Manticore Documentation

Release 0.1.0

Trail of Bits

Apr 02, 2018

Contents:

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

CHAPTER 1

API Reference

This API is under active development, and should be considered unstable.

1.1 Helpers

manticore.**issymbolic**(*value*)

Helper to determine whether an object is symbolic (e.g checking if data read from memory is symbolic)

Parameters **value** (*object*) – object to check

Returns whether *value* is symbolic

Return type bool

manticore.**variadic**(*func*)

A decorator used to mark a function model as variadic. This function should take two parameters: a *State* object, and a generator object for the arguments.

Parameters **func** (*callable*) – Function model

1.2 Manticore

class manticore.**Manticore**(*path_or_state*, *argv=None*, *workspace_url=None*, *policy='random'*, ***kwargs*)

The central analysis object.

This should generally not be invoked directly; the various class method constructors should be preferred: *linux()*, *decree()*, *evm()*.

Parameters

- **path_or_state** (*str or State*) – Path to a binary to analyze (**deprecated**) or *State* object
- **argv** (*list[str]*) – Arguments to provide to binary (**deprecated**)

Variables **context** (*dict*) – Global context for arbitrary data storage

add_hook(*pc*, *callback*)
Add a callback to be invoked on executing a program counter. Pass *None* for pc to invoke callback on every instruction. *callback* should be a callable that takes one *State* argument.

Parameters

- **pc** (*int or None*) – Address of instruction to hook
- **callback** (*callable*) – Hook function

classmethod decree(*path*, *concrete_start=''*, ***kwargs*)
Constructor for Decree binary analysis.

Parameters

- **path** (*str*) – Path to binary to analyze
- **concrete_start** (*str*) – Concrete stdin to use before symbolic inputt
- **kwargs** – Forwarded to the Manticore constructor

Returns Manticore instance, initialized with a Decree State

Return type *Manticore*

classmethod evm(***kwargs*)
Constructor for Ethereum virtual machine bytecode analysis.

Parameters **kwargs** – Forwarded to the Manticore constructor

Returns Manticore instance, initialized with a EVM State

Return type *Manticore*

hook(*pc*)
A decorator used to register a hook function for a given instruction address. Equivalent to calling *add_hook()*.

Parameters **pc** (*int or None*) – Address of instruction to hook

init(*f*)
A decorator used to register a hook function to run before analysis begins. Hook function takes one *State* argument.

classmethod linux(*path*, *argv=None*, *envp=None*, *symbolic_files=None*, *concrete_start=''*, ***kwargs*)
Constructor for Linux binary analysis.

Parameters

- **path** (*str*) – Path to binary to analyze
- **argv** (*list[str]*) – Arguments to provide to the binary
- **envp** (*dict[str, str]*) – Environment to provide to the binary
- **symbolic_files** (*list[str]*) – Filenames to mark as having symbolic input
- **concrete_start** (*str*) – Concrete stdin to use before symbolic inputt
- **kwargs** – Forwarded to the Manticore constructor

Returns Manticore instance, initialized with a Linux State

Return type *Manticore*

locked_context (**args*, ***kwds*)
A context manager that provides safe parallel access to the global Manticore context. This should be used to access the global Manticore context when parallel analysis is activated. Code within the *with* block is executed atomically, so access of shared variables should occur within.

Example use:

```
with m.locked_context() as context:
    visited = context['visited']
    visited.append(state.cpu.PC)
    context['visited'] = visited
```

Optionally, parameters can specify a key and type for the object paired to this key.:

```
with m.locked_context('feature_list', list) as feature_list:
    feature_list.append(1)
```

Parameters

- **key** (*object*) – Storage key
- **value_type** (*list or dict or set*) – type of value associated with key

run (*procs=1*, *timeout=0*, *should_profile=False*)
Runs analysis.

Parameters

- **procs** (*int*) – Number of parallel worker processes
- **timeout** – Analysis timeout, in seconds

terminate ()
Gracefully terminate the currently-executing run. Typically called from within a *hook()*.

static verbosity (*level*)
Convenience interface for setting logging verbosity to one of several predefined logging presets. Valid values: 0-5.

1.3 State

class manticore.core.state.**State** (*constraints*, *platform*, ***kwargs*)
Representation of a unique program state/path.

Parameters

- **constraints** (*ConstraintSet*) – Initial constraints
- **platform** (*Platform*) – Initial operating system state

Variables **context** (*dict*) – Local context for arbitrary data storage

abandon ()
Abandon the currently-active state.

Note: This must be called from the Executor loop, or a *hook()*.

constrain (*constraint*)
Constrain state.

Parameters **constraint** (*manticore.core.smtlib.Bool*) – Constraint to add

generate_testcase(*name*, *message='State generated testcase'*)
Generate a testcase for this state and place in the analysis workspace.

Parameters

- **name** (*str*) – Short string identifying this testcase used to prefix workspace entries.
- **message** (*str*) – Longer description

invoke_model(*model*)
Invoke a *model*. A *model* is a callable whose first argument is a *State*. If the *model* models a normal (non-variadic) function, the following arguments correspond to the arguments of the C function being modeled. If the *model* models a variadic function, the following argument is a generator object, which can be used to access function arguments dynamically. The *model* callable should simply return the value that should be returned by the native function being modeled.

Parameters **model** (*callable*) – Model to invoke

new_symbolic_buffer(*nbytes*, ***options*)
Create and return a symbolic buffer of length *nbytes*. The buffer is not written into State's memory; write it to the state's memory to introduce it into the program state.

Parameters

- **nbytes** (*int*) – Length of the new buffer
- **label** (*str*) – (keyword arg only) The label to assign to the buffer
- **cstring** (*bool*) – (keyword arg only) Whether or not to enforce that the buffer is a cstring (i.e. no NULL bytes, except for the last byte). (bool)
- **taint** (*tuple or frozenset*) – Taint identifier of the new buffer

Returns `Expression` representing the buffer.

new_symbolic_value(*nbits*, *label='val'*, *taint=frozenset([])*)
Create and return a symbolic value that is *nbits* bits wide. Assign the value to a register or write it into the address space to introduce it into the program state.

Parameters

- **nbits** (*int*) – The bitwidth of the value returned
- **label** (*str*) – The label to assign to the value
- **taint** (*tuple or frozenset*) – Taint identifier of this value

Returns `Expression` representing the value

solve_buffer(*addr*, *nbytes*)
Reads *nbytes* of symbolic data from a buffer in memory at *addr* and attempts to concretize it

Parameters

- **address** (*int*) – Address of buffer to concretize
- **nbytes** (*int*) – Size of buffer to concretize

Returns Concrete contents of buffer

Return type list[int]

solve_n(*expr*, *nsolves*)
Concretize a symbolic `Expression` into *nsolves* solutions.

Parameters **expr** (*manticore.core.smtlib.Expression*) – Symbolic value to concretize

Returns Concrete value

Return type list[int]

solve_one(*expr*)
Concretize a symbolic `Expression` into one solution.

Parameters **expr** (*manticore.core.smtlib.Expression*) – Symbolic value to concretize

Returns Concrete value

Return type int

symbolicate_buffer(*data*, *label='INPUT'*, *wildcard='+'*, *string=False*, *taint=frozenset([])*)
Mark parts of a buffer as symbolic (demarked by the wildcard byte)

Parameters

- **data** (*str*) – The string to symbolicate. If no wildcard bytes are provided, this is the identity function on the first argument.
- **label** (*str*) – The label to assign to the value
- **wildcard** (*str*) – The byte that is considered a wildcard
- **string** (*bool*) – Ensure bytes returned can not be NULL
- **taint** (*tuple or frozenset*) – Taint identifier of the symbolicated data

Returns If data does not contain any wildcard bytes, data itself. Otherwise, a list of values derived from data. Non-wildcard bytes are kept as is, wildcard bytes are replaced by Expression objects.

1.4 SLinux

Symbolic Linux

class manticore.platforms.linux.**SLinux**(*programs*, *argv=None*, *envp=None*, *symbolic_files=None*, *disasm='capstone'*)
Builds a symbolic extension of a Linux OS

Parameters

- **programs** (*str*) – path to ELF binary
- **disasm** (*str*) – disassembler to be used
- **argv** (*list*) – argv not including binary
- **envp** (*list*) – environment variables
- **symbolic_files** (*tuple[str]*) – files to consider symbolic

add_symbolic_file(*symbolic_file*)
Add a symbolic file. Each '+' in the file will be considered as symbolic, other char are concretized. Symbolic files must have been defined before the call to *run()*.

Parameters **symbolic_file** (*str*) – the name of the symbolic file

1.5 Cpu

class manticore.core.cpu.abstractcpu.**Cpu**(*regfile*, *memory*, ***kwargs*)

Base class for all Cpu architectures. Functionality common to all architectures (and expected from users of a Cpu) should be here. Commonly used by platforms and py:class:manticore.core.Executor

The following attributes need to be defined in any derived class

- arch
- mode
- max_instr_width
- address_bit_size
- pc_alias
- stack_alias

all_registers

Returns all register names for this CPU. Any register returned can be accessed via a *cpu.REG* convenience interface (e.g. *cpu.EAX*) for both reading and writing.

Returns valid register names

Return type tuple[str]

read_bytes(*where*, *size*, *force=False*)

Read from memory.

Parameters

- **where** (*int*) – address to read data from
- **size** (*int*) – number of bytes
- **force** – whether to ignore memory permissions

Returns data

Return type list[int or Expression]

read_int(*where*, *size=None*, *force=False*)

Reads int from memory

Parameters

- **where** (*int*) – address to read from
- **size** – number of bits to read
- **force** – whether to ignore memory permissions

Returns the value read

Return type int or BitVec

read_register(*register*)

Dynamic interface for reading cpu registers

Parameters **register** (*str*) – register name (as listed in *self.all_registers*)

Returns register value

Return type int or long or Expression

write_bytes(*where*, *data*, *force=False*)
Write a concrete or symbolic (or mixed) buffer to memory

Parameters

- **where** (*int*) – address to write to
- **data** (*str or list*) – data to write
- **force** – whether to ignore memory permissions

write_int(*where*, *expression*, *size=None*, *force=False*)
Writes int to memory

Parameters

- **where** (*int*) – address to write to
- **expr** (*int or BitVec*) – value to write
- **size** – bit size of *expr*
- **force** – whether to ignore memory permissions

write_register(*register*, *value*)
Dynamic interface for writing cpu registers

Parameters

- **register** (*str*) – register name (as listed in *self.all_registers*)
- **value** (*int or long or Expression*) – register value

1.6 Models

Models here are intended to be passed to *invoke_model()*, not invoked directly.

manticore.models.**strlen**()

manticore.models.**strcmp**()

1.7 EVM

Symbolic EVM implementation based on the yellow paper: http://gavwood.com/paper.pdf

class manticore.ethereum.**ABI**
This class contains methods to handle the ABI. The Application Binary Interface is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction.

class SByte(*size=1*)
Unconstrained symbolic byte, not associated with any ConstraintSet

static make_function_arguments(**args*)
Serializes a sequence of arguments

static make_function_id(*method_name_and_signature*)
Makes a function hash id from a method signature

static parse(*signature*, *data*)
Deserialize function ID and arguments specified in *signature* from *data*

static serialize(*value*)
Translates a Python object to its EVM ABI serialization.

static serialize_string(*value*)
Translates a string or a tuple of chars its EVM ABI serialization

static serialize_uint(*value*, *size=32*)
Translates a Python int into a 32 byte string, MSB first

class manticore.ethereum.**ManticoreEVM**(*procs=1*, ***kwargs*)
Manticore EVM manager

Usage Ex:

```
from manticore.ethereum import ManticoreEVM, ABI
seth = ManticoreEVM()
#And now make the contract account to analyze
source_code = """
    pragma solidity ^0.4.15;
    contract AnInt {
        uint private i=0;
        function set(uint value){
            i=value
        }
    }
"""
#Initialize user and contracts
user_account = seth.create_account(balance=1000)
contract_account = seth.solidity_create_contract(source_code, owner=user_account,␣
↪balance=0)
contract_account.set(12345, value=100)

seth.report()
print seth.coverage(contract_account)
```

class SByte(*size=1*)
Unconstrained symbolic byte, not associated with any ConstraintSet

all_state_ids
IDs of the all states

Note: state with id -1 is already in memory and it is not backed on the storage

all_states
Iterates over the all states (terminated and alive)

static compile(*source_code*, *contract_name=None*)
Get initialization bytecode from a Solidity source code

count_running_states()
Running states count

count_states()
Total states count

count_terminated_states()
Terminated states count

create_account(*balance=0*, *address=None*, *code=''*)
Creates a normal account

Parameters

- **balance** (*int or SValue*) – balance to be transfered on creation
- **address** (*int*) – the address for the new contract (optional)

Returns an EVMAccount

create_contract(*owner*, *balance=0*, *address=None*, *init=None*)
Creates a contract

Parameters

- **owner** (*int or EVMAccount*) – owner account (will be default caller in any transactions)
- **balance** (*int or SValue*) – balance to be transferred on creation
- **address** (*int*) – the address for the new contract (optional)
- **init** (*str*) – initializing evm bytecode and arguments

Return type EVMAccount

finalize()
Terminate and generate testcases for all currently alive states (contract states that cleanly executed to a STOP or RETURN in the last symbolic transaction).

get_balance(*address*, *state_id=None*)
Balance for account *address* on state *state_id*

get_code(*address*, *state_id=None*)
Storage data for *offset* on account *address* on state *state_id*

get_metadata(*address*)
Gets the solidity metadata for address. This is available only if address is a contract created from solidity

get_storage(*address*, *offset*, *state_id=None*)
Storage data for *offset* on account *address* on state *state_id*

get_world(*state_id=None*)
Returns the evm world of *state_id* state.

global_coverage(*account_address*)
Returns code coverage for the contract on *account_address*. This sums up all the visited code lines from any of the explored states.

last_return(*state_id=None*)
Last returned buffer for state *state_id*

load(*state_id=None*)
Load one of the running or final states.

Parameters **state_id** (*int or None*) – If None it assumes there is a single running state

make_symbolic_buffer(*size*)
Creates a symbolic buffer of size bytes to be used in transactions. You can not operate on it. It is intended as a place holder for the real expression.

Example use:

```
symbolic_data = seth.make_symbolic_buffer(320)
seth.transaction(caller=attacker_account,
                 address=contract_account,
                 data=symbolic_data,
                 value=100000 )
```

make_symbolic_value()
Creates a symbolic value, normally a uint256, to be used in transactions. You can not operate on it. It is intended as a place holder for the real expression.

Example use:

```
symbolic_value = seth.make_symbolic_value()
seth.transaction(caller=attacker_account,
                 address=contract_account,
                 data=data,
                 value=symbolic_data )
```

run(***kwargs*)
Run any pending transaction on any running state

running_state_ids
IDs of the running states

running_states
Iterates over the running states

save(*state*, *final=False*)
Save a state in secondary storage and add it to running or final lists

Parameters

- **state** – A manticore State
- **final** – True if state is final

Returns a state id

solidity_create_contract(*source_code*, *owner*, *contract_name=None*, *balance=0*, *address=None*, *args=()*)
Creates a solidity contract

Parameters

- **source_code** (*str*) – solidity source code
- **owner** (*int or EVMAccount*) – owner account (will be default caller in any transactions)
- **contract_name** (*str*) – Name of the contract to analyze (optional if there is a single one in the source code)
- **balance** (*int or SValue*) – balance to be transferred on creation
- **address** (*int or EVMAccount*) – the address for the new contract (optional)
- **args** (*tuple*) – constructor arguments

Return type EVMAccount

terminate_state_id(*state_id*)
Manually terminates a states by state_id. Moves the state from the running list into the terminated list and generates a testcase for it

terminated_state_ids
IDs of the terminated states

terminated_states
Iterates over the terminated states

transaction(*caller*, *address*, *value*, *data*)
Issue a symbolic transaction

Parameters

- **caller** (*int or EVMAccount*) – the address of the account sending the transaction
- **address** (*int or EVMAccount*) – the address of the contract to call
- **value** (*int or SValue*) – balance to be transfered on creation
- **data** – initial data

Returns an EVMAccount

Raises **NoAliveStates** – if there are no alive states to execute

transactions(*state_id=None*)
Transactions list for state *state_id*

world
The world instance or None if there is more than one state

1.8 EVM Assembler

class EVMAsm.**Instruction**(*opcode*, *name*, *operand_size*, *pops*, *pushes*, *fee*, *description*, *operand=None*, *offset=0*)

bytes
Encoded instruction

description
Coloquial description of the instruction

fee
The basic gas fee of the instruction

group
Instruction classification as per the yellow paper

has_operand
True if the instruction uses an immediate operand

is_arithmetic
True if the instruction is an arithmetic operation

is_branch
True if the instruction is a jump

is_environmental
True if the instruction access enviromental data

is_system
True if the instruction is a system operation

is_terminator
True if the instruction is a basic block terminator

name
The instruction name/mnemonic

offset
Location in the program (optional)

opcode
The opcode as an integer

operand
The immediate operand

operand_size
The immediate operand size

parse_operand(*buf*)
Parses an operand from buf

Parameters **buf** (*iterator/generator/string*) – a buffer

pops
Number words popped from the stack

pushes
Number words pushed to the stack

reads_from_memory
True if the instruction reads from memory

reads_from_stack
True if the instruction reads from stack

reads_from_storage
True if the instruction reads from the storage

semantics
Canonical semantics

size
Size of the encoded instruction

uses_block_info
True if the instruction access block information

uses_stack
True if the instruction reads/writes from/to the stack

writes_to_memory
True if the instruction writes to memory

writes_to_stack
True if the instruction writes to the stack

writes_to_storage
True if the instruction writes to the storage

class manticore.platforms.evm.**EVMAsm**
EVM Instruction factory

Example use:

```
>>> from manticore.platforms.evm import EVMAsm
>>> EVMAsm.disassemble_one('\x60\x10')
Instruction(0x60, 'PUSH', 1, 0, 1, 0, 'Place 1 byte item on stack.', 16, 0)
>>> EVMAsm.assemble_one('PUSH1 0x10')
Instruction(0x60, 'PUSH', 1, 0, 1, 0, 'Place 1 byte item on stack.', 16, 0)
>>> tuple(EVMAsm.disassemble_all('\x30\x31'))
```

```
(Instruction(0x30, 'ADDRESS', 0, 0, 1, 2, 'Get address of currently executing␣
↪account.', None, 0),
 Instruction(0x31, 'BALANCE', 0, 1, 1, 20, 'Get balance of the given account.',␣
↪None, 1))
>>> tuple(EVMAsm.assemble_all('ADDRESS\nBALANCE'))
(Instruction(0x30, 'ADDRESS', 0, 0, 1, 2, 'Get address of currently executing␣
↪account.', None, 0),
 Instruction(0x31, 'BALANCE', 0, 1, 1, 20, 'Get balance of the given account.',␣
↪None, 1))
>>> EVMAsm.assemble_hex(
...                          """PUSH1 0x60
...                             BLOCKHASH
...                             MSTORE
...                             PUSH1 0x2
...                             PUSH2 0x100
...                          """
...                      )
'0x606040526002610100'
>>> EVMAsm.disassemble_hex('0x606040526002610100')
'PUSH1 0x60\nBLOCKHASH\nMSTORE\nPUSH1 0x2\nPUSH2 0x100'
```

static assemble(*asmcode*, *offset=0*)

Assemble an EVM program

Parameters

- **asmcode** (*str*) – an evm assembler program
- **offset** – offset of the first instruction in the bytecode(optional)

Returns the hex representation of the bytecode

Example use:

```
>>> EVMAsm.assemble(  """PUSH1 0x60
                          BLOCKHASH
                          MSTORE
                          PUSH1 0x2
                          PUSH2 0x100
                       """
                   )
...
"```@R`}a{\"
```

static assemble_all(*assembler*, *offset=0*)

Assemble a sequence of textual representation of EVM instructions

Parameters

- **assembler** – assembler code for any number of instructions
- **offset** – offset of the first instruction in the bytecode(optional)

Returns An generator of Instruction objects

Example use:

```
>>> evm.EVMAsm.encode_one("""PUSH1 0x60
    PUSH1 0x40
    MSTORE
    PUSH1 0x2
```

```
    PUSH2 0x108
    PUSH1 0x0
    POP
    SSTORE
    PUSH1 0x40
    MLOAD
    """)
```

static assemble_hex(*asmcode*, *offset=0*)

Assemble an EVM program

Parameters

- **asmcode** (*str*) – an evm assembler program
- **offset** – offset of the first instruction in the bytecode(optional)

Returns the hex representation of the bytecode

Example use:

```
>>> EVMAsm.assemble_hex(  """PUSH1 0x60
                           BLOCKHASH
                           MSTORE
                           PUSH1 0x2
                           PUSH2 0x100
                       """
                    )
...
"0x6060604052600261010"
```

static assemble_one(*assembler*, *offset=0*)

Assemble one EVM instruction from its textual representation.

Parameters

- **assembler** – assembler code for one instruction
- **offset** – offset of the instruction in the bytecode (optional)

Returns An Instruction object

Example use:

```
>>> print evm.EVMAsm.assemble_one('LT')
```

static disassemble(*bytecode*, *offset=0*)

Disassemble an EVM bytecode

Parameters

- **bytecode** (*str*) – binary representation of an evm bytecode (hexadecimal)
- **offset** – offset of the first instruction in the bytecode(optional)

Returns the text representation of the aseembler code

Example use:

```
>>> EVMAsm.disassemble("```@R`}a{\")
...
PUSH1 0x60
BLOCKHASH
```

```
MSTORE
PUSH1 0x2
PUSH2 0x100
```

static disassemble_all(*bytecode*, *offset=0*)
Decode all instructions in bytecode

Parameters

- **bytecode** (*iterator/sequence/str*) – an evm bytecode (binary)
- **offset** – offset of the first instruction in the bytecode(optional)

Returns An generator of Instruction objects

Example use:

```
>>> for inst in EVMAsm.decode_all(bytecode):
...     print inst

...
PUSH1 0x60
PUSH1 0x40
MSTORE
PUSH1 0x2
PUSH2 0x108
PUSH1 0x0
POP
SSTORE
PUSH1 0x40
MLOAD
```

static disassemble_hex(*bytecode*, *offset=0*)
Disassemble an EVM bytecode

Parameters

- **bytecode** (*str*) – canonical representation of an evm bytecode (hexadecimal)
- **offset** (*int*) – offset of the first instruction in the bytecode(optional)

Returns the text representation of the aseembler code

Example use:

```
>>> EVMAsm.disassemble_hex("0x60606040526002610100")
...
PUSH1 0x60
BLOCKHASH
MSTORE
PUSH1 0x2
PUSH2 0x100
```

static disassemble_one(*bytecode*, *offset=0*)
Decode a single instruction from a bytecode

Parameters

- **bytecode** (*iterator/sequence/str*) – the bytecode stream
- **offset** – offset of the instruction in the bytecode(optional)

Returns an Instruction object

Example use:

```
>>> print EVMAsm.assemble_one('PUSH1 0x10')
```

CHAPTER 2

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.

2.1 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++++++++++"

2.2 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.Manticore.linux()`* classmethod.:

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

2.3 Symbolic stdin

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

2.4 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()* hook.:

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

2.5 Symbolic sockets

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

CHAPTER 3

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.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.

CHAPTER 4

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.

4.1 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"

4.2 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']
```

4.3 “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 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

T

U

V

W