# Manticore Documentation

***Release 0.3.1***

**Trail of Bits**

**Aug 20, 2019**

# Contents:

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

# CHAPTER 1

# ManticoreBase

**class** manticore.core.manticore.**ManticoreBase**(*initial_state*, *workspace_url=None*, *policy='random'*, ***kwargs*)

**__init__**(*initial_state*, *workspace_url=None*, *policy='random'*, ***kwargs*)

**Parameters** **initial_state** – State to start from.

Manticore symbolically explores program states.

**Manticore phases**

Manticore has multiprocessing capabilities. Several worker processes could be registered to do concurrent exploration of the READY states. Manticore can be itself at different phases: STANDBY, RUNNING.

```
      +---------+               +---------+
----->| STANDBY +<------------->+ RUNNING |
      +---------+               +----+----+
```

*Phase STANDBY*

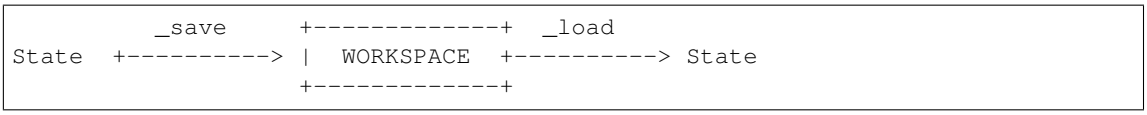
Manticore starts at STANDBY with a single initial state. Here the user can inspect, modify and generate testcases for the different states. The workers are paused and not doing any work. Actions: run()

*Phase RUNNING*

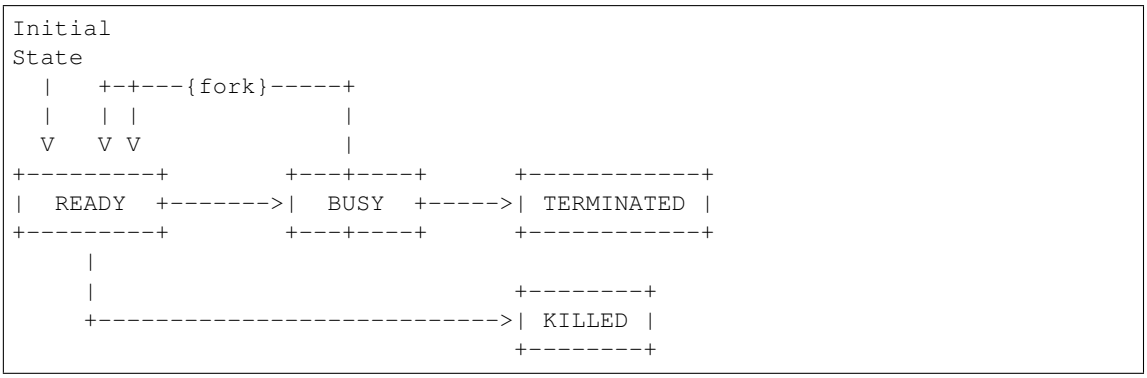
At RUNNING the workers consume states from the READY state list and potentially fork new states or terminate states. A RUNNING manticore can be stopped back to STANDBY. Actions: stop()

**States and state lists**

A state contains all the information of the running program at a given moment. State snapshots are saved to the workspace often. Internally Manticore associates a fresh id with each saved state. The memory copy of the state is then changed by the emulation of the specific arch. Stored snapshots are periodically updated using: _save() and _load().




During exploration Manticore spawns a number of temporary states that are maintained in different lists:




At any given time a state must be at the READY, BUSY, TERMINATED or KILLED list.

*State list: READY*

The READY list holds all the runnable states. Internally a state is added to the READY list via method *_put_state(state)*. Workers take states from the READY list via the *_get_state(wait=True|False)* method. A worker mainloop will consume states from the READY list and mark them as BUSYwhile working on them. States in the READY list can go to BUSY or KILLED

*State list: BUSY*

When a state is selected for exploration from the READY list it is marked as busy and put in the BUSY list. States being explored will be constantly modified and only saved back to storage when moved out of the BUSY list. Hence, when at BUSY the stored copy of the state will be potentially outdated. States in the BUSY list can go to TERMINATED, KILLED or they can be {forked} back to READY. The forking process could involve generating new child states and removing the parent from all the lists.

*State list: TERMINATED*

TERMINATED contains states that have reached a final condition and raised TerminateState. Worker's mainloop simply moves the states that requested termination to the TERMINATED list. This is a final list.

`An inherited Manticore class like ManticoreEVM could internally revive the states in TERMINATED that pass some condition and move them back to READY so the user can apply a following transaction.`

*State list: KILLED*

KILLED contains all the READY and BUSY states found at a cancel event. Manticore supports interactive analysis and has a prominetnt event system A useror ui can stop or cancel the exploration at any time. The unfinnished states cought at this situation are simply moved to its own list for further user action. This is a final list.

**Parameters**

- **initial_state** – the initial root *State* object
- **workspace_url** – workspace folder name
- **policy** – scheduling policy
- **kwargs** – other kwargs, e.g.

**at_not_running**()
Allows the decorated method to run only when manticore is NOT exploring states

**at_running**()
Allows the decorated method to run only when manticore is actively exploring states

**context**
Convenient access to shared context. We maintain a local copy of the share context during the time manticore is not running. This local context is copied to the shared context when a run starts and copied back when a run finishes

**count_states**()
Total states count

**finalize**()
Generate a report testcase for every state in the system and remove all temporary files/streams from the workspace

**is_killed**()
True if workers are killed. It is safe to join them

**is_running**()
True if workers are exploring BUSY states or waiting for READY states

**kill**()
Attempt to cancel and kill all the workers. Workers must terminate RUNNING, STANDBY -> KILLED

**kill_timeout**(*timeout=None*)
A convenient context manager that will kill a manticore run after timeout seconds

**locked_context**(*key=None*, *value_type=<class 'list'>*)
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['visited'].append(state.cpu.PC)
```

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)
```

Note: If standard (non-proxy) list or dict objects are contained in a referent, modifications to those mutable values will not be propagated through the manager because the proxy has no way of knowing when the values contained within are modified. However, storing a value in a container proxy (which triggers a __setitem__ on the proxy object) does propagate through the manager and so to effectively modify such an item, one could re-assign the modified value to the container proxy:

**Parameters**

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

**remove_all**()
Deletes all streams from storage and clean state lists

**run**(*timeout=None*)
Runs analysis.

**Parameters** **timeout** – Analysis timeout, in seconds

**subscribe**(*name*, *callback*)
Register a callback to an event

**sync**()
Synchronization decorator

**unregister_plugin**(*plugin*)
Removes a plugin from manticore. No events should be sent to it after

**static verbosity**(*level*)
Sets global vervosity level. This will activate different logging profiles globally depending on the provided numeric value

**wait**(*condition*)
Waits for the condition callable to return True

# CHAPTER 2

# Workers

**class** manticore.core.worker.**Worker**(*, *id*, *manticore*, *single=False*)

A Manticore Worker. This will run forever potentially in a different process. Normally it will be spawned at Manticore constructor and will stay alive until killed. A Worker can be in 3 phases: STANDBY, RUNNING, KILLED. And will react to different events: start, stop, kill. The events are transmitted via 2 conditional variable: m._killed and m._started.

```
STANDBY:   Waiting for the start event
RUNNING:   Exploring and spawning states until no more READY states or
the cancel event is received
KIlLED:    This is the end. No more manticoring in this worker process

             +---------+     +---------+
        +--->+ STANDBY +<--->+ RUNNING |
             +-+-------+     +-------+-+
               |                     |
               |      +--------+     |
               +----->+ KILLED <-----+
                      +----+---+
                           |
                           #
```

**join**()

**run**(**args*)

**start**()

manticore.core.**worker**

alias of *manticore.core.worker*

CHAPTER 3

# States

## 3.1 Accessing

**class** manticore.core.manticore.**ManticoreBase**(*initial_state*, *workspace_url=None*, *policy='random'*, ***kwargs*)

**all_states**
Iterates over the all states (ready and terminated and cancelled) It holds a lock so no changes state lists are allowed

See also *ready_states*.

**count_busy_states**()
Busy states count

**count_killed_states**()
Cancelled states count

**count_ready_states**()
Ready states count

**count_terminated_states**()
Terminated states count

**killed_states**
Iterates over the cancelled/killed states.

See also *ready_states*.

**ready_states**
Iterator over ready states. It supports state changes. State changes will be saved back at each iteration.

The state data change must be done in a loop, e.g. *for state in ready_states: ...* as we re-save the state when the generator comes back to the function.

This means it is not possible to change the state used by Manticore with *states = list(m.ready_states)*.

**terminated_states**
Iterates over the terminated states.

See also *ready_states*.

## 3.2 Operations

**class** manticore.core.state.**StateBase**(*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().

**can_be_false**(*expr*)

**can_be_true**(*expr*)

**concretize**(*symbolic*, *policy*, *maxcount=7*)
This finds a set of solutions for symbolic using policy. This raises TooManySolutions if more solutions than maxcount

**constrain**(*constraint*)
Constrain state.

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

**constraints**

**context**

**execute**()

**id**

**input_symbols**

**is_feasible**()

**migrate_expression**(*expression*)

**must_be_true**(*expr*)

**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=None*, *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

**platform**

**solve_buffer**(*addr*, *nbytes*, *constrain=False*)
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
- **constrain** (*bool*) – If True, constrain the buffer to the concretized value

**Returns** Concrete contents of buffer

**Return type** list[int]

**solve_max**(*expr*)
Solves a symbolic Expression into its maximum solution

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

**Returns** Concrete value

**Return type** list[int]

**solve_min**(*expr*)
Solves a symbolic Expression into its minimum solution

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

**Returns** Concrete value

**Return type** list[int]

**solve_minmax**(*expr*)
Solves a symbolic Expression into its minimum and maximun solution. Only defined for bitvects.

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

**Returns** Concrete value

**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*, *constrain=False*)
Concretize a symbolic `Expression` into one solution.

**Parameters**

- **expr** (*manticore.core.smtlib.Expression*) – Symbolic value to concretize
- **constrain** (*bool*) – If True, constrain expr to concretized value

**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.

# CHAPTER 4

# EVM

## 4.1 ABI

**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.

**static deserialize**(*type_spec*, *data*)

**static function_call**(*type_spec*, **args*)

Build transaction data from function signature and arguments

**static function_selector**(*method_name_and_signature*)

Makes a function hash id from a method signature

**static serialize**(*ty*, **values*, ***kwargs*)

Serialize value using type specification in ty. ABI.serialize('int256', 1000) ABI.serialize('(int, int256)', 1000, 2000)

## 4.2 Manager

**class** manticore.ethereum.**ManticoreEVM**(*workspace_url: str = None*, *policy: str = 'random'*)

Manticore EVM manager

Usage Ex:

```
from manticore.ethereum import ManticoreEVM, ABI
m = ManticoreEVM()
#And now make the contract account to analyze
source_code = '''
    pragma solidity ^0.4.15;
    contract AnInt {
```

(continues on next page)

(continued from previous page)

```
        uint private i=0;
        function set(uint value){
            i=value
        }
    }
'''
#Initialize user and contracts
user_account = m.create_account(balance=1000)
contract_account = m.solidity_create_contract(source_code, owner=user_account,␣
↪balance=0)
contract_account.set(12345, value=100)

m.finalize()
```

**account_name**(*address*)

**accounts**

**static compile**(*source_code*, *contract_name=None*, *libraries=None*, *runtime=False*, *cry-tic_compile_args={}*)

Get initialization bytecode from a Solidity source code

**completed_transactions**

**constrain**(*constraint*)

**contract_accounts**

**create_account**(*balance=0*, *address=None*, *code=None*, *name=None*)

Low level creates an account. This won't generate a transaction.

**Parameters**

- **balance** (*int or BitVecVariable*) – balance to be set on creation (optional)
- **address** (*int*) – the address for the new account (optional)
- **code** – the runtime code for the new account (None means normal account), str or bytes (optional)
- **name** – a global account name eg. for use as reference in the reports (optional)

**Returns** an EVMAccount

**create_contract**(*owner*, *balance=0*, *address=None*, *init=None*, *name=None*, *gas=None*)

Creates a contract

**Parameters**

- **owner** (*int or EVMAccount*) – owner account (will be default caller in any transactions)
- **balance** (*int or BitVecVariable*) – balance to be transferred on creation
- **address** (*int*) – the address for the new contract (optional)
- **init** (*str*) – initializing evm bytecode and arguments
- **name** (*str*) – a unique name for reference
- **gas** – gas budget for the creation/initialization of the contract

**Return type** EVMAccount

**current_location**(*state*)

**finalize**(*procs=10*)
Terminate and generate testcases for all currently alive states (contract states that cleanly executed to a STOP or RETURN in the last symbolic transaction).

**Parameters** **procs** – nomber of local processes to use in the reporting generation

**generate_testcase**(*state*, *message=''*, *only_if=None*, *name='user'*)
Generate a testcase to the workspace for the given program state. The details of what a testcase is depends on the type of Platform the state is, but involves serializing the state, and generating an input (concretizing symbolic variables) to trigger this state.

The only_if parameter should be a symbolic expression. If this argument is provided, and the expression *can be true* in this state, a testcase is generated such that the expression will be true in the state. If it *is impossible* for the expression to be true in the state, a testcase is not generated.

This is useful for conveniently checking a particular invariant in a state, and generating a testcase if the invariant can be violated.

For example, invariant: "balance" must not be 0. We can check if this can be violated and generate a testcase:

```
m.generate_testcase(state, 'balance CAN be 0', only_if=balance == 0)
# testcase generated with an input that will violate invariant (make balance
↪== 0)
```

**Parameters**

- **state** (*manticore.core.state.State*) –
- **message** (*str*) – longer description of the testcase condition
- **only_if** (*manticore.core.smtlib.Bool*) – only if this expr can be true, generate testcase. if is None, generate testcase unconditionally.
- **name** (*str*) – short string used as the prefix for the workspace key (e.g. filename prefix for testcase files)

**Returns** If a testcase was generated

**Return type** bool

**get_account**(*name*)

**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*) → Optional[manticore.ethereum.solidity.SolidityMetadata]
Gets the solidity metadata for address. This is available only if address is a contract created from solidity

**get_nonce**(*address*)

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

**global_findings**

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

**json_create_contract**(*jfile*, *owner=None*, *name=None*, *contract_name=None*, *balance=0*, *gas=None*, *network_id=None*, *args=()*)
Creates a solidity contract based on a truffle json artifact. https://github.com/trufflesuite/truffle/tree/develop/packages/truffle-contract-schema

**Parameters**

- **jfile** (*str or IOBase*) – truffle json artifact
- **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 BitVecVariable*) – balance to be transferred on creation
- **gas** (*int*) – gas budget for each contract creation needed (may be more than one if several related contracts defined in the solidity source)
- **network_id** – Truffle network id to instantiate
- **args** (*tuple*) – constructor arguments

**Return type** EVMAccount

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

**make_symbolic_address**(*name=None*, *select='both'*)
Creates a symbolic address and constrains it to pre-existing addresses or the 0 address.

**Parameters**

- **name** – Name of the symbolic variable. Defaults to 'TXADDR' and later to 'TXADDR_<number>'
- **select** – Whether to select contracts or normal accounts. Not implemented for now.

**Returns** Symbolic address in form of a BitVecVariable.

**make_symbolic_arguments**(*types*)
Build a reasonable set of symbolic arguments matching the types list

**make_symbolic_buffer**(*size*, *name=None*, *avoid_collisions=False*)
Creates a symbolic buffer of size bytes to be used in transactions. You can operate on it normally and add constraints to manticore.constraints via manticore.constrain(constraint_expression)

Example use:

```
symbolic_data = m.make_symbolic_buffer(320)
m.constrain(symbolic_data[0] == 0x65)
m.transaction(caller=attacker_account,
                address=contract_account,
                data=symbolic_data,
                value=100000 )
```

**make_symbolic_value**(*nbits=256*, *name=None*)
Creates a symbolic value, normally a uint256, to be used in transactions. You can operate on it normally and add constraints to manticore.constraints via manticore.constrain(constraint_expression)

Example use:

```
symbolic_value = m.make_symbolic_value()
m.constrain(symbolic_value > 100)
m.constrain(symbolic_value < 1000)
m.transaction(caller=attacker_account,
                address=contract_account,
                data=data,
                value=symbolic_value )
```

**multi_tx_analysis**(*solidity_filename*, *contract_name=None*, *tx_limit=None*, *tx_use_coverage=True*, *tx_send_ether=True*, *tx_account='attacker'*, *tx_preconstrain=False*, *args=None*, *crytic_compile_args={}*)

**new_address**()
Create a fresh 160bit address

**normal_accounts**

**preconstraint_for_call_transaction**(*address: Union[int, manticore.ethereum.account.EVMAccount], data: manticore.core.smtlib.expression.Array, value: Union[int, manticore.core.smtlib.expression.Expression, None] = None, contract_metadata: Optional[manticore.ethereum.solidity.SoftwareMetadata] = None*) → manticore.core.smtlib.expression.BoolOperation
Returns a constraint that excludes combinations of value and data that would cause an exception in the EVM contract dispatcher.

**Parameters**

- **address** – address of the contract to call
- **value** – balance to be transferred (optional)
- **data** – symbolic transaction data
- **contract_metadata** – SolidityMetadata for the contract (optional)

**register_detector**(*d*)
Unregisters a plugin. This will invoke detector's *on_unregister* callback. Shall be called after *.finalize*.

**run**(***kwargs*)
Runs analysis.

**Parameters** **timeout** – Analysis timeout, in seconds

**solidity_create_contract**(*source_code*, *owner*, *name=None*, *contract_name=None*, *libraries=None*, *balance=0*, *address=None*, *args=()*, *gas=None*, *crytic_compile_args=None*)
Creates a solidity contract and library dependencies

**Parameters**

- **source_code** (*string (filename, directory, etherscan address) or a file handle*) – 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 BitVecVariable*) – balance to be transferred on creation
- **address** (*int or EVMAccount*) – the address for the new contract (optional)
- **args** (*tuple*) – constructor arguments
- **crytic_compile_args** (*dict*) – crytic compile options (https://github.com/crytic/crytic-compile/wiki/Configuration)
- **gas** (*int*) – gas budget for each contract creation needed (may be more than one if several related contracts defined in the solidity source)

**Return type** EVMAccount

**transaction**(*caller*, *address*, *value*, *data*, *gas=None*)
Issue a symbolic transaction in all running states

**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 BitVecVariable*) – balance to be transfered on creation
- **data** – initial data
- **gas** – gas budget

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

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

**unregister_detector**(*d*)
Unregisters a detector. This will invoke detector's *on_unregister* callback. Shall be called after *.finalize* - otherwise, finalize won't add detector's finding to *global.findings*.

**workspace**

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

## 4.3 EVM

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

**exception** manticore.platforms.evm.**ConcretizeArgument**(*pos*, *expression=None*, *policy='SAMPLED'*)
Raised when a symbolic argument needs to be concretized.

**exception** manticore.platforms.evm.**ConcretizeFee**(*policy='MINMAX'*)
Raised when a symbolic gas fee needs to be concretized.

**exception** manticore.platforms.evm.**ConcretizeGas**(*policy='MINMAX'*)
Raised when a symbolic gas needs to be concretized.

**class** manticore.platforms.evm.**EVM**(*constraints*, *address*, *data*, *caller*, *value*, *bytecode*, *world=None*, *gas=210000*, ***kwargs*)
Machine State. The machine state is defined as the tuple (g, pc, m, i, s) which are the gas available, the program counter pc , the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents are a series of zeroes of bitsize 256

**SAR**(*a*, *b*)
Arithmetic Shift Right operation

**SHL**(*a*, *b*)
Shift Left operation

**SHR**(*a*, *b*)
Logical Shift Right operation

**allocated**

**bytecode**

**change_last_result**(*result*)

**static check256int**(*value*)

**constraints**

**disassemble**()

**execute**()

**gas**

**instruction**
Current instruction pointed by self.pc

**read_buffer**(*offset*, *size*)

**read_code**(*address*, *size=1*)
Read size byte from bytecode. If less than size bytes are available result will be pad with

**safe_add**(*a*, *b*)

**safe_mul**(*a*, *b*)

**class transact**(*pre=None*, *pos=None*, *doc=None*)

**pos**(*pos*)

**try_simplify_to_constant**(*data*)

**world**

**write_buffer**(*offset*, *data*)

**exception** manticore.platforms.evm.**EVMException**

**class** manticore.platforms.evm.**EVMLog**(*address*, *memlog*, *topics*)

**address**
Alias for field number 0

**memlog**
Alias for field number 1

**topics**
Alias for field number 2

**class** manticore.platforms.evm.**EVMWorld**(*constraints*, *storage=None*, *blocknumber=None*, *timestamp=None*, *difficulty=0*, *gaslimit=0*, *coinbase=0*, ***kwargs*)

**accounts**

**add_refund**(*value*)

**add_to_balance**(*address*, *value*)

**all_transactions**

**block_coinbase**()

**block_difficulty**()

**block_gaslimit**()

**block_hash**(*block_number=None*, *force_recent=True*)
Calculates a block's hash

**Parameters**

- **block_number** – the block number for which to calculate the hash, defaulting to the most recent block
- **force_recent** – if True (the default) return zero for any block that is in the future or older than 256 blocks

**Returns** the block hash

**block_number**()

**block_prevhash**()

**block_timestamp**()

**static calculate_new_address**(*sender=None*, *nonce=None*)

**constraints**

**contract_accounts**

**create_account**(*address=None*, *balance=0*, *code=None*, *storage=None*, *nonce=None*)
Low level account creation. No transaction is done.

**Parameters**

- **address** – the address of the account, if known. If omitted, a new address will be generated as closely to the Yellow Paper as possible.
- **balance** – the initial balance of the account in Wei
- **code** – the runtime code of the account, if a contract
- **storage** – storage array
- **nonce** – the nonce for the account; contracts should have a nonce greater than or equal to 1

**create_contract**(*price=0*, *address=None*, *caller=None*, *balance=0*, *init=None*, *gas=None*)
Create a contract account. Sends a transaction to initialize the contract

**Parameters**

- **address** – the address of the new account, if known. If omitted, a new address will be generated as closely to the Yellow Paper as possible.
- **balance** – the initial balance of the account in Wei
- **init** – the initialization code of the contract

The way that the Solidity compiler expects the constructor arguments to be passed is by appending the arguments to the byte code produced by the Solidity compiler. The arguments are formatted as defined in the Ethereum ABI2. The arguments are then copied from the init byte array to the EVM memory through the CODECOPY opcode with appropriate values on the stack. This is done when the byte code in the init byte array is actually run on the network.

**current_human_transaction**
Current ongoing human transaction

**current_transaction**
current tx

**current_vm**
current vm

**delete_account**(*address*)

**deleted_accounts**

**depth**

**dump**(*stream*, *state*, *mevm*, *message*)

**execute**()

**get_balance**(*address*)

**get_code**(*address*)

**get_nonce**(*address*)

**get_storage**(*address*)
Gets the storage of an account

**Parameters** **address** – account address

**Returns** account storage

**Return type** bytearray or ArrayProxy

**get_storage_data**(*storage_address*, *offset*)
Read a value from a storage slot on the specified account

**Parameters**

- **storage_address** – an account address
- **offset** (*int or BitVec*) – the storage slot to use.

**Returns** the value

**Return type** int or BitVec

**get_storage_items**(*address*)
Gets all items in an account storage

**Parameters** **address** – account address

**Returns** all items in account storage. items are tuple of (index, value). value can be symbolic

**Return type** list[(storage_index, storage_value)]

**has_code**(*address*)

**has_storage**(*address*)
True if something has been written to the storage. Note that if a slot has been erased from the storage this function may lose any meaning.

**human_transactions**
Completed human transaction

**increase_nonce**(*address*)

**last_human_transaction**
Last completed human transaction

**last_transaction**
Last completed transaction

**log**(*address*, *topics*, *data*)

**log_storage**(*addr*)

**logs**

**new_address**(*sender=None*, *nonce=None*)
Create a fresh 160bit address

**normal_accounts**

**send_funds**(*sender*, *recipient*, *value*)

**set_balance**(*address*, *value*)

**set_code**(*address*, *data*)

**set_storage_data**(*storage_address*, *offset*, *value*)
Writes a value to a storage slot in specified account

**Parameters**

- **storage_address** – an account address
- **offset** (*int or BitVec*) – the storage slot to use.
- **value** (*int or BitVec*) – the value to write

**start_transaction**(*sort*, *address*, *, *price=None*, *data=None*, *caller=None*, *value=0*, *gas=2300*)
Initiate a transaction

**Parameters**

- **sort** – the type of transaction. CREATE or CALL or DELEGATECALL
- **address** – the address of the account which owns the code that is executing.
- **price** – the price of gas in the transaction that originated this execution.
- **data** – the byte array that is the input data to this execution
- **caller** – the address of the account which caused the code to be executing. A 160-bit code used for identifying Accounts
- **value** – the value, in Wei, passed to this account as part of the same procedure as execution. One Ether is defined as being 10**18 Wei.
- **bytecode** – the byte array that is the machine code to be executed.
- **gas** – gas budget for this transaction.

**transaction**(*address*, *price=0*, *data=''*, *caller=None*, *value=0*, *gas=2300*)

**transactions**
Completed completed transaction

**tx_gasprice**()

**tx_origin**()

**exception** manticore.platforms.evm.**EndTx**(*result*, *data=None*)
The current transaction ends

**is_rollback**()

**exception** manticore.platforms.evm.**InvalidOpcode**
Trying to execute invalid opcode

**exception** manticore.platforms.evm.**NotEnoughGas**
Not enough gas for operation

**class** manticore.platforms.evm.**PendingTransaction**(*type*, *address*, *price*, *data*, *caller*, *value*, *gas*)

**address**
Alias for field number 1

**caller**
Alias for field number 4

**data**
Alias for field number 3

**gas**
Alias for field number 6

**price**
Alias for field number 2

**type**
Alias for field number 0

**value**
Alias for field number 5

**exception** manticore.platforms.evm.**Return**(*data=bytearray(b")*)
Program reached a RETURN instruction

**exception** manticore.platforms.evm.**Revert**(*data*)
Program reached a REVERT instruction

**exception** manticore.platforms.evm.**SelfDestruct**
Program reached a SELFDESTRUCT instruction

**exception** manticore.platforms.evm.**StackOverflow**
Attempted to push more than 1024 items

**exception** manticore.platforms.evm.**StackUnderflow**
Attempted to pop from an empty stack

**exception** manticore.platforms.evm.**StartTx**
A new transaction is started

**exception** manticore.platforms.evm.**Stop**
Program reached a STOP instruction

**exception** manticore.platforms.evm.**TXError**
A failed Transaction

**class** manticore.platforms.evm.**Transaction**(*sort*, *address*, *price*, *data*, *caller*, *value*, *gas=0*, *depth=None*, *result=None*, *return_data=None*)

**address**

**caller**

**concretize**(*state*)

**data**

**depth**

**dump**(*stream*, *state*, *mevm*, *conc_tx=None*)
Concretize and write a human readable version of the transaction into the stream. Used during testcase generation.

**Parameters**

- **stream** – Output stream to write to. Typically a file.
- **state** (*manticore.ethereum.State*) – state that the tx exists in
- **mevm** (manticore.ethereum.ManticoreEVM) – manticore instance

**Returns**

**gas**

**is_human**
Returns whether this is a transaction made by human (in a script).

**As an example for:** contract A { function a(B b) { b.b(); } } contract B { function b() {} }

Calling *B.b()* makes a human transaction. Calling *A.a(B)* makes a human transaction which makes an internal transaction (b.b()).

**price**

**result**

**return_data**

**return_value**

**set_result**(*result*, *return_data=None*)

**sort**

**to_dict**(*mevm*)
Only meant to be used with concrete Transaction objects! (after calling .concretize())

**value**

manticore.platforms.evm.**ceil32**(*x*)

manticore.platforms.evm.**concretized_args**(***policies*)
Make sure an EVM instruction has all of its arguments concretized according to provided policies.

Example decoration:

@concretized_args(size='ONE', address='') def LOG(self, address, size, *topics): ...

The above will make sure that the *size* parameter to LOG is Concretized when symbolic according to the 'ONE' policy and concretize *address* with the default policy.

**Parameters** **policies** – A kwargs list of argument names and their respective policies. Provide None or '' as policy to use default.

**Returns** A function decorator

manticore.platforms.evm.**to_signed**(*i*)

# CHAPTER 5

# Native

## 5.1 Platforms

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

**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 input
- **kwargs** – Forwarded to the Manticore constructor

**Returns** Manticore instance, initialized with a Decree State

**Return type** Manticore

**classmethod linux**(*path*, *argv=None*, *envp=None*, *entry_symbol=None*, *symbolic_files=None*, *concrete_start=''*, *pure_symbolic=False*, *stdin_size=None*, ***kwargs*)

Constructor for Linux binary analysis.

**Parameters**

- **path** (*str*) – Path to binary to analyze
- **argv** (*list[str]*) – Arguments to provide to the binary
- **envp** (*str*) – Environment to provide to the binary
- **entry_symbol** – Entry symbol to resolve to start execution
- **symbolic_files** (*list[str]*) – Filenames to mark as having symbolic input
- **concrete_start** (*str*) – Concrete stdin to use before symbolic input
- **stdin_size** (*int*) – symbolic stdin size to use

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

**Returns** Manticore instance, initialized with a Linux State

**Return type** Manticore

## 5.2 Linux

**class** manticore.platforms.linux.**SLinux**(*programs*, *argv=None*, *envp=None*, *symbolic_files=None*, *disasm='capstone'*, *pure_symbolic=False*)

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 chars are concretized. Symbolic files must have been defined before the call to *run()*.

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

## 5.3 Models

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

manticore.native.models.**isvariadic**(*model*)

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

**Returns** Whether *model* models a variadic function

**Return type** bool

manticore.native.models.**strcmp**(*state*, *s1*, *s2*)

strcmp symbolic model.

Algorithm: Walks from end of string (minimum offset to NULL in either string) to beginning building tree of ITEs each time either of the bytes at current offset is symbolic.

Points of Interest: - We've been building up a symbolic tree but then encounter two concrete bytes that differ. We can throw away the entire symbolic tree! - If we've been encountering concrete bytes that match at the end of the string as we walk forward, and then we encounter a pair where one is symbolic, we can forget about that 0 *ret* we've been tracking and just replace it with the symbolic subtraction of the two

**Parameters**

- **state** (State) – Current program state
- **s1** (*int*) – Address of string 1
- **s2** (*int*) – Address of string 2

**Returns** Symbolic strcmp result

**Return type** Expression or int

manticore.native.models.**strlen**(*state*, *s*)
strlen symbolic model.

Algorithm: Walks from end of string not including NULL building ITE tree when current byte is symbolic.

**Parameters**

- **state** (State) – current program state
- **s** (*int*) – Address of string

**Returns** Symbolic strlen result

**Return type** Expression or int

manticore.native.models.**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

## 5.4 State

**class** manticore.native.state.**State**(*constraints*, *platform*, ***kwargs*)

**cpu**
Current cpu state

**execute**()
Perform a single step on the current state

**invoke_model**(*model*)
Invokes a *model*. Modelling can be used to override a function in the target program with a custom implementation.

For more information on modelling see docs/models.rst

A *model* is a callable whose first argument is a *manticore.native.State* instance. If 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.f

**Parameters** **model** – callable, model to invoke

**mem**
Current virtual memory mappings

## 5.5 Cpu

**class** manticore.native.state.**State**(*constraints*, *platform*, ***kwargs*)

**cpu**
Current cpu state

**class** manticore.native.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]

**backup_emulate**(*insn*)

If we could not handle emulating an instruction, use Unicorn to emulate it.

**Parameters** **instruction** (*capstone.CsInsn*) – The instruction object to emulate

**canonical_registers**

Returns the list of all register names for this CPU.

**Return type** tuple

**Returns** the list of register names for this CPU.

**canonicalize_instruction_name**(*instruction*)

Get the semantic name of an instruction.

**concrete_emulate**(*insn*)

Start executing in Unicorn from this point until we hit a syscall or reach break_unicorn_at

**Parameters** **insn** (*capstone.CsInsn*) – The instruction object to emulate

**decode_instruction**(*pc*)

This will decode an instruction from memory pointed by *pc*

**Parameters** **pc** (*int*) – address of the instruction

**emulate**(*insn*)

Pick the right emulate function (maintains API compatiblity)

**Parameters** **insn** – single instruction to emulate/start emulation from

**emulate_until**(*target: int*)

Tells the CPU to set up a concrete unicorn emulator and use it to execute instructions until target is reached.

**Parameters** **target** – Where Unicorn should hand control back to Manticore. Set to 0 for all instructions.

**execute**()

Decode, and execute one instruction pointed by register PC

**icount**

**instruction**

**memory**

**pop_bytes**(*nbytes*, *force=False*)
Read *nbytes* from the stack, increment the stack pointer, and return data.

**Parameters**

- **nbytes** (*int*) – How many bytes to read
- **force** – whether to ignore memory permissions

**Returns** Data read from the stack

**pop_int**(*force=False*)
Read a value from the stack and increment the stack pointer.

**Parameters** **force** – whether to ignore memory permissions

**Returns** Value read

**push_bytes**(*data*, *force=False*)
Write *data* to the stack and decrement the stack pointer accordingly.

**Parameters**

- **data** (*str*) – Data to write
- **force** – whether to ignore memory permissions

**push_int**(*value*, *force=False*)
Decrement the stack pointer and write *value* to the stack.

**Parameters**

- **value** (*int*) – The value to write
- **force** – whether to ignore memory permissions

**Returns** New stack pointer

**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

**read_string**(*where*, *max_length=None*, *force=False*)
Read a NUL-terminated concrete buffer from memory. Stops reading at first symbolic byte.

**Parameters**

- **where** (*int*) – Address to read string from
- **max_length** (*int*) – The size in bytes to cap the string at, or None [default] for no limit.
- **force** – whether to ignore memory permissions

**Returns** string read

**Return type** str

**regfile**
The RegisterFile of this cpu

**render_instruction**(*insn=None*)

**render_register**(*reg_name*)

**render_registers**()

**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

**write_string**(*where*, *string*, *max_length=None*, *force=False*)
Writes a string to memory, appending a NULL-terminator at the end.

**Parameters**

- **where** (*int*) – Address to write the string to
- **string** (*str*) – The string to write to memory
- **max_length** (*int*) –

The size in bytes to cap the string at, or None [default] for no limit. This includes the NULL terminator.

**Parameters** **force** – whether to ignore memory permissions

## 5.6 Memory

**class** manticore.native.state.**State**(*constraints*, *platform*, ***kwargs*)

**mem**
Current virtual memory mappings

**class** manticore.native.memory.**SMemory**(*constraints*, *symbols=None*, **args*, ***kwargs*)
The symbolic memory manager. This class handles all virtual memory mappings and symbolic chunks.

**Todo** improve comments

**munmap**(*start*, *size*)
Deletes the mappings for the specified address range and causes further references to addresses within the range to generate invalid memory references.

**Parameters**

- **start** – the starting address to delete.
- **size** – the length of the unmapping.

**read**(*address*, *size*, *force=False*)
Read a stream of potentially symbolic bytes from a potentially symbolic address

**Parameters**

- **address** – Where to read from
- **size** – How many bytes
- **force** – Whether to ignore permissions

**Return type** list

**write**(*address*, *value*, *force=False*)
Write a value at address.

**Parameters**

- **address** (*int or long or Expression*) – The address at which to write
- **value** (*str or list*) – Bytes to write
- **force** – Whether to ignore permissions

## 5.7 State

**class** manticore.native.state.**State**(*constraints*, *platform*, ***kwargs*)

**cpu**
Current cpu state

**execute**()
Perform a single step on the current state

**invoke_model**(*model*)
Invokes a *model*. Modelling can be used to override a function in the target program with a custom implementation.

For more information on modelling see docs/models.rst

A *model* is a callable whose first argument is a *manticore.native.State* instance. If 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.f

**Parameters** **model** – callable, model to invoke

**mem**
Current virtual memory mappings

## 5.8 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.

## 5.9 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 6

# Plugins

## 6.1 Core

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

**did_fork_state_callback**(*self*, *new_state*, *expression*, *new_value*, *policy*)

**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*)

## 6.2 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*)

## 6.3 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*, *operators*)

**did_evm_write_memory_callback**(*self*, *offset*, *operators*)

**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*)

## 6.4 memory

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

**did_map_memory_callback(self, addr, size, perms, filename, offset, addr) # little confused**

**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*)

## 6.5 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*)

## 6.6 x86

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

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

CHAPTER 7

# 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.

## 7.1 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"

## 7.2 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']
```

## 7.3 “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 8

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## m

# Index

## Symbols



## A



## B



## C

## D

## E



## F



## G

## H



## I



## J



## K



## L



## M

## N



## O



## P



## R

## S



## T



## U



## V



## W