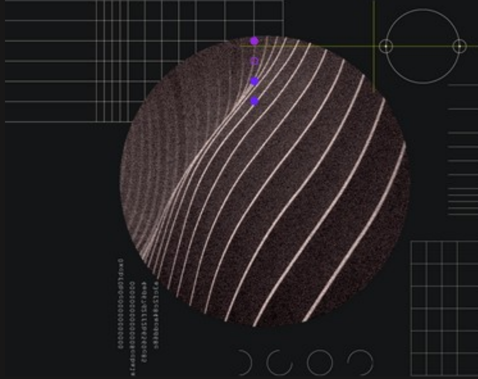
- Code has exponential (or infinite) number of paths
  - how can you cover all?
  - only need to simulate the parts of the code that involve jump labels, not all values!
- Compiler has performed very complex optimizations, merged code from far-away source positions

*- is that a label at the top of the stack?*

*- depends!*

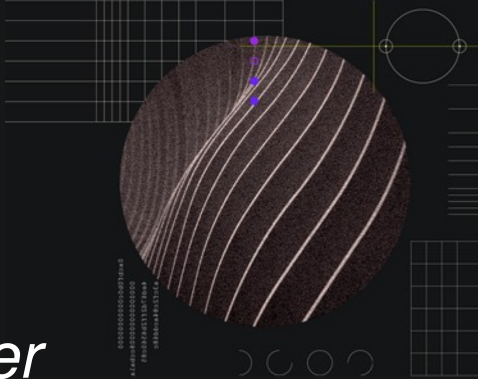
```
0x14c: JUMPDEST

0x14d: SWAP3

0x14e: SWAP2

0x14f: POP

0x150: POP

0x151: JUMP
```

# What We Do Instead: *Static Analysis*



- Approximation of what the code does for all possible executions
  - how to distinguish ones that do things differently?
    - "*context sensitivity*"
    - classify executions into a finite (but large) number of bins
  - use the same algorithm to also tell us which parts of the code involve jump labels positions

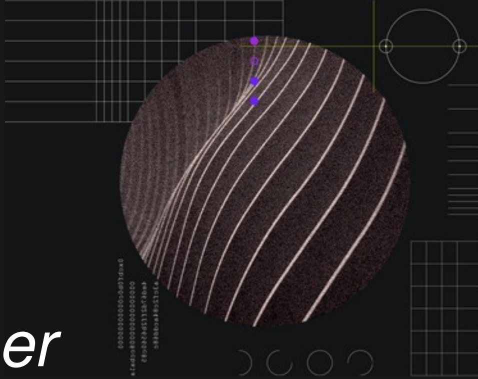
# Our Algorithm *(slightly simplified )*



*When calling a function do we need to remember where it was called from?*

Yes.

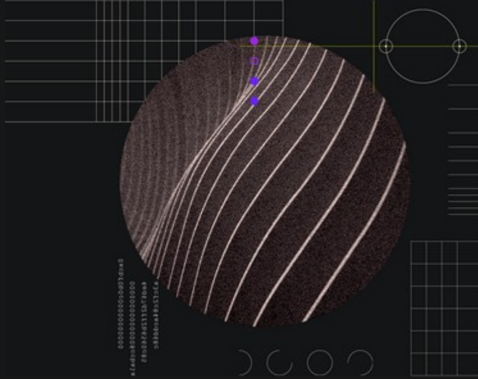
# Our Algorithm *(slightly simplified )*



*When a function returns do we need to remember it was ever called?*

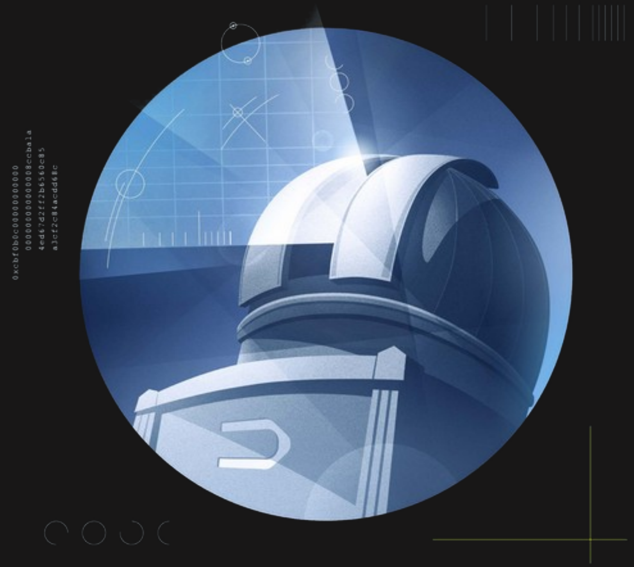
No.

# Similar Complications: Recognizing (Private) Functions



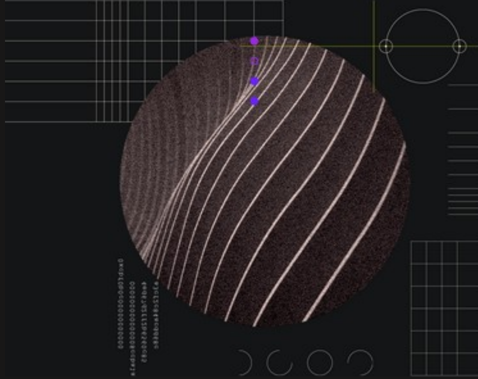
- How many arguments are passed in this call? Of what types? How many values are returned?
  - maybe it depends?
  - some call site may be ignoring some returns?
  - EVM has no well-structured stack
- Need again complex algorithms

```
0x45: JUMPDEST
0x46: PUSH2  0x4d
0x49: PUSH2 0x104
0x4c: JUMP
```

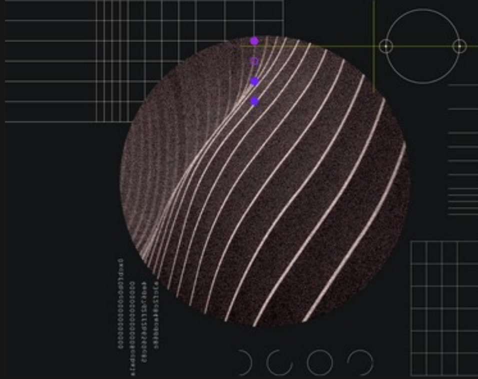


# Difficulties #3, #4: Memory Modeling, Storage Modeling

# EVM Data Stores, through Solidity



- Two kinds of stores:
  - **Storage:** Persistent, kept on the blockchain’s state
  - **Memory:** Volatile, per transaction
- Memory is used by the compiler as a scratchpad for all computations of data with undetermined length
  - arrays, strings



# Solidity Smart Contract Example

```
contract Example{
    string onStorage;

    function setIt(string memory newStr) public {
        onStorage = newStr;
    }

    function getHash() public view returns (bytes32) {
        return keccak256(onStorage);
    }
}
```

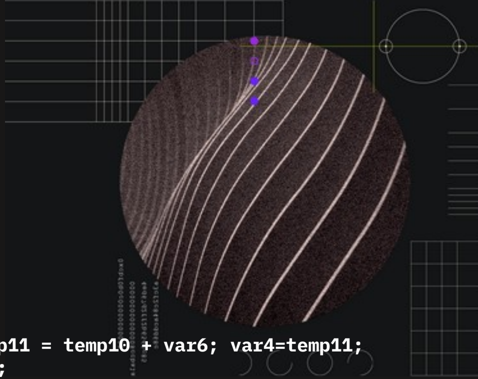
On persistent storage

Argument passed through memory

Return passed through memory

Keccak hashes memory contents

# “Memory” hides a lot of implicit computation!



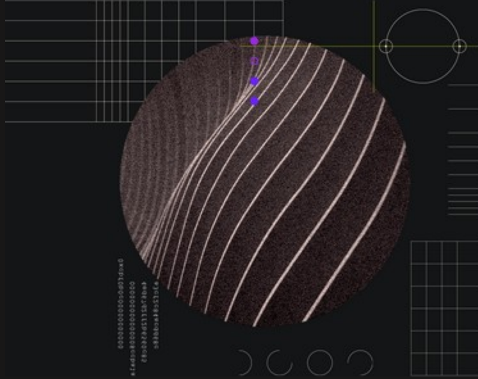
```
mapping(string => string) mTokens; …
function getToken(string pDocumentHash) returns(string)
{ return mTokens[pDocumentHash]; }
```

becomes:

```
function getToken(var arg0) returns (var r0) {
  var var0 = 0x053b;  var0 = func_06C6();  var var1 = 0x02;
  var temp0 = arg0;  var var2 = temp0;
  var var3 = memory[0x40:0x60];  var var4 = var3;
  var var5 = var2 + 0x20;  var var6 = memory[var2:var2 + 0x20];
  var var7 = var6;  var var8 = var4;  var var9 = var5;
  if (var7 < 0x20) {
  label_0573:
    var temp1 = 0x0100 ** (0x20 - var7) - 0x01;  var temp2 = var8;
    memory[temp2:temp2 + 0x20] = (memory[var9:var9 + 0x20] &
~temp1) | (memory[temp2:temp2 + 0x20] & temp1);
    var temp3 = var6 + var4;
    memory[temp3:temp3 + 0x20] = var1;
    var temp4 = memory[0x40:0x60];
    var temp5 = keccak256(memory[temp4:temp4+(temp3+0x20)-temp4]);
    var temp6 = storage[temp5];
    var temp7 = (!(temp6 & 0x01) * 0x0100 - 0x01 & temp6) / 0x02;
    var temp8 = memory[0x40:0x60];
    memory[0x40:0x60] = temp8 + (temp7+0x1f) / 0x20 * 0x20 + 0x20;
    var1 = temp8;  var2 = temp5;  var3 = temp7;
    memory[var1:var1 + 0x20] = var3;
    var4 = var1 + 0x20;  var5 = var2;
    var temp9 = storage[var5];
    var6 = (!(temp9 & 0x01) * 0x0100 - 0x01 & temp9) / 0x02;
    if (!var6) {
    label_063A:
      return var1;
    } else if (0x1f < var6) {
      var temp10=var4; var temp11 = temp10 + var6; var4=temp11;
      memory[0x00:0x20] = var5;
      var temp12 = keccak256(memory[0x00:0x20]);
      memory[temp10:temp10 + 0x20] = storage[temp12];
      var5 = temp12 + 0x01;  var6 = temp10 + 0x20;
      if (var4 <= var6) { goto label_0631; }
    label_061D:
      var temp13 = var5;  var temp14 = var6;
      memory[temp14:temp14 + 0x20] = storage[temp13];
      var5 = temp13 + 0x01;  var6 = temp14 + 0x20;
      if (var4 > var6) { goto label_061D; }
    label_0631:
      var temp15 = var4; var temp16 = temp15+(var6 - temp15&0x1f);
      var6 = temp15;  var4 = temp16;
      goto label_063A;
    } else {
      var temp17 = var4;
      memory[temp17:temp17+0x20] = storage[var5]/0x0100 * 0x0100;
      var4 = temp17 + 0x20;  var6 = var6;
      goto label_063A;
    }
  } else {
  label_0559:
    var temp18 = var9;  var temp19 = var8;
    memory[temp19:temp19 + 0x20] = memory[temp18:temp18 + 0x20];
    var8 = temp19 + 0x20; var9 = temp18 + 0x20; var7 = var7-0x20;
    if (var7 < 0x20) { goto label_0573; }
    else { goto label_0559; }
  }
}
```

*So, You Think You Can Write an EVM Decompiler?*

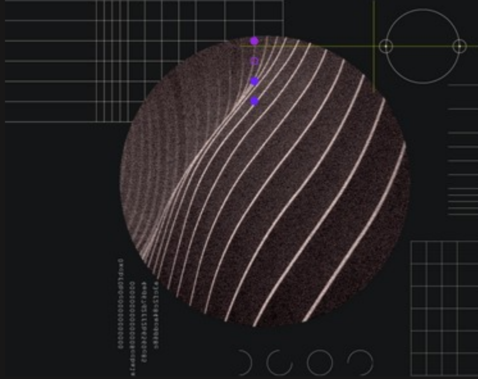
# Memory Modeling: A Pretty Hard Problem



- Need to truly reverse engineer fairly arbitrary pointer computation
  - Solidity *free memory pointer*
  - and getting worse

# Status of Decompilation

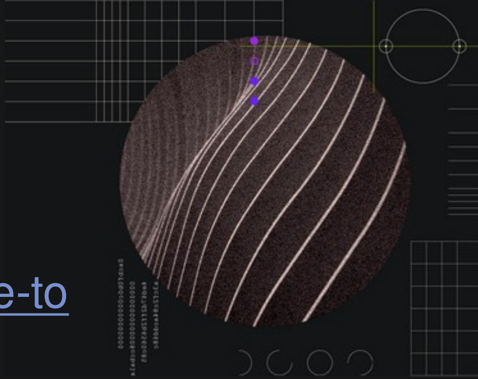
# Parting Thoughts: Are the Tough Problems Solved?



*Disclaimer*: Subjective

- *Difficulty #1*: Control Flow
  - **99%**, you may want to do it differently
- *Difficulty #2*: Function Recovery
  - **95%**
- *Difficulty #3*: Memory Modeling
  - **90%**, may never be 100%
- *Difficulty #4*: Storage Modeling
  - **99+%**: you probably want to imitate what we do *[ICSE'26]*

# Dedaub Gigahorse 3.0



Open source, available at: https://github.com/nevillegrech/gigahorse-toolchain

Deployed on all deployed contracts on many blockchains at: https://app.dedaub.com/