Manticore Documentation

Release 0.3.7

Trail of Bits

Jul 11, 2023

CONTENTS:

Manticore is a symbolic execution tool for analysis of binaries and smart contracts.

CHAPTER

ONE

PROPERTY BASED SYMBOLIC EXECUTOR: MANTICORE-VERIFIER

Manticore installs a separated CLI tool to do property based symbolic execution of smart contracts.

```
$ manticore-verifier your_contract.sol
```

manticore-verifier initializes an emulated blockchain environment with a configurable set of accounts and then sends various symbolic transactions to the target contract containing property methods. If a way to break a property is found the full transaction trace to reproduce the behavior is provided. A configurable stopping condition bounds the exploration, properties not failing are considered to pass.

1.1 Writing properties in {Solidity/ Vyper}

manticore-verifier will detect and test property methods written in the original contract language. A property can be written in the original language by simply naming a method in a specific way. For example methods names starting with `crytic_`.

```
function crytic_test_true_property() view public returns (bool){
        return true;
    }
```

You can select your own way to name property methods using the `--propre` commandline argument.

```
--propre PROPRE       A regular expression for selecting properties
```

1.1.1 Normal properties

In the most common case after some precondition is met some logic property must always be true. Normal properties are property methods that must always return true (or REVERT).

1.1.2 Reverting properties

Sometimes it is difficult to detect that a revert has happened in an internal transaction. manticore-verifier allows to test for ALWAYS REVERTing property methods. Revert properties are property methods that must always REVERT. Reverting property are any property method that contains "revert". For example:

```
function crytic_test_must_always_revert() view public returns (bool){
        return true;
}
```

CHAPTER

TWO

SELECTING A TARGET CONTRACT

manticore-verifier needs to be pointed to the target contract containing any number of property methods. The target contract is the entry point of the exploration. It needs to initialize any internal structure or external contracts to a correct initial state. All methods of this contract matching the property name criteria will be tested.

```
--contract_name CONTRACT_NAME The target contract name defined in the source code
```

CHAPTER

THREE

USER ACCOUNTS

You can specify what are the accounts used in the exploration. Normally you do not want the owner or deployer of the contract to send the symbolic transaction and to use a separate unused account to actually check the property methods. There are 3 types of user accounts:

- deployer: The account used to create the target contract
- senders: A set of accounts used to send symbolic transactions. Think that these transactions are the ones trying to put the contract in a state that makes the property fail.
- psender: The account used as caller to test the actual property methods

You can specify those via command line arguments

```
--deployer DEPLOYER   (optional) address of account used to deploy the contract
--senders SENDERS     (optional) a comma separated list of sender addresses.
                       The properties are going to be tested sending
                       transactions from these addresses.
--psender PSENDER     (optional) address from where the property is tested
```

Or, if you prefer, you can specify a yaml file like this

```
deployer: "0x41414141414141414141"
sender: ["0x5151515151515151515151", "0x5252525252525252525252"]
psender: "0x616161616161616161"
```

If you specify the accounts both ways the commandline takes precedence over the yaml file. If you do not provide specific accounts **manticore-verifier** will choose them for you.

CHAPTER

FOUR

STOPPING CONDITION

The exploration will continue to send symbolic transactions until one of the stopping criteria is met.

4.1 Maximum number of transactions

You can be interested only in what could happen under a number of transactions. After a maximum number of transactions is reached the explorations ends. Properties that had not been found to be breakable are considered a pass. You can modify the max number of transactions to test vis a command line argument, otherwise it will stop at 3 transactions.

```
--maxt MAXT           Max transaction count to explore
```

4.2 Maximum coverage % attained

By default, if a transaction does not produce new coverage, the exploration is stopped. But you can add a further constraint so that if the provided coverage percentage is obtained, stop. Note that this is the total % of runtime bytecode covered. By default, compilers add dead code, and also in this case the runtime contains the code of the properties methods. So use with care.

```
--maxcov MAXCOV       Stop after maxcov % coverage is obtained in the main
                       contract
```

4.3 Timeout

Exploration will stop after the timeout seconds have passed.

```
--timeout TIMEOUT     Exploration timeout in seconds
```

4.4 Walkthrough

Consider this little contract containing a bug:

```
contract Ownership{  // It can have an owner!
        address owner = msg.sender;
        function Onwer() public{
                owner = msg.sender;
        }
        modifier isOwner(){
                require(owner == msg.sender);
                _;
        }
}
contract Pausable is Ownership{ //It is also pausable. You can pause it. You can resume␣
↪it.
    bool is_paused;
    modifier ifNotPaused(){
        require(!is_paused);
        _;
    }
    function paused() isOwner public{
        is_paused = true;
    }
    function resume() isOwner public{
        is_paused = false;
    }
}
contract Token is Pausable{ //<< HERE it is.
    mapping(address => uint) public balances; // It maintains a balance sheet
    function transfer(address to, uint value) ifNotPaused public{  //and can transfer␣
↪value
        balances[msg.sender] -= value; // from one account
        balances[to] += value;         // to the other
    }
}
```

Assuming the programmer did not want to allow the magic creation of tokens. We can design a property around the fact that the initial token count can not be increased over time. Even more relaxed, after the contract creation any account must have less that total count of tokens. The property looks like this :

```
contract TestToken is Token{
        constructor() public{
                //here lets initialize the thing
                balances[msg.sender] = 10000; //deployer account owns it all!
        }

        function crytic_test_balance() view public returns (bool){
                return balances[msg.sender] <= 10000; //nobody can have more than 100%␣
↪of the tokens
        }

}
```

And you can unleash the verifier like this:

```
$manticore-verifier testtoken.sol  --contract_name TestToken
```

CHAPTER

FIVE

MANTICOREBASE

CHAPTER

SIX

WORKERS

CHAPTER

SEVEN

STATES

7.1 Accessing

7.2 Operations

7.3 Inspecting

CHAPTER

EIGHT

EVM

8.1 ABI

8.2 Manager

8.3 EVM

CHAPTER

NINE

NATIVE

9.1 Platforms

9.2 Linux

9.3 Models

9.4 State

9.5 Cpu

9.6 Memory

9.7 State

9.8 Function Models

The Manticore function modeling API can be used to override a certain function in the target program with a custom implementation in Python. This can greatly increase performance.

Manticore comes with implementations of function models for some common library routines (core models), and also offers a user API for defining user-defined models.

To use a core model, use the `invoke_model()` API. The available core models are documented in the API Reference:

```
from manticore.native.models import strcmp
addr_of_strcmp = 0x400510
@m.hook(addr_of_strcmp)
def strcmp_model(state):
    state.invoke_model(strcmp)
```

To implement a user-defined model, implement your model as a Python function, and pass it to `invoke_model()`. See the `invoke_model()` documentation for more. The core models are also good examples to look at and use the same external user API.

9.9 Symbolic Input

Manticore allows you to execute programs with symbolic input, which represents a range of possible inputs. You can do this in a variety of manners.

Wildcard byte

Throughout these various interfaces, the '+' character is defined to designate a byte of input as symbolic. This allows the user to make input that mixes symbolic and concrete bytes (e.g. known file magic bytes).

For example: `"concretedata++++++++moreconcretedata++++++++++"`

Symbolic arguments/environment

To provide a symbolic argument or environment variable on the command line, use the wildcard byte where arguments and environment are specified.:

```
$ manticore ./binary +++++ +++++
$ manticore ./binary --env VAR1=+++++ --env VAR2=++++++
```

For API use, use the `argv` and `envp` arguments to the `manticore.native.Manticore.linux()` classmethod.:

```
Manticore.linux('./binary', ['++++++', '++++++'], dict(VAR1='+++++', VAR2='++++++'))
```

Symbolic stdin

Manticore by default is configured with 256 bytes of symbolic stdin data which is configurable with the `stdin_size` kwarg of `manticore.native.Manticore.linux()` , after an optional concrete data prefix, which can be provided with the `concrete_start` kwarg of `manticore.native.Manticore.linux()`.

Symbolic file input

To provide symbolic input from a file, first create the files that will be opened by the analyzed program, and fill them with wildcard bytes where you would like symbolic data to be.

For command line use, invoke Manticore with the `--file` argument.:

```
$ manticore ./binary --file my_symbolic_file1.txt --file my_symbolic_file2.txt
```

For API use, use the `add_symbolic_file()` interface to customize the initial execution state from an `__init__()`

```
@m.init
def init(initial_state):
    initial_state.platform.add_symbolic_file('my_symbolic_file1.txt')
```

Symbolic sockets

Manticore's socket support is experimental! Sockets are configured to contain 64 bytes of symbolic input.

CHAPTER

TEN

WEB ASSEMBLY

10.1 ManticoreWASM

10.2 WASM World

10.3 Executor

10.4 Module Structure

10.5 Types

CHAPTER

ELEVEN

PLUGINS

11.1 Core

will_fork_state_callback(*self*, *state*, *expression*, *solutions*, *policy*)

did_fork_state_callback(*self*, *new_state*, *expression*, *solutions*, *policy*, *children*)

will_load_state_callback(*self*, *state_id*)

did_load_state_callback(*self*, *state*, *state_id*)

will_run_callback(*self*, *ready_states*)

did_run_callback(*self*)

11.2 Worker

will_start_worker_callback(*self*, *workerid*)

will_terminate_state_callback(*self*, *current_state*, *exception*)

did_terminate_state_callback(*self*, *current_state*, *exception*)

will_kill_state_callback(*self*, *current_state*, *exception*)

did_sill_state_callback(*self*, *current_state*, *exception*)

did_terminate_worker_callback(*self*, *workerid*)

11.3 EVM

will_decode_instruction_callback(*self*, *pc*)

will_evm_execute_instruction_callback(*self*, *instruction*, *args*)

did_evm_execute_instruction_callback(*self*, *last_unstruction*, *last_arguments*, *result*)

did_evm_read_memory_callback(*self*, *offset*, *value*, *size*)

did_evm_write_memory_callback(*self*, *offset*, *value*, *size*)

on_symbolic_sha3_callback(*self*, *data*, *know_sha3*)

on_concreate_sha3_callback(*self*, *data*, *value*)

did_evm_read_code_callback(*self*, *code_offset*, *size*)

will_evm_read_storage_callback(*self*, *storage_address*, *offset*)

did_evm_read_storage_callback(*self*, *storage_address*, *offset*, *value*)

will_evm_write_storage_callback(*self*, *storage_address*, *offset*, *value*)

did_evm_write_storage_callback(*self*, *storage_address*, *offset*, *value*)

will_open_transaction_callback(*self*, *tx*)

did_open_transaction_callback(*self*, *tx*)

will_close_transaction_callback(*self*, *tx*)

did_close_transaction_callback(*self*, *tx*)

11.4 memory

will_map_memory_callback(*self*, *addr*, *size*, *perms*, *filename*, *offset*)

did_map_memory_callback(self, addr, size, perms, filename, offset, addr) # little confused on this one

will_map_memory_callback(*self*, *addr*, *size*, *perms*, *None*, *None*)

did_map_memory_callback(*self*, *addr*, *size*, *perms*, *None*, *None*, *addr*)

will_unmap_memory_callback(*self*, *start*, *size*)

did_unmap_memory_callback(*self*, *start*, *size*)

will_protect_memory_callback(*self*, *start*, *size*, *perms*)

did_protect_memory_callback(*self*, *addr*, *size*, *perms*, *filename*, *offset*)

11.5 abstractcpu

will_execute_syscall_callback(*self*, *model*)

did_execute_syscall_callback(*self*, *func_name*, *args*, *ret*)

will_write_register_callback(*self*, *register*, *value*)

did_write_register_callback(*self*, *register*, *value*)

will_read_register_callback(*self*, *register*)

did_read_register_callback(*self*, *register*, *value*)

will_write_memory_callback(*self*, *where*, *expression*, *size*)

did_write_memory_callback(*self*, *where*, *expression*, *size*)

will_read_memory_callback(*self*, *where*, *size*)

did_read_memory_callback(*self*, *where*, *size*)

did_write_memory_callback(self, where, data, num_bits) # iffy

will_decode_instruction_callback(*self*, *pc*)

will_execute_instruction_callback(*self*, *pc*, *insn*)

did_execute_instruction_callback(*self*, *last_pc*, *pc*, *insn*)

11.6 x86

will_set_descriptor_callback(*self*, *selector*, *base*, *limit*, *perms*)

did_set_descriptor_callback(*self*, *selector*, *base*, *limit*, *perms*)

CHAPTER

TWELVE

GOTCHAS

Manticore has a number of "gotchas": quirks or little things you need to do in a certain way otherwise you'll have crashes and other unexpected results.

12.1 Mutable context entries

Something like `m.context['flag'].append('a')` inside a hook will not work. You need to (unfortunately, for now) do `m.context['flag'] += ['a']`. This is related to Manticore's built in support for parallel analysis and use of the *multiprocessing* library. This gotcha is specifically related to this note from the Python documentation :

> "Note: Modifications to mutable values or items in dict and list proxies will not be propagated through the manager, because the proxy has no way of knowing when its values or items are modified. To modify such an item, you can re-assign the modified object to the container proxy"

12.2 Context locking

Manticore natively supports parallel analysis; if this is activated, client code should always be careful to properly lock the global context when accessing it.

An example of a global context race condition, when modifying two context entries.:

```
m.context['flag1'] += ['a']
--- interrupted by other worker
m.context['flag2'] += ['b']
```

Client code should use the `locked_context()` API:

```
with m.locked_context() as global_context:
    global_context['flag1'] += ['a']
    global_context['flag2'] += ['b']
```

12.3 “Random” Policy

The *random* policy, which is the Manticore default, is not actually random and is instead deterministically seeded. This means that running the same analysis twice should return the same results (and get stuck in the same places).

CHAPTER

THIRTEEN

UTILITIES

13.1 Logging

manticore.utils.log.**init_logging**(*handler: Optional[logging.Handler] = None*) → None
Initialize logging for Manticore, given a handler or by default use *default_logger()*

manticore.utils.log.**get_manticore_logger_names**() → List[str]

manticore.utils.log.**set_verbosity**(*setting: int*) → None
Set the global verbosity (0-5).

manticore.utils.log.**default_handler**() → logging.Handler
Return a default Manticore logger with a nice formatter and filter.

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

G

I

O

S

W