May 6-7, 2021

ARSENAL

#BHASIA @BLACKHATEVENTS

black hat ASIA 2021
May 6-7, 2021
ARSENAL

Ghidra-EVM: Reversing Smart Contracts with Ghidra

Antonio de la Piedra

Motivation

- Importance of validating the security of deployed contracts
- Push the limits of Ghidra for analyzing EVM byte code

Agenda

Introduction

Ethereum and the EVM
Ghidra extension capabilities

Ghidra-EVM Architecture

Related work
Design of ghidra-EVM
Modules and dependencies
Challenges and limitations

Demo

Recovering the CFG
Analyzing creation bytecode
Solving a challenge

Introduction

Ethereum

- The Ethereum blockchain is the second largest blockchain after bitcoin.
- Support of smart contracts, executable code located at an address of the blockchain that is triggered through a transaction.
- Deployed contracts cannot be changed and updates are usually complex.
- In the past, a considerable amount of funds was stolen due to
- programming errors, such as the DAO attack, or the King of the Ether Throne attack.

Smart contracts

- It is possible to deploy smart contracts at an address in the Ethereum network.
- It provides different functions that are triggered through a transaction in the blockchain
- The contracts are generally written in languages such as solidity although in the last few years security-oriented constrained languages have been proposed such as LLL and Vyper.

The Ethereum Virtual Machine (EVM)

- Compiled smart contracts into bytecode are executed in the Ethereum Virtual Machine (EVM).
- Stack machine, that means that typically each instruction pops the operands from the stack and pushed back the result after the instruction execution.
- The machine works with 256-bit words and utilized two memory units: "memory" and storage.

Ghidra

- In 2019, the National Security Agency released the Ghidra reverse engineering tool, equipped with a decompiler and plugin support.

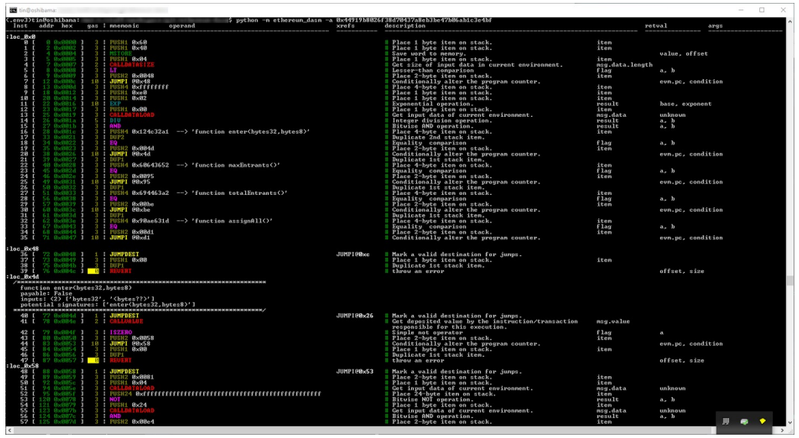
Ghidra Extension Capabilities

- **Processor module**: where the different instructions of a specific processor are described in the SLEIGH language.
- **Loader**: detects the object code that is loaded into Ghidra
- and relates it to a certain processor and compiler configuration.
- **Plugins**: provides access to the Ghidra API for modifying memory blocks, adding labels, modifying the object code, locating instructions and performing analysis on the disassembled code: Java, Python 2 and Python 3 (via ghidra-bridge).

Architecture of Ghidra-EVM

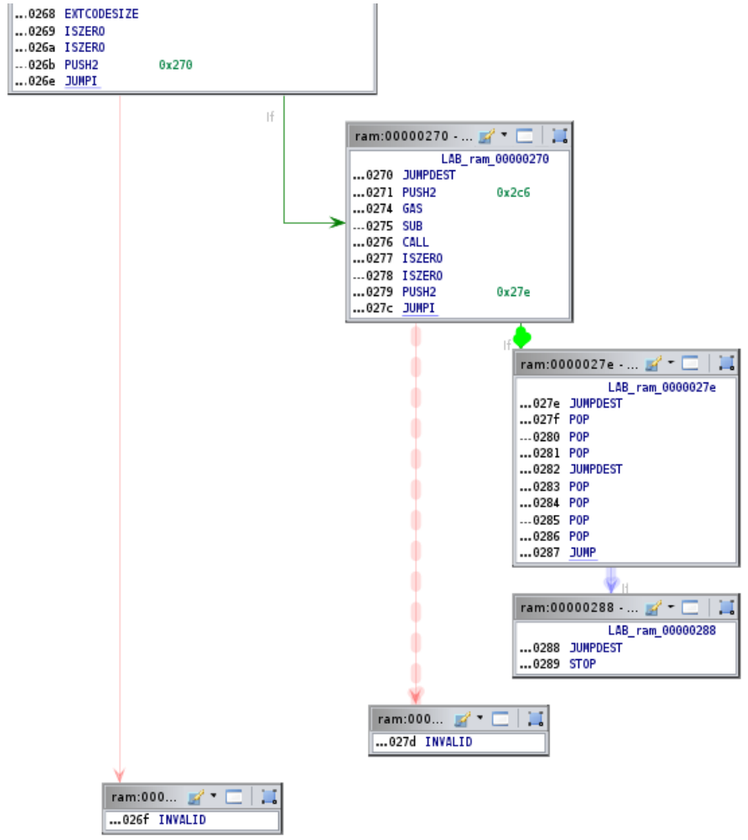
Related work

- **Recovering the CFG**: ethereum-graph-debugger, Consensys surya (solidity) and evm cfg builder: retrieves the CFG of a compiled contract in EVM bytecode.
- **Disassemble plugins for EVM byte code**: IDA Pro and Binary Ninja.
- **Binary analysis frameworks**: Rattle, ethereum-dasm, pyevmasm (library).



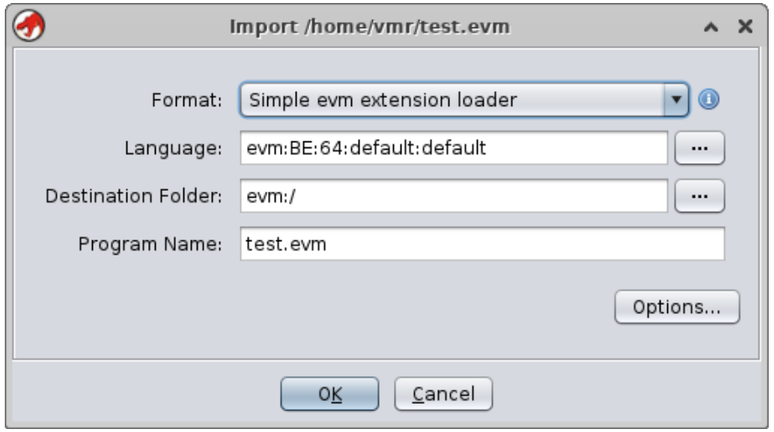
Related work

- On Nov 6, 2019 Crytic opened a bounty for building a Ghidra plugin that, using evm cfg builder, could retrieve a show the CFG of a smart contract in Ghidra.
- However, this bounty was cancelled months after without a proposed solution (https://github.com/crytic/evm_cfg_builder/issues/23).
- This work, inspired by the cancelled bounty, proposes an architecture to reverse engineering smart contracts in Ghidra.



Design of Ghidra-EVM: Loader

- Ghidra-EVM expects an hexadecimal string representing bytecode or a binary file with bytecode:
  - Solc's output
  - Downloaded from the blockchain
- The loader expects a file with the extension .evm or .evm_h.



Design of Ghidra-EVM: Processor Module

- Described in 5 different files:
  - **Cspec**: Compiler specification.
  - **Ldefs**: Processor identification and language definitions.
  - **Opinion**: links the loader to the compiler definition in ldefs.
  - **Psec**: Defines the size of the memory blocks.
  - **Slaspec**: Register definitions and instructions:

```
define space register   type=register_space  size=$(SIZE) wordsize=1;
define register offset=0x00 size=$(SIZE) [sp pc];
```

Design of Ghidra-EVM: Memory model

- In the EVM, a smart contract relies on three main memory structures:
  - **Storage memory**: of 256 bits per slot, utilized by data structures in the smart contract defined as storage.
  - **Volatile memory**: utilized for storing computations, and data structures
  - defined as memory and the freepointer data structure.
  - **Stack**: utilized by the majority of instructions since the arguments
  - are typically pushed and poped to/from the stack.

Design of Ghidra-EVM: Memory model

- The storage and volatile memories are defined in Ghidra-EVM to aid in the decompilation process.

```
    <default_memory_blocks>
        <memory_block name="evm_storage" start_address="evm_storage:0x00000000"
length="0xffffffff" mode="rw" initialized="true"/>
        <memory_block name="evm_jump_table" start_address="evm_jump_table:0x00000000"
length="0xffffffff" mode="r" initialized="true"/>
        <memory_block name="evm_ram" start_address="evm_ram:0x00000000" length="0xffffffff"
mode="rw" initialized="true"/>
    </default_memory_blocks>
```

- The third memory block, `evm_jump_table` that is used to generate, the Control Flow Graph (CFG) and to resolve the jump destinations of the JUMP and JUMPI instructions.

Design of Ghidra-EVM: Instruction definition

- Around 140 opcodes in the Ethereum yellow paper.
- Defined in SLEIGH in the slaspec file e.g.

```
:AND is op=0x16 {

         _value1 :$(SIZE) = 0;
         _value2 :$(SIZE) = 0;
         _value3 :$(SIZE) = 0;

         pop(_value1);
         pop(_value2);

        _value3 = _value1 & _value2;

         push(_value3);
 }
```

Design of Ghidra-EVM: Instruction definition

- Instructions that obtain data from the blockchain are defined as pcodeop e.g.

```
define pcodeop get_block_gas_limit;
:GASLIMIT is op=0x45 {
        _value1 :$(SIZE) = 0;
        _value1 = get_block_gas_limit();
        push(_value1);
}
```

Design of Ghidra-EVM: CFG and branches

- evm_cfg_builder is used to recover the CFG
  - Python 3 code: ghidra_bridge is used (RPC proxy for Python objects)
  - A jump table is created during analysis: contains the destination address of every jump in the disassembled code
- The SLEIGH definitions of JUMP and JUMPI recover the JUMPDEST
- location via the jump table.
- Through evm_cfg_builder contract methods are defined as functions in Ghidra, method names and properties are extracted and showed in the disassembler.

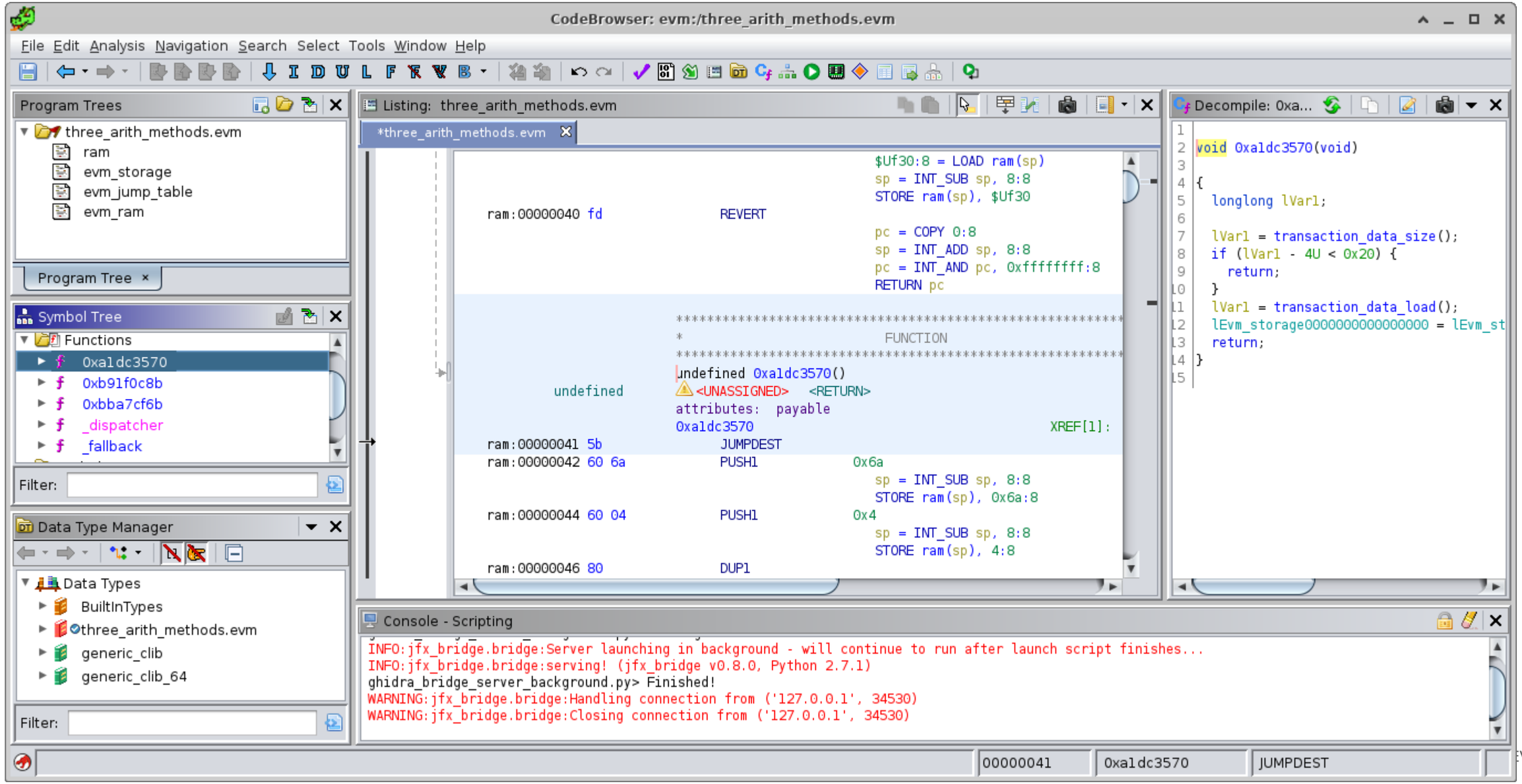
Tilix: vmr@vmr-ThinkPad-T430:~/work/ghidra-evm/helper

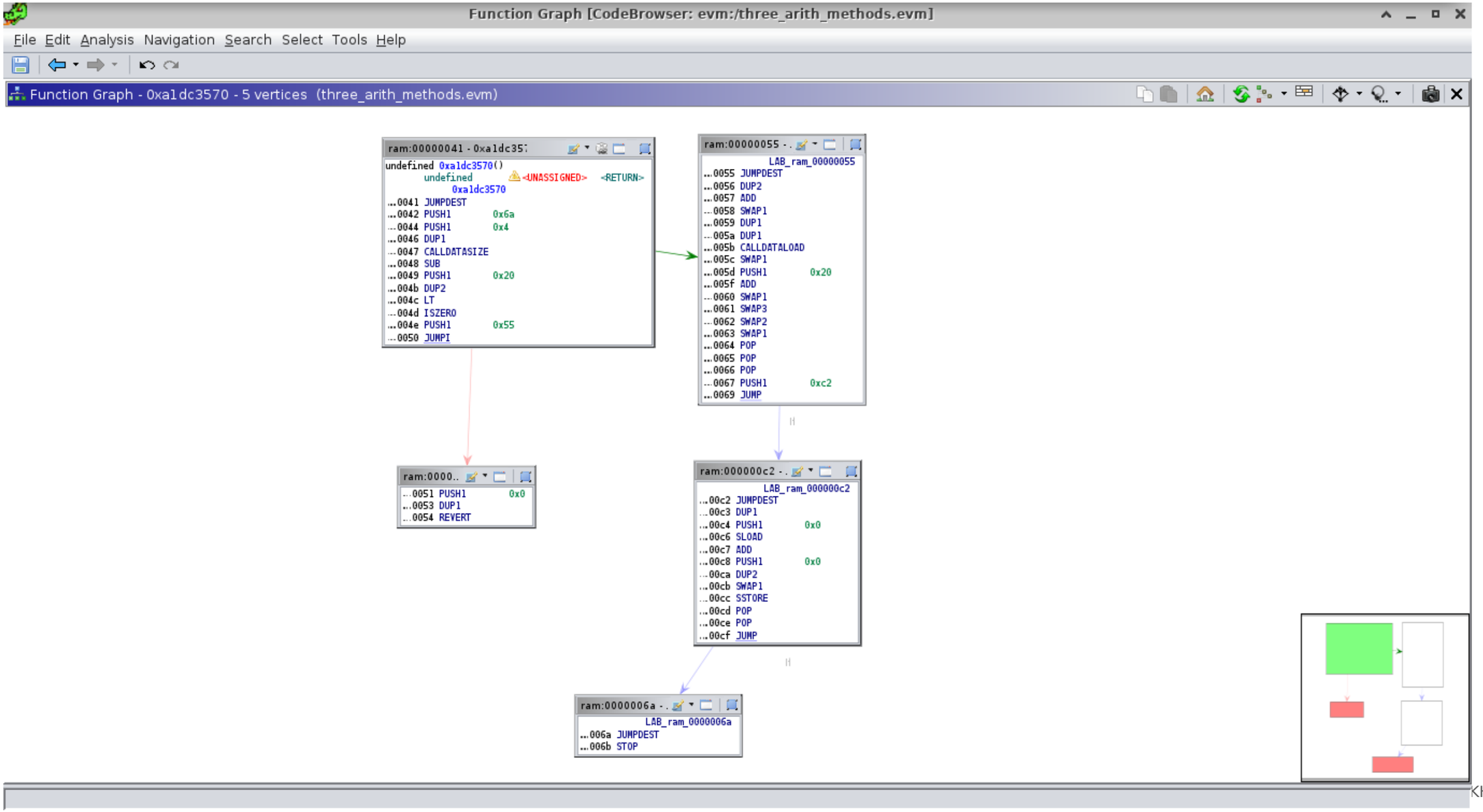
```
✗ ~/work/ghidra-evm/helper  main  python3 evm_helper.py ../examples/three_arith_methods.evm
       _     _     _
  __ _| |__ (_) __| |_ __ __ _        _____   ___ __ ___
 / _` | '_ \| |/ _` | '__/ _` |_____ / _ \ \ / / '_ ` _ \
| (_| | | | | | (_| | | | (_| |_____|  __/\ V /| | | | | |
 \__, |_| |_|_|\__,_|_|  \__,_|      \___| \_/ |_| |_| |_| v.0.1
 |___/

[*] Parsing bytecode...
b'6080604052348015600f57600080fd5b5060043610603c5760003560e01c8063a1dc3570146041578063b91f0c8b14606c578063bba7cf6b146097575b600080fd5b606a6004803603602
0811015605557600080fd5b810190808035906020019092919050505060c2565b005b6095600480360360208110156080576000 80fd5b810190808035906020019092919050505060d0565b
005b60c0600480360360208110 1560ab57600080fd5b810190808035906020019092919050505060de565b005b806000540160008190555050565b806000540260008190555050565b80600
054016000819055505056fea2646970667358221220f5f3ce908bd58858c580bf943a338b2b60c5db54ce39d09e6e166cfba87bc11264736f6c63430007030033'
[*] Setting analysis options....
[*] Creating CFG...
[*] Resolving jumps...
<cfg BasicBlock@18-27>
Finishes at:  0x27
        JUMPI to:  0x41
<cfg BasicBlock@97-a6>
Finishes at:  0xa6
        JUMPI to:  0xab
<cfg BasicBlock@0-a>
Finishes at:  0xa
        JUMPI to:  0xf
<cfg BasicBlock@de-eb>
Finishes at:  0xeb
        JUMP to: 0xc0
<cfg BasicBlock@32-3b>
Finishes at:  0x3b
        JUMPI to:  0x97
<cfg BasicBlock@55-69>
Finishes at:  0x69
        JUMP to: 0xc2
<cfg BasicBlock@6c-7b>
Finishes at:  0x7b
        JUMPI to:  0x80
<cfg BasicBlock@80-94>
Finishes at:  0x94
        JUMP to: 0xd0
<cfg BasicBlock@ab-bf>
```

Tilix: vmr@vmr-ThinkPad-T430:~/work/ghidra-evm/helper

```
        JUMP to: 0xc0
<cfg BasicBlock@32-3b>
Finishes at:  0x3b
        JUMPI to:  0x97
<cfg BasicBlock@55-69>
Finishes at:  0x69
        JUMP to: 0xc2
<cfg BasicBlock@6c-7b>
Finishes at:  0x7b
        JUMPI to:  0x80
<cfg BasicBlock@80-94>
Finishes at:  0x94
        JUMP to: 0xd0
<cfg BasicBlock@ab-bf>
Finishes at:  0xbf
        JUMP to: 0xde
<cfg BasicBlock@c2-cf>
Finishes at:  0xcf
        JUMP to: 0x6a
<cfg BasicBlock@28-31>
Finishes at:  0x31
        JUMPI to:  0x6c
<cfg BasicBlock@d0-dd>
Finishes at:  0xdd
        JUMP to: 0x95
<cfg BasicBlock@41-50>
Finishes at:  0x50
        JUMPI to:  0x55
<cfg BasicBlock@f-17>
Finishes at:  0x17
        JUMPI to:  0x3c
[*] Exploring functions...
        Found function _dispatcher
        Found function _fallback
        Found function 0xa1dc3570
        Found function 0xb91f0c8b
        Found function 0xbba7cf6b
[*] Analyzing....
[*] Disassemble all....
~/work/ghidra-evm/helper  main
```


CodeBrowser: evm:/three_arith_methods.evm
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
three_arith_methods.evm
ram
evm_storage
evm_jump_table
evm_ram
Program Tree
Symbol Tree
Functions
0xa1dc3570
0xb91f0c8b
0xbba7cf6b
_dispatcher
_fallback
Filter:
Data Type Manager
Data Types
BuiltInTypes
three_arith_methods.evm
generic_clib
generic_clib_64
Filter:
Listing: three_arith_methods.evm
*three_arith_methods.evm
$Uf30:8 = LOAD ram(sp)
sp = INT_SUB sp, 8:8
STORE ram(sp), $Uf30
ram:00000040 fd REVERT
pc = COPY 0:8
sp = INT_ADD sp, 8:8
pc = INT_AND pc, 0xffffffff:8
RETURN pc
FUNCTION
undefined 0xa1dc3570()
undefined <UNASSIGNED> <RETURN>
attributes: payable
0xa1dc3570 XREF[1]:
ram:00000041 5b JUMPDEST
ram:00000042 60 6a PUSH1 0x6a
sp = INT_SUB sp, 8:8
STORE ram(sp), 0x6a:8
ram:00000044 60 04 PUSH1 0x4
sp = INT_SUB sp, 8:8
STORE ram(sp), 4:8
ram:00000046 80 DUP1
Decompile: 0xa...
void 0xa1dc3570(void)
{
longlong lVar1;
lVar1 = transaction_data_size();
if (lVar1 - 4U < 0x20) {
return;
}
lVar1 = transaction_data_load();
lEvm_storage0000000000000000 = lEvm_st
return;
}
Console - Scripting
INFO:jfx_bridge.bridge:Server launching in background - will continue to run after launch script finishes...
INFO:jfx_bridge.bridge:serving! (jfx_bridge v0.8.0, Python 2.7.1)
ghidra_bridge_server_background.py> Finished!
WARNING:jfx_bridge.bridge:Handling connection from ('127.0.0.1', 34530)
WARNING:jfx_bridge.bridge:Closing connection from ('127.0.0.1', 34530)
00000041
0xa1dc3570
JUMPDEST


Function Graph [CodeBrowser: evm:/three_arith_methods.evm]
File Edit Analysis Navigation Search Select Tools Help
Function Graph - 0xa1dc3570 - 5 vertices (three_arith_methods.evm)
ram:00000041 - 0xa1dc357
undefined 0xa1dc3570()
undefined <UNASSIGNED> <RETURN>
0xa1dc3570
...0041 JUMPDEST
...0042 PUSH1 0x6a
...0044 PUSH1 0x4
...0046 DUP1
...0047 CALLDATASIZE
...0048 SUB
...0049 PUSH1 0x20
...004b DUP2
...004c LT
...004d ISZERO
...004e PUSH1 0x55
...0050 JUMPI
ram:00000055 - .
LAB_ram_00000055
...0055 JUMPDEST
...0056 DUP2
...0057 ADD
...0058 SWAP1
...0059 DUP1
...005a DUP1
...005b CALLDATALOAD
...005c SWAP1
...005d PUSH1 0x20
...005f ADD
...0060 SWAP1
...0061 SWAP3
...0062 SWAP2
...0063 SWAP1
...0064 POP
...0065 POP
...0066 POP
...0067 PUSH1 0xc2
...0069 JUMP
ram:0000..
...0051 PUSH1 0x0
...0053 DUP1
...0054 REVERT
ram:000000c2 - .
LAB_ram_000000c2
...00c2 JUMPDEST
...00c3 DUP1
...00c4 PUSH1 0x0
...00c6 SLOAD
...00c7 ADD
...00c8 PUSH1 0x0
...00ca DUP2
...00cb SWAP1
...00cc SSTORE
...00cd POP
...00ce POP
...00cf JUMP
ram:0000006a - .
LAB_ram_0000006a
...006a JUMPDEST
...006b STOP

Design of Ghidra-EVM: This version

- Initially designed for Ghidra 9.1.2, tested in Linux
- Disassemble and CFG recovery (via ghidra_bridge and cfg evm builder)
- Identification of contract methods and labeling as functions in Ghidra
- Identification of method properties (via cfg evm builder)
- Identification of typical method names (via cfg evm builder)

Design of Ghidra-EVM: This version

- Identification of problematic instructions and labeling
- Experimental decompilation

Design of Ghidra-EVM: Limitations and challenges

- Ghidra seems to be designed to work with up 64-bit architectures in mind
- It’s difficult to support a 256-bit architecture in many ways:
  - Memory definitions with that word size fail to compile. Affects storage and volatile memory.
  - Same for instructions with large operands in SLEIGH: affects decompilation output.
  - This limits the analysis to a restricted size of a EVM contract.
  - Contributions and help appreciated :)

Demo 1: Simple utilization

Demo 2: Analyzing creation code

Demo 3: Extending Ghidra-EVM