Manticore Documentation

Release 0.3.5

Trail of Bits

Dec 08, 2020

Contents:

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

CHAPTER 1

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 behaivor 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 2

Selecting a target contract

manticore-verifier needs to be pointed to a the target contract containing any number of property methods. The target contract is the entry point of the exploration. It needs to initilize 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 3

User accounts

You can specify what are the accounts used in the exploration. Normaly 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", "0x52525252525252525252"]
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 4

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 maximun number of transactions is reached the explorations ends. Properties that had not be 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 Maximun 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 pass.

```
--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 TestToken
```

f/

CHAPTER 5

ManticoreBase

class manticore.core.manticore.**ManticoreBase**(*initial_state*, *workspace_url=None*, *outputspace_url=None*, *introspection_plugin_type: type = <class 'manticore.core.plugin.IntrospectionAPIPlugin'>*, ***kwargs*)

__init__(*initial_state*, *workspace_url=None*, *outputspace_url=None*, *introspection_plugin_type: type = <class 'manticore.core.plugin.IntrospectionAPIPlugin'>*, ***kwargs*)

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().

```
          _save     +-------------+  _load
State  +----------> |  WORKSPACE  +----------> State
                    +-------------+
```

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

```
Initial
State
  |   +-+---{fork}-----+
  |   | |              |
  V   V V              |
+---------+        +---+----+      +------------+
|  READY  +------->|  BUSY  +----->| TERMINATED |
+---------+        +---+----+      +------------+
     |
     |                              +--------+
     +----------------------------->| KILLED |
                                    +--------+
```

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 prominent event system. A user can stop or cancel the exploration at any time. The unfinished states caught in this situation are simply moved to their own list for further user action. This is a final list.

Parameters

- **initial_state** – the initial root *State* object to start from
- **workspace_url** – workspace folder name
- **outputspace_url** – Folder to place final output. Defaults to workspace
- **kwargs** – other kwargs, e.g.

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

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

clear_ready_states()
Remove all states from the ready list

clear_snapshot()
Remove any saved states

clear_terminated_states()
Remove all states from the terminated list

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_all_states()
Total states count

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

classmethod from_saved_state(*filename: str*, **args*, ***kwargs*)
Creates a Manticore object starting from a serialized state on the disk.

Parameters

- **filename** – File to load the state from
- **args** – Arguments forwarded to the Manticore object
- **kwargs** – Keyword args forwarded to the Manticore object

Returns An instance of a subclass of ManticoreBase with the given initial state

goto_snapshot()
REMOVE current ready states and replace them with the saved states in a snapshot

introspect() → Dict[int, manticore.core.plugin.StateDescriptor]
Allows callers to view descriptors for each state

Returns the latest copy of the State Descriptor dict

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

is_main()
True if called from the main process/script Note: in "single" mode this is _most **likely**_ True

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_state(*state: Union[manticore.core.state.StateBase, int]*, *delete: bool = False*)

Kill a state. A state is moved from any list to the kill list or fully removed from secondary storage

Parameters

- **state** – a state
- **delete** – if true remove the state from the secondary storage

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

only_from_main_script() → Callable
Allows the decorated method to run only from the main manticore script

pretty_print_states(**_args*)
Calls pretty_print_state_descriptors on the current set of state descriptors

register_daemon(*callback: Callable*)
Allows the user to register a function that will be called at *ManticoreBase.run()* and can run in the background. Infinite loops are acceptable as it will be killed when Manticore exits. The provided function is passed a thread as an argument, with the current Manticore object available as thread.manticore.

Parameters **callback** – function to be called

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

run()
Runs analysis.

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

sync() → Callable
Synchronization decorator

take_snapshot()
Copy/Duplicate/backup all ready states and save it in a snapshot. If there is a snapshot already saved it will be overrwritten

unregister_plugin(*plugin: Union[str, manticore.core.plugin.Plugin]*)
Removes a plugin from manticore. No events should be sent to it after

static verbosity(*level*)
Sets global verbosity 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 6

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 7

States

7.1 Accessing

class manticore.core.manticore.**ManticoreBase**(*initial_state*, *workspace_url=None*, *outputspace_url=None*, *introspection_plugin_type: type = <class 'manticore.core.plugin.IntrospectionAPIPlugin'>*, ***kwargs*)

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

Notably the cancelled states are not included here.

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*.

7.2 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 limits the number of solutions returned to *maxcount* to avoid a blowup in the state space. **This means that if there are more than 'maxcount' feasible solutions, some states will be silently ignored.**

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*)
A version of solver_one_n for a single expression. See solve_one_n.

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

Parameters

- **exprs** – An iterable of manticore.core.smtlib.Expression
- **constrain** (*bool*) – If True, constrain expr to solved solution value

Returns Concrete value or a tuple of concrete values

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.

7.3 Inspecting

class manticore.core.plugin.**StateDescriptor**(*state_id: int*, *state_list: Optional[manticore.utils.enums.StateLists] = None*, *children: set = <factory>*, *last_update: datetime.datetime = <factory>*, *last_intermittent_update: Optional[datetime.datetime] = None*, *created_at: datetime.datetime = <factory>*, *status: manticore.utils.enums.StateStatus = <StateStatus.waiting_for_worker: 'waiting_for_worker'>*, *_old_status: Optional[manticore.utils.enums.StateStatus] = None*, *total_execs: Optional[int] = None*, *own_execs: Optional[int] = None*, *pc: Optional[Any] = None*, *field_updated_at: Dict[str, datetime.datetime] = <factory>*, *termination_msg: Optional[str] = None*)

Dataclass that tracks information about a State.

children = None
State IDs of any states that forked from this one

created_at = None
The time at which this state was created (or first detected, if the did_enque callback didn't fire for some reason)

field_updated_at = None
Dict mapping field names to the time that field was last updated

last_intermittent_update = None
The time at which the on_execution_intermittent callback was last applied to this state. This is when the PC and exec count get updated.

last_update = None
The time that any field of this Descriptor was last updated

own_execs = None
Number of executions that took place in this state alone, excluding its parents

pc = None
Last program counter (if set)

state_id = None
State ID Number

state_list = None
Which State List the state currently resides in (or None if it's been removed entirely)

status = 'waiting_for_worker'
What the state is currently doing (ie waiting for a worker, running, solving, etc.) See enums.StateStatus

termination_msg = None
Message attached to the TerminateState exception that ended this state

total_execs = None
Total number of instruction executions in this state, including those in its parents

CHAPTER 8

EVM

8.1 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)

8.2 Manager

class manticore.ethereum.**ManticoreEVM**(*plugins=None*, ***kwargs*)

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

all_sound_states

Iterator over all sound states. This tries to solve any symbolic imprecision added by unsound_symbolication and then iterates over the resultant set.

This is the recommended to iterate over resultant steas after an exploration that included unsound symbolication

completed_transactions

constrain(*constraint*)

contract_accounts

create_account(*balance=0*, *address=None*, *code=None*, *name=None*, *nonce=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)
- **nonce** – force a specific nonce
- **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*)

end_block()

finalize(*procs=None*, *only_alive_states=False*)
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** – force the number of local processes to use in the reporting
- **only_alive_states** (*bool*) – if True, killed states (revert/throw/txerror) do not generate testscases

generation. Uses global configuration constant by default

fix_unsound_all(*procs=None*)

Parameters **procs** – force the number of local processes to use

fix_unsound_symbolication(*state*)

fix_unsound_symbolication_fake(*state*)
This method goes through all the applied symbolic functions and tries to find a concrete matching set of pairs

fix_unsound_symbolication_sound(*state*)
This method goes through all the applied symbolic functions and tries to find a concrete matching set of pairs

generate_testcase(*state*, *message=''*, *only_if=None*, *name='user'*)
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)
```

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*

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

make_symbolic_address(**accounts*, *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*, *compile_args=None*)

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.SolidityMetadata] = None*)

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)

ready_sound_states

Iterator over sound ready states. This tries to solve any symbolic imprecision added by unsound_symbolication and then iterates over the resultant set.

This is the recommended way to iterate over the resultant states after an exploration that included unsound symbolication

register_detector(*d*)

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

run(***kwargs*)

Runs analysis.

solidity_create_contract(*source_code*, *owner*, *name=None*, *contract_name=None*, *libraries=None*, *balance=0*, *address=None*, *args=()*, *gas=None*, *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
- **compile_args** (*dict*) – crytic compile options #FIXME(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

start_block(*blocknumber=None*, *timestamp=None*, *difficulty=0*, *gaslimit=0*, *coinbase=None*)

transaction(*caller*, *address*, *value*, *data*, *gas=None*, *price=1*)
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
- **price** – gas unit price

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

8.3 EVM

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

class manticore.platforms.evm.**BlockHeader**(*blocknumber*, *timestamp*, *difficulty*, *gaslimit*, *coinbase*)

blocknumber
Alias for field number 0

coinbase
Alias for field number 4

difficulty
Alias for field number 2

gaslimit
Alias for field number 3

timestamp
Alias for field number 1

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=None*, *fork='istanbul'*, ***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

CHAINID()
Get current chainid.

EXTCODEHASH(*account*)
Get hash of code

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

SELFBALANCE()

SELFDESTRUCT_gas(*recipient*)

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

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

allocated

bytecode

change_last_result(*result*)

static check256int(*value*)

check_oog()

constraints

disassemble()

execute()

fail_if(*failed*)

gas

instruction
Current instruction pointed by self.pc

is_failed()

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*, **args*)

safe_mul(*a*, *b*)

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

pos(*pos*)

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*, *fork='istanbul'*, ***kwargs*)

account_exists(*address*)

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*)
Initiates a CREATE a contract account. Sends a transaction to initialize the contract. Do a world.run() after this to explore all _possible_ outputs

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*)

end_block(*block_reward=None*)

evmfork

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_block(*blocknumber=4370000*, *timestamp=1524785992*, *difficulty=512*, *gaslimit=2147483647*, *coinbase=0*)

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.
- **failed** – True if the transaction must fail

sub_from_balance(*address*, *value*)

sub_refund(*value*)

symbolic_function(*func*, *data*)
Get an unsound symbolication for function *func*

transaction(*address*, *price=0*, *data=''*, *caller=None*, *value=0*, *gas=2300*)
Initiates a CALL transaction on current state. Do a world.run() after this to explore all _possible_ outputs

transactions
Completed completed transaction

try_simplify_to_constant(*data*)

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*, *failed*)

address
Alias for field number 1

caller
Alias for field number 4

data
Alias for field number 3

failed
Alias for field number 7

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=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

exception manticore.platforms.evm.**Throw**

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

address

caller

concretize(*state*, *constrain=False*)

Parameters

- **state** – a manticore state
- **constrain** (*bool*) – If True, constrain expr to concretized value

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*, *used_gas=None*)

sort

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

used_gas

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.**globalfakesha3**(*data*)

manticore.platforms.evm.**globalsha3**(*data*)

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

CHAPTER 9

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 10

Web Assembly

10.1 ManticoreWASM

class manticore.wasm.manticore.**ManticoreWASM**(*path_or_state*, *env={}*, *sup_env={}*, *workspace_url=None*, *policy='random'*, ***kwargs*)

Manticore class for interacting with WASM, analagous to ManticoreNative or ManticoreEVM.

collect_returns(*n=1*)

Iterates over the terminated states and collects the top n values from the stack. Generally only used for testing.

Parameters **n** – Number of values to collect

Returns

A list of list of lists. > One list for each state

> One list for each n > The output from solver.get_all_values

default_invoke(*func_name: str = 'main'*)

Looks for a *main* function or *start* function and invokes it with symbolic arguments :param func_name: Optional name of function to look for

exported_functions = None

List of exported function names in the default module

finalize()

Finish a run and solve for test cases. Calls save_run_data

generate_testcase(*state*, *message='test'*, *name='test'*)

invoke(*name='main'*, *argv_generator=<function ManticoreWASM.<lambda>>*)

Maps the "invoke" command over all the ready states :param name: The function to invoke :param argv_generator: A function that takes the current state and returns a list of arguments

run(*timeout=None*)

Begins the Manticore run

Parameters **timeout** – number of seconds after which to kill execution

save_run_data()

10.2 WASM World

class manticore.platforms.wasm.**WASMWorld**(*filename*, *name='self'*, ***kwargs*)

Manages global environment for a WASM state. Analagous to EVMWorld.

advice = None

Stores concretized information used to advise execution of the next instruction.

constraints = None

Initial set of constraints

exec_for_test(*funcname*, *module=None*)

Helper method that simulates the evaluation loop without creating workers or states, forking, or concretizing symbolic values. Only used for concrete unit testing.

Parameters

- **funcname** – The name of the function to test
- **module** – The name of the module to test the function in (if not the default module)

Returns The top n items from the stack where n is the expected number of return values from the function

execute(*current_state*)

Tells the underlying ModuleInstance to execute a single WASM instruction. Raises TerminateState if there are no more instructions to execute, or if the instruction raises a Trap.

get_export(*export_name*, *mod_name=None*) → Union[manticore.wasm.structure.ProtoFuncInst, manticore.wasm.structure.TableInst, manticore.wasm.structure.MemInst, manticore.wasm.structure.GlobalInst, Callable, None]

Gets the export _instance_ for a given export & module name (basically just dereferences _get_export_addr into the store)

Parameters

- **export_name** – Name of the export to look for
- **mod_name** – Name of the module the export lives in

Returns The export itself

get_module_imports(*module*, *exec_start*, *stub_missing*) → List[Union[manticore.wasm.structure.FuncAddr, manticore.wasm.structure.TableAddr, manticore.wasm.structure.MemAddr, manticore.wasm.structure.GlobalAddr]]

Builds the list of imports that should be passed to the given module upon instantiation

Parameters

- **module** – The module to find the imports for
- **exec_start** – Whether to execute the start function of the module
- **stub_missing** – Whether to replace missing imports with stubs (TODO: symbolicate)

Returns List of addresses for the imports within the store

import_module(*module_name*, *exec_start*, *stub_missing*)

Collect all of the imports for the given module and instantiate it

Parameters

- **module_name** – module to import
- **exec_start** – whether to run the start functions automatically
- **stub_missing** – whether to replace missing imports with stubs

Returns None

instance

Returns the ModuleInstance for the first module registered

instantiate(*env_import_dict: Dict[str, Union[manticore.wasm.structure.ProtoFuncInst, manticore.wasm.structure.TableInst, manticore.wasm.structure.MemInst, manticore.wasm.structure.GlobalInst, Callable]], supplemental_env: Dict[str, Dict[str, Union[manticore.wasm.structure.ProtoFuncInst, manticore.wasm.structure.TableInst, manticore.wasm.structure.MemInst, manticore.wasm.structure.GlobalInst, Callable]]] = {}, exec_start=False, stub_missing=True*)

Prepares the underlying ModuleInstance for execution. Calls import_module under the hood, so this is probably the only import-y function you ever need to call externally.

TODO: stubbed imports should be symbolic

Parameters

- **env_import_dict** – Dict mapping strings to functions. Functions should accept the current ConstraintSet as the first argument.
- **supplemental_env** – Maps strings w/ module names to environment dicts using the same format as env_import_dict
- **exec_start** – Whether or not to automatically execute the *start* function, if it is set.
- **stub_missing** – Whether or not to replace missing imports with empty stubs

Returns None

instantiated = None

Prevents users from calling run without instantiating the module

invoke(*name='main', argv=[], module=None*)

Sets up the WASMWorld to run the function specified by *name* when *ManticoreWASM.run* is called

Parameters

- **name** – Name of the function to invoke
- **argv** – List of arguments to pass to the function. Should typically be I32, I64, F32, or F64
- **module** – name of a module to call the function in (if not the default module)

Returns None

module

Returns The first module registered

register_module(*name, filename_or_alias*)

Provide an explicit path to a WASM module so the importer will know where to find it

Parameters

- **name** – Module name to register the module under
- **filename_or_alias** – Name of the .wasm file that module lives in

Returns

set_env(*exports: Dict[str, Union[manticore.wasm.structure.ProtoFuncInst, manticore.wasm.structure.TableInst, manticore.wasm.structure.MemInst, manticore.wasm.structure.GlobalInst, Callable]], mod_name='env'*)
Manually insert exports into the global environment

Parameters

- **exports** – Dict mapping names to functions/tables/globals/memories
- **mod_name** – The name of the module these exports should fall under

stack = None
Stores numeric values, branch labels, and execution frames

store = None
Backing store for functions, memories, tables, and globals

manticore.platforms.wasm.**stub**(*arity*, *_state*, **args*)
Default function used for hostfunc calls when a proper import wasn't provided

10.3 Executor

class manticore.wasm.executor.**Executor**(**args*, ***kwargs*)
Contains execution semantics for all WASM instructions that don't involve control flow (and thus only need access to the store and the stack).

In lieu of annotating every single instruction with the relevant link to the docs, we direct you here: https://www.w3.org/TR/wasm-core-1/#a7-index-of-instructions

check_overflow(*expression*) → bool

check_zero_div(*expression*) → bool

current_memory(*store*, *stack*, *imm: manticore.wasm.types.CurGrowMemImm*)

dispatch(*inst*, *store*, *stack*)
Selects the correct semantics for the given instruction, and executes them

Parameters

- **inst** – the Instruction to execute
- **store** – the current Store
- **stack** – the current Stack

Returns the result of the semantic function, which is (probably) always None

drop(*store*, *stack*)

f32_abs(*store*, *stack*)

f32_add(*store*, *stack*)

f32_binary(*store*, *stack*, *op*, *rettype: type = <class 'manticore.wasm.types.I32'>*)

f32_ceil(*store*, *stack*)

f32_const(*store*, *stack*, *imm: manticore.wasm.types.F32ConstImm*)

f32_convert_s_i32(*store*, *stack*)

f32_convert_s_i64(*store*, *stack*)

f32_convert_u_i32 (*store*, *stack*)

f32_convert_u_i64 (*store*, *stack*)

f32_copysign (*store*, *stack*)

f32_demote_f64 (*store*, *stack*)

f32_div (*store*, *stack*)

f32_eq (*store*, *stack*)

f32_floor (*store*, *stack*)

f32_ge (*store*, *stack*)

f32_gt (*store*, *stack*)

f32_le (*store*, *stack*)

f32_load (*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

f32_lt (*store*, *stack*)

f32_max (*store*, *stack*)

f32_min (*store*, *stack*)

f32_mul (*store*, *stack*)

f32_ne (*store*, *stack*)

f32_nearest (*store*, *stack*)

f32_neg (*store*, *stack*)

f32_reinterpret_i32 (*store*, *stack*)

f32_sqrt (*store*, *stack*)

f32_store (*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

f32_sub (*store*, *stack*)

f32_trunc (*store*, *stack*)

f32_unary (*store*, *stack*, *op*, *rettype: type = <class 'manticore.wasm.types.I32'>*)

f64_abs (*store*, *stack*)

f64_add (*store*, *stack*)

f64_binary (*store*, *stack*, *op*, *rettype: type = <class 'manticore.wasm.types.I32'>*)

f64_ceil (*store*, *stack*)

f64_const (*store*, *stack*, *imm: manticore.wasm.types.F64ConstImm*)

f64_convert_s_i32 (*store*, *stack*)

f64_convert_s_i64 (*store*, *stack*)

f64_convert_u_i32 (*store*, *stack*)

f64_convert_u_i64 (*store*, *stack*)

f64_copysign (*store*, *stack*)

f64_div (*store*, *stack*)

f64_eq (*store*, *stack*)

f64_floor(*store*, *stack*)

f64_ge(*store*, *stack*)

f64_gt(*store*, *stack*)

f64_le(*store*, *stack*)

f64_load(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

f64_lt(*store*, *stack*)

f64_max(*store*, *stack*)

f64_min(*store*, *stack*)

f64_mul(*store*, *stack*)

f64_ne(*store*, *stack*)

f64_nearest(*store*, *stack*)

f64_neg(*store*, *stack*)

f64_promote_f32(*store*, *stack*)

f64_reinterpret_i64(*store*, *stack*)

f64_sqrt(*store*, *stack*)

f64_store(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

f64_sub(*store*, *stack*)

f64_trunc(*store*, *stack*)

f64_unary(*store*, *stack*, *op*, *rettype: type = <class 'manticore.wasm.types.F64'>*)

float_load(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*, *ty: type*)

float_push_compare_return(*stack*, *v*, *rettype=<class 'manticore.wasm.types.I32'>*)

float_store(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*, *ty: type*, *n=None*)

get_global(*store*, *stack*, *imm: manticore.wasm.types.GlobalVarXsImm*)

get_local(*store*, *stack*, *imm: manticore.wasm.types.LocalVarXsImm*)

grow_memory(*store*, *stack*, *imm: manticore.wasm.types.CurGrowMemImm*)

i32_add(*store*, *stack*)

i32_and(*store*, *stack*)

i32_clz(*store*, *stack*)

i32_const(*store*, *stack*, *imm: manticore.wasm.types.I32ConstImm*)

i32_ctz(*store*, *stack*)

i32_div_s(*store*, *stack*)

i32_div_u(*store*, *stack*)

i32_eq(*store*, *stack*)

i32_eqz(*store*, *stack*)

i32_ge_s(*store*, *stack*)

i32_ge_u(*store*, *stack*)

i32_gt_s(*store*, *stack*)

i32_gt_u(*store*, *stack*)

i32_le_s(*store*, *stack*)

i32_le_u(*store*, *stack*)

i32_load(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_load16_s(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_load16_u(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_load8_s(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_load8_u(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_lt_s(*store*, *stack*)

i32_lt_u(*store*, *stack*)

i32_mul(*store*, *stack*)

i32_ne(*store*, *stack*)

i32_or(*store*, *stack*)

i32_popcnt(*store*, *stack*)

i32_reinterpret_f32(*store*, *stack*)

i32_rem_s(*store*, *stack*)

i32_rem_u(*store*, *stack*)

i32_rotl(*store*, *stack*)

i32_rotr(*store*, *stack*)

i32_shl(*store*, *stack*)

i32_shr_s(*store*, *stack*)

i32_shr_u(*store*, *stack*)

i32_store(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_store16(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_store8(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i32_sub(*store*, *stack*)

i32_trunc_s_f32(*store*, *stack*)

i32_trunc_s_f64(*store*, *stack*)

i32_trunc_u_f32(*store*, *stack*)

i32_trunc_u_f64(*store*, *stack*)

i32_wrap_i64(*store*, *stack*)

i32_xor(*store*, *stack*)

i64_add(*store*, *stack*)

i64_and(*store*, *stack*)

i64_clz(*store*, *stack*)

i64_const(*store*, *stack*, *imm: manticore.wasm.types.I64ConstImm*)

i64_ctz(*store*, *stack*)

i64_div_s(*store*, *stack*)

i64_div_u(*store*, *stack*)

i64_eq(*store*, *stack*)

i64_eqz(*store*, *stack*)

i64_extend_s_i32(*store*, *stack*)

i64_extend_u_i32(*store*, *stack*)

i64_ge_s(*store*, *stack*)

i64_ge_u(*store*, *stack*)

i64_gt_s(*store*, *stack*)

i64_gt_u(*store*, *stack*)

i64_le_s(*store*, *stack*)

i64_le_u(*store*, *stack*)

i64_load(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_load16_s(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_load16_u(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_load32_s(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_load32_u(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_load8_s(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_load8_u(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_lt_s(*store*, *stack*)

i64_lt_u(*store*, *stack*)

i64_mul(*store*, *stack*)

i64_ne(*store*, *stack*)

i64_or(*store*, *stack*)

i64_popcnt(*store*, *stack*)

i64_reinterpret_f64(*store*, *stack*)

i64_rem_s(*store*, *stack*)

i64_rem_u(*store*, *stack*)

i64_rotl(*store*, *stack*)

i64_rotr(*store*, *stack*)

i64_shl(*store*, *stack*)

i64_shr_s(*store*, *stack*)

i64_shr_u(*store*, *stack*)

i64_store(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_store16(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_store32(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_store8(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*)

i64_sub(*store*, *stack*)

i64_trunc_s_f32(*store*, *stack*)

i64_trunc_s_f64(*store*, *stack*)

i64_trunc_u_f32(*store*, *stack*)

i64_trunc_u_f64(*store*, *stack*)

i64_xor(*store*, *stack*)

int_load(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*, *ty: type*, *size: int*, *signed: bool*)

int_store(*store*, *stack*, *imm: manticore.wasm.types.MemoryImm*, *ty: type*, *n=None*)

nop(*store*, *stack*)

select(*store*, *stack*)

set_global(*store*, *stack*, *imm: manticore.wasm.types.GlobalVarXsImm*)

set_local(*store*, *stack*, *imm: manticore.wasm.types.LocalVarXsImm*)

tee_local(*store*, *stack*, *imm: manticore.wasm.types.LocalVarXsImm*)

unreachable(*store*, *stack*)

manticore.wasm.executor.**operator_ceil**(*a*)

manticore.wasm.executor.**operator_div**(*a*, *b*)

manticore.wasm.executor.**operator_floor**(*a*)

manticore.wasm.executor.**operator_max**(*a*, *b*)

manticore.wasm.executor.**operator_min**(*a*, *b*)

manticore.wasm.executor.**operator_nearest**(*a*)

manticore.wasm.executor.**operator_trunc**(*a*)

10.4 Module Structure

class manticore.wasm.structure.**Activation**(*arity*, *frame*, *expected_block_depth=0*)

Pushed onto the stack with each function invocation to keep track of the call stack

https://www.w3.org/TR/wasm-core-1/#activations-and-frames%E2%91%A0

arity = None

The expected number of return values from the function call associated with the underlying frame

expected_block_depth = None

Internal helper used to track the expected block depth when we exit this label

frame = None

The nested frame

class manticore.wasm.structure.**Addr**

class manticore.wasm.structure.**AtomicStack**(*parent: manticore.wasm.structure.Stack*)

Allows for the rolling-back of the stack in the event of a concretization exception. Inherits from Stack so that the types will be correct, but never calls *super*. Provides a context manager that will intercept Concretization Exceptions before raising them.

class PopItem(*val: Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec, manticore.wasm.structure.Label, manticore.wasm.structure.Activation]*)

class PushItem

empty()

Returns True if the stack is empty, otherwise False

find_type(*t: type*)

Parameters **t** – The type to look for

Returns The depth of the first value of type t

get_frame() → manticore.wasm.structure.Activation

Returns the topmost frame (Activation) on the stack

get_nth(*t: type*, *n: int*)

Parameters

- **t** – type to look for
- **n** – number to look for

Returns the nth item of type t from the top of the stack, or None

has_at_least(*t: type*, *n: int*)

Parameters

- **t** – type to look for
- **n** – number to look for

Returns whether the stack contains at least n values of type t

has_type_on_top(*t: Union[type, Tuple[type, ...]], n: int*)

Asserts that the stack has at least n values of type t or type BitVec on the top

Parameters

- **t** – type of value to look for (Bitvec is always included as an option)
- **n** – Number of values to check

Returns True

peek()

Returns the item on top of the stack (without removing it)

pop() → Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec, manticore.wasm.structure.Label, manticore.wasm.structure.Activation]

Pop a value from the stack

Returns the popped value

push(*val: Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec, manticore.wasm.structure.Label, manticore.wasm.structure.Activation]*) → None
Push a value to the stack

Parameters **val** – The value to push

Returns None

rollback()

exception manticore.wasm.structure.**ConcretizeCondition**(*message: str, condition: manticore.core.smtlib.expression.Bool, current_advice: Optional[List[bool]], **kwargs*)
Tells Manticore to concretize a condition required to direct execution.

class manticore.wasm.structure.**Data**(*data: manticore.wasm.types.MemIdx, offset: List[manticore.wasm.types.Instruction], init: List[int]*)
Vector of bytes that initializes part of a memory

https://www.w3.org/TR/wasm-core-1/#data-segments%E2%91%A0

data = None
Which memory to put the data in. Currently only supports 0

init = None
List of bytes to copy into the memory

offset = None
WASM instructions that calculate offset into the memory

class manticore.wasm.structure.**Elem**(*table: manticore.wasm.types.TableIdx, offset: List[manticore.wasm.types.Instruction], init: List[manticore.wasm.types.FuncIdx]*)
List of functions to initialize part of a table

https://www.w3.org/TR/wasm-core-1/#element-segments%E2%91%A0

init = None
list of function indices that get copied into the table

offset = None
WASM instructions that calculate an offset to add to the table index

table = None
Which table to initialize

class manticore.wasm.structure.**Export**(*name: manticore.wasm.types.Name, desc: Union[manticore.wasm.types.FuncIdx, manticore.wasm.types.TableIdx, manticore.wasm.types.MemIdx, manticore.wasm.types.GlobalIdx]*)
Something the module exposes to the outside world once it's been instantiated

https://www.w3.org/TR/wasm-core-1/#exports%E2%91%A0

desc = None
Whether this is a function, table, memory, or global

name = None
The name of the thing we're exporting

class manticore.wasm.structure.**ExportInst**(*name: manticore.wasm.types.Name, value: Union[manticore.wasm.structure.FuncAddr, manticore.wasm.structure.TableAddr, manticore.wasm.structure.MemAddr, manticore.wasm.structure.GlobalAddr]*)

Runtime representation of any thing that can be exported

https://www.w3.org/TR/wasm-core-1/#export-instances%E2%91%A0

name = None
The name to export under

value = None
FuncAddr, TableAddr, MemAddr, or GlobalAddr

class manticore.wasm.structure.**Frame**(*locals: List[Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec]], module: manticore.wasm.structure.ModuleInstance*)

Holds more call data, nested inside an activation (for reasons I don't understand)

https://www.w3.org/TR/wasm-core-1/#activations-and-frames%E2%91%A0

locals = None
The values of the local variables for this function call

module = None
A reference to the parent module instance in which the function call was made

class manticore.wasm.structure.**FuncAddr**

class manticore.wasm.structure.**FuncInst**(*type: manticore.wasm.types.FunctionType, module: manticore.wasm.structure.ModuleInstance, code: manticore.wasm.structure.Function*)

Instance type for WASM functions

class manticore.wasm.structure.**Function**(*type: manticore.wasm.types.TypeIdx, locals: List[type], body: List[manticore.wasm.types.Instruction]*)

A WASM Function

https://www.w3.org/TR/wasm-core-1/#functions%E2%91%A0

allocate(*store: manticore.wasm.structure.Store, module: manticore.wasm.structure.ModuleInstance*) → manticore.wasm.structure.FuncAddr
https://www.w3.org/TR/wasm-core-1/#functions%E2%91%A5

Parameters

- **store** – Destination Store that we'll insert this Function into after allocation
- **module** – The module containing the type referenced by self.type

Returns The address of this within *store*

body = None
Sequence of WASM instructions, should leave the appropriate type on the stack

locals = None
Vector of mutable local variables (and their types)

type = None
The index of a type defined in the module that corresponds to this function's type signature

class manticore.wasm.structure.**Global**(*type: manticore.wasm.types.GlobalType, init: List[manticore.wasm.types.Instruction]*)

A global variable of a given type

https://www.w3.org/TR/wasm-core-1/#globals%E2%91%A0

allocate(*store: manticore.wasm.structure.Store, val: Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec]*) → manticore.wasm.structure.GlobalAddr

https://www.w3.org/TR/wasm-core-1/#globals%E2%91%A5

Parameters

- **store** – Destination Store that we'll insert this Global into after allocation
- **val** – The initial value of the new global

Returns The address of this within *store*

init = None

A (constant) sequence of WASM instructions that calculates the value for the global

type = None

The type of the variable

class manticore.wasm.structure.**GlobalAddr**

class manticore.wasm.structure.**GlobalInst**(*value: Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec], mut: bool*)

Instance of a global variable. Stores the value (calculated from evaluating a Global.init) and the mutable flag (taken from GlobalType.mut)

https://www.w3.org/TR/wasm-core-1/#global-instances%E2%91%A0

mut = None

Whether the global can be modified

value = None

The actual value of this global

class manticore.wasm.structure.**HostFunc**(*type: manticore.wasm.types.FunctionType, hostcode: function*)

Instance type for native functions that have been provided via import

allocate(*store: manticore.wasm.structure.Store, functype: manticore.wasm.types.FunctionType, host_func: function*) → manticore.wasm.structure.FuncAddr

Currently not needed.

https://www.w3.org/TR/wasm-core-1/#host-functions%E2%91%A2

hostcode = None

the native function. Should accept ConstraintSet as the first argument

class manticore.wasm.structure.**Import**(*module: manticore.wasm.types.Name, name: manticore.wasm.types.Name, desc: Union[manticore.wasm.types.TypeIdx, manticore.wasm.types.TableType, manticore.wasm.types.LimitType, manticore.wasm.types.GlobalType]*)

Something imported from another module (or the environment) that we need to instantiate a module

https://www.w3.org/TR/wasm-core-1/#imports%E2%91%A0

desc = None
Specifies whether this is a function, table, memory, or global

module = None
The name of the module we're importing from

name = None
The name of the thing we're importing

class manticore.wasm.structure.**Label**(*arity: int, instr: List[manticore.wasm.types.Instruction]*)
A branch label that can be pushed onto the stack and then jumped to

https://www.w3.org/TR/wasm-core-1/#labels%E2%91%A0

arity = None
the number of values this branch expects to read from the stack

instr = None
The sequence of instructions to execute if we branch to this label

class manticore.wasm.structure.**MemAddr**

class manticore.wasm.structure.**MemInst**(*starting_data*, *max=None*, **args*, ***kwargs*)
Runtime representation of a memory. As with tables, if you're dealing with a memory at runtime, it's probably a MemInst. Currently doesn't support any sort of symbolic indexing, although you can read and write symbolic bytes using smtlib. There's a minor quirk where uninitialized data is stored as bytes, but smtlib tries to convert concrete data into ints. That can cause problems if you try to read from the memory directly (without using smtlib) but shouldn't break any of the built-in WASM instruction implementations.

Memory in WASM is broken up into 65536-byte pages. All pages behave the same way, but note that operations that deal with memory size do so in terms of pages, not bytes.

TODO: We should implement some kind of symbolic memory model

https://www.w3.org/TR/wasm-core-1/#memory-instances%E2%91%A0

dump()

grow(*n: int*) → bool
Adds n blank pages to the current memory

See: https://www.w3.org/TR/wasm-core-1/#grow-mem

Parameters **n** – The number of pages to attempt to add

Returns True if the operation succeeded, otherwise False

max = None
Optional maximum number of pages the memory can contain

npages

read_bytes(*base: int*, *size: int*) → List[Union[int, bytes]]
Reads bytes from memory

Parameters

- **base** – Address to read from
- **size** – number of bytes to read

Returns List of bytes

read_int(*base: int*, *size: int = 32*) → int
Reads bytes from memory and combines them into an int

Parameters

- **base** – Address to read the int from
- **size** – Size of the int (in bits)

Returns The int in question

write_bytes(*base: int, data: Union[str, Sequence[int], Sequence[bytes]]*)
Writes a stream of bytes into memory

Parameters

- **base** – Index to start writing at
- **data** – Data to write

write_int(*base: int, expression: Union[manticore.core.smtlib.expression.Expression, int], size: int = 32*)
Writes an integer into memory.

Parameters

- **base** – Index to write at
- **expression** – integer to write
- **size** – Optional size of the integer

class manticore.wasm.structure.**Memory**(*type: manticore.wasm.types.LimitType*)
Big chunk o' raw bytes

https://www.w3.org/TR/wasm-core-1/#memories%E2%91%A0

allocate(*store: manticore.wasm.structure.Store*) → manticore.wasm.structure.MemAddr
https://www.w3.org/TR/wasm-core-1/#memories%E2%91%A5

Parameters **store** – Destination Store that we'll insert this Memory into after allocation

Returns The address of this within *store*

type = None
secretly a LimitType that specifies how big or small the memory can be

class manticore.wasm.structure.**Module**
Internal representation of a WASM Module

data

elem

exports

funcs

function_names

get_funcnames() → List[manticore.wasm.types.Name]

globals

imports

classmethod load(*filename: str*)
Converts a WASM module in binary format into Python types that Manticore can understand

Parameters **filename** – name of the WASM module

Returns Module

local_names

mems

start
https://www.w3.org/TR/wasm-core-1/#start-function%E2%91%A0

tables

types

class manticore.wasm.structure.**ModuleInstance**(*constraints=None*)
Runtime instance of a module. Stores function types, list of addresses within the store, and exports. In this implementation, it's also responsible for managing the instruction queue and executing control-flow instructions.

https://www.w3.org/TR/wasm-core-1/#module-instances%E2%91%A0

allocate(*store: manticore.wasm.structure.Store, module: manticore.wasm.structure.Module, extern_vals: List[Union[manticore.wasm.structure.FuncAddr, manticore.wasm.structure.TableAddr, manticore.wasm.structure.MemAddr, manticore.wasm.structure.GlobalAddr]], values: List[Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec]]*)
Inserts imports into the store, then creates and inserts function instances, table instances, memory instances, global instances, and export instances.

https://www.w3.org/TR/wasm-core-1/#allocation%E2%91%A0 https://www.w3.org/TR/wasm-core-1/#modules%E2%91%A6

Parameters

- **store** – The Store to put all of the allocated subcomponents in
- **module** – Tne Module containing all the items to allocate
- **extern_vals** – Imported values
- **values** – precalculated global values

block(*store: manticore.wasm.structure.Store, stack: manticore.wasm.structure.Stack, ret_type: List[type], insts: List[manticore.wasm.types.Instruction]*)
Execute a block of instructions. Creates a label with an empty continuation and the proper arity, then enters the block of instructions with that label.

https://www.w3.org/TR/wasm-core-1/#exec-block

Parameters

- **ret_type** – List of expected return types for this block. Really only need the arity
- **insts** – Instructions to execute

br(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*, *label_depth: int*)
Branch to the ‘label_depth‘th label deep on the stack

https://www.w3.org/TR/wasm-core-1/#exec-br

br_if(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*, *imm: manticore.wasm.types.BranchImm*)
Perform a branch if the value on top of the stack is nonzero

https://www.w3.org/TR/wasm-core-1/#exec-br-if

br_table(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*, *imm: manticore.wasm.types.BranchTableImm*)
Branch to the nth label deep on the stack where n is found by looking up a value in a table given by the immediate, indexed by the value on top of the stack.

https://www.w3.org/TR/wasm-core-1/#exec-br-table

call(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*, *imm: manticore.wasm.types.CallImm*)
Invoke the function at the address in the store given by the immediate.

https://www.w3.org/TR/wasm-core-1/#exec-call

call_indirect(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*, *imm: manticore.wasm.types.CallIndirectImm*)
A function call, but with extra steps. Specifically, you find the index of the function to call by looking in the table at the index given by the immediate.

https://www.w3.org/TR/wasm-core-1/#exec-call-indirect

else_(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*)
Marks the end of the first block of an if statement. Typically, *if* blocks look like: *if <instructions> else <instructions> end*. That's not always the case. See: https://webassembly.github.io/spec/core/text/instructions.html#abbreviations

end(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*)
Marks the end of an instruction block or function

enter_block(*insts*, *label: manticore.wasm.structure.Label*, *stack: manticore.wasm.structure.Stack*)
Push the instructions for the next block to the queue and bump the block depth number

https://www.w3.org/TR/wasm-core-1/#exec-instr-seq-enter

Parameters

- **insts** – Instructions for this block
- **label** – Label referencing the continuation of this block
- **stack** – The execution stack (where we push the label)

exec_expression(*store: manticore.wasm.structure.Store, stack: manticore.wasm.structure.Stack, expr: List[manticore.wasm.types.Instruction]*)
Pushes the given expression to the stack, calls exec_instruction until there are no more instructions to exec, then returns the top value on the stack. Used during initialization to calculate global values, memory offsets, element offsets, etc.

Parameters **expr** – The expression to execute

Returns The result of the expression

exec_instruction(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.Stack*, *advice: Optional[List[bool]] = None*, *current_state=None*) → bool
The core instruction execution function. Pops an instruction from the queue, then dispatches it to the Executor if it's a numeric instruction, or executes it internally if it's a control-flow instruction.

Parameters **store** – The execution Store to use, passed in from the parent WASMWorld. This is passed to almost all

instruction implementations, but for brevity's sake, it's only explicitly documented here.

Parameters **stack** – The execution Stack to use, likewise passed in from the parent WASM-World and only documented here,

despite being passed to all the instruction implementations.

Parameters **advice** – A list of concretized conditions to advice execution of the instruction.

Returns True if execution succeeded, False if there are no more instructions to execute

executor
Contains instruction implementations for all non-control-flow instructions

exit_block(*stack: manticore.wasm.structure.Stack*)
Cleans up after execution of a code block.

https://www.w3.org/TR/wasm-core-1/#exiting–hrefsyntax-instrmathitinstrast-with-label–l

exit_function(*stack: manticore.wasm.structure.AtomicStack*)
Discards the current frame, allowing execution to return to the point after the call

https://www.w3.org/TR/wasm-core-1/#returning-from-a-function%E2%91%A0

export_map
Maps the names of exports to their index in the list of exports

exports
Stores records of everything exported by this module

extract_block(*partial_list: Deque[manticore.wasm.types.Instruction]*) → Deque[manticore.wasm.types.Instruction]
Recursively extracts blocks from a list of instructions, similar to self.look_forward. The primary difference is that this version takes a list of instructions to operate over, instead of popping instructions from the instruction queue.

Parameters **partial_list** – List of instructions to extract the block from

Returns The extracted block

funcaddrs
Stores the *indices* of functions within the store

function_names
Stores names of store functions, if available

get_export(*name: str*, *store: manticore.wasm.structure.Store*) → Union[manticore.wasm.structure.ProtoFuncInst, manticore.wasm.structure.TableInst, manticore.wasm.structure.MemInst, manticore.wasm.structure.GlobalInst, Callable]
Retrieves a value exported by this module instance from store

Parameters

- **name** – The name of the exported value to get
- **store** – The current execution store (where the export values live)

Returns The value of the export

get_export_address(*name: str*) → Union[manticore.wasm.structure.FuncAddr, manticore.wasm.structure.TableAddr, manticore.wasm.structure.MemAddr, manticore.wasm.structure.GlobalAddr]
Retrieves the address of a value exported by this module within the store

Parameters **name** – The name of the exported value to get

Returns The address of the desired export

globaladdrs
Stores the indices of globals

if_(*store: manticore.wasm.structure.Store, stack: manticore.wasm.structure.AtomicStack, ret_type: List[type]*)
Brackets two nested sequences of instructions. If the value on top of the stack is nonzero, enter the first block. If not, enter the second.

https://www.w3.org/TR/wasm-core-1/#exec-if

instantiate(*store: manticore.wasm.structure.Store, module: manticore.wasm.structure.Module, extern_vals: List[Union[manticore.wasm.structure.FuncAddr, manticore.wasm.structure.TableAddr, manticore.wasm.structure.MemAddr, manticore.wasm.structure.GlobalAddr]], exec_start: bool = False*)
Type checks the module, evaluates globals, performs allocation, and puts the element and data sections into their proper places. Optionally calls the start function _outside_ of a symbolic context if exec_start is true.

https://www.w3.org/TR/wasm-core-1/#instantiation%E2%91%A1

Parameters

- **store** – The store to place the allocated contents in
- **module** – The WASM Module to instantiate in this instance
- **extern_vals** – Imports needed to instantiate the module
- **exec_start** – whether or not to execute the start section (if present)

instantiated = None
Prevents the user from invoking functions before instantiation

invoke(*stack: manticore.wasm.structure.Stack, funcaddr: manticore.wasm.structure.FuncAddr, store: manticore.wasm.structure.Store, argv: List[Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec]]*)
Invocation wrapper. Checks the function type, pushes the args to the stack, and calls _invoke_inner. Unclear why the spec separates the two procedures, but I've tried to implement it as close to verbatim as possible.

Note that this doesn't actually _run_ any code. It just sets up the instruction queue so that when you call ‘exec_instruction, it'll actually have instructions to execute.

https://www.w3.org/TR/wasm-core-1/#invocation%E2%91%A1

Parameters

- **funcaddr** – Address (in Store) of the function to call
- **argv** – Arguments to pass to the function. Can be BitVecs or Values

invoke_by_name(*name: str*, *stack*, *store*, *argv*)
Iterates over the exports, attempts to find the function specified by *name*. Calls *invoke* with its FuncAddr, passing argv

Parameters

- **name** – Name of the function to look for
- **argv** – Arguments to pass to the function. Can be BitVecs or Values

local_names
Stores names of local variables, if available

look_forward(**opcodes*) → List[manticore.wasm.types.Instruction]
Pops contents of the instruction queue until it finds an instruction with an opcode in the argument *opcodes. Used to find the end of a code block in the flat instruction queue. For this reason, it calls itself recursively (looking for the *end* instruction) if it encounters a *block*, *loop*, or *if* instruction.

Parameters **opcodes** – Tuple of instruction opcodes to look for

Returns The list of instructions popped before encountering the target instruction.

loop(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*, *loop_inst*)
Enter a loop block. Creates a label with a copy of the loop as a continuation, then enters the loop instructions with that label.

https://www.w3.org/TR/wasm-core-1/#exec-loop

Parameters **loop_inst** – The current insrtuction

memaddrs
Stores the indices of memories (at time of writing, WASM only allows one memory)

push_instructions(*insts: List[manticore.wasm.types.Instruction]*)
Pushes instructions into the instruction queue. :param insts: Instructions to push

reset_internal()
Empties the instruction queue and clears the block depths

return_(*store: manticore.wasm.structure.Store*, *stack: manticore.wasm.structure.AtomicStack*)
Return from the function (ie branch to the outermost block)

https://www.w3.org/TR/wasm-core-1/#exec-return

tableaddrs
Stores the indices of tables

types
Stores the type signatures of all the functions

manticore.wasm.structure.**PAGESIZE = 65536**
Size of a standard WASM memory page

class manticore.wasm.structure.**ProtoFuncInst**(*type: manticore.wasm.types.FunctionType*)
Groups FuncInst and HostFuncInst into the same category

type = None
The type signature of this function

class manticore.wasm.structure.**Stack**(*init_data=None*)
Stores the execution stack & provides helper methods

https://www.w3.org/TR/wasm-core-1/#stack%E2%91%A0

data = None
Underlying datastore for the "stack"

empty() → bool

Returns True if the stack is empty, otherwise False

find_type(*t: type*) → Optional[int]

Parameters **t** – The type to look for

Returns The depth of the first value of type t

get_frame() → manticore.wasm.structure.Activation

Returns the topmost frame (Activation) on the stack

get_nth(*t: type*, *n: int*) → Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec, manticore.wasm.structure.Label, manticore.wasm.structure.Activation, None]

Parameters

- **t** – type to look for
- **n** – number to look for

Returns the nth item of type t from the top of the stack, or None

has_at_least(*t: type*, *n: int*) → bool

Parameters

- **t** – type to look for
- **n** – number to look for

Returns whether the stack contains at least n values of type t

has_type_on_top(*t: Union[type, Tuple[type, ...]], n: int*)
Asserts that the stack has at least n values of type t or type BitVec on the top

Parameters

- **t** – type of value to look for (Bitvec is always included as an option)
- **n** – Number of values to check

Returns True

peek() → Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec, manticore.wasm.structure.Label, manticore.wasm.structure.Activation, None]

Returns the item on top of the stack (without removing it)

pop() → Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec, manticore.wasm.structure.Label, manticore.wasm.structure.Activation]
Pop a value from the stack

Returns the popped value

push(*val: Union[manticore.wasm.types.I32, manticore.wasm.types.I64, manticore.wasm.types.F32, manticore.wasm.types.F64, manticore.core.smtlib.expression.BitVec, manticore.wasm.structure.Label, manticore.wasm.structure.Activation]*) → None
Push a value to the stack

Parameters **val** – The value to push

Returns None

class manticore.wasm.structure.**Store**
Implementation of the WASM store. Nothing fancy here, just collects lists of functions, tables, memories, and globals. Because the store is not atomic, instructions SHOULD NOT make changes to the Store or any of its contents (including memories and global variables) before raising a Concretize exception.

https://www.w3.org/TR/wasm-core-1/#store%E2%91%A0

funcs

globals

mems

tables

class manticore.wasm.structure.**Table**(*type: manticore.wasm.types.TableType*)
Vector of opaque values of type self.type

https://www.w3.org/TR/wasm-core-1/#tables%E2%91%A0

allocate(*store: manticore.wasm.structure.Store*) → manticore.wasm.structure.TableAddr
https://www.w3.org/TR/wasm-core-1/#tables%E2%91%A5

Parameters **store** – Destination Store that we'll insert this Table into after allocation

Returns The address of this within *store*

type = None
union of a limit and a type (currently only supports funcref)s

class manticore.wasm.structure.**TableAddr**

class manticore.wasm.structure.**TableInst**(*elem: List[Optional[manticore.wasm.structure.FuncAddr]], max: Optional[manticore.wasm.types.U32]*)
Runtime representation of a table. Remember that the Table type stores the type of the data contained in the table and basically nothing else, so if you're dealing with a table at runtime, it's probably a TableInst. The WASM spec has a lot of similar-sounding names for different versions of one thing.

https://www.w3.org/TR/wasm-core-1/#table-instances%E2%91%A0

elem = None
A list of FuncAddrs (any of which can be None) that point to funcs in the Store

max = None
Optional maximum size of the table

manticore.wasm.structure.**strip_quotes**(*rough_name: str*) → manticore.wasm.types.Name
For some reason, the parser returns the function names with quotes around them

Parameters **rough_name** –

Returns

10.5 Types

class manticore.wasm.types.**BlockImm**(*sig: int*)

class manticore.wasm.types.**BranchImm**(*relative_depth: manticore.wasm.types.U32*)

class manticore.wasm.types.**BranchTableImm**(*target_count: manticore.wasm.types.U32, target_table: List[manticore.wasm.types.U32], default_target: manticore.wasm.types.U32*)

class manticore.wasm.types.**CallImm**(*function_index: manticore.wasm.types.U32*)

class manticore.wasm.types.**CallIndirectImm**(*type_index: manticore.wasm.types.U32, reserved: manticore.wasm.types.U32*)

exception manticore.wasm.types.**ConcretizeStack**(*depth: int, ty: type, message: str, expression, policy=None, **kwargs*)
Tells Manticore to concretize the value *depth* values from the end of the stack.

class manticore.wasm.types.**CurGrowMemImm**(*reserved: bool*)

manticore.wasm.types.**ExternType = typing.Union[manticore.wasm.types.FunctionType, manticore**
https://www.w3.org/TR/wasm-core-1/#external-types%E2%91%A0

class manticore.wasm.types.**F32**
Subclass of float that's restricted to 32-bit values

classmethod cast(*other*)

Parameters **other** – Value to convert to F32

Returns If other is symbolic, other. Otherwise, F32(other)

class manticore.wasm.types.**F32ConstImm**(*value: manticore.wasm.types.F32*)

class manticore.wasm.types.**F64**
Subclass of float that's restricted to 64-bit values

classmethod cast(*other*)

Parameters **other** – Value to convert to F64

Returns If other is symbolic, other. Otherwise, F64(other)

class manticore.wasm.types.**F64ConstImm**(*value: manticore.wasm.types.F64*)

class manticore.wasm.types.**FuncIdx**

class manticore.wasm.types.**FunctionType**(*param_types: List[type], result_types: List[type]*)
https://www.w3.org/TR/wasm-core-1/#syntax-functype

param_types = None
Sequential types of each of the parameters

result_types = None
Sequential types of each of the return values

class manticore.wasm.types.**GlobalIdx**

class manticore.wasm.types.**GlobalType**(*mut: bool*, *value: type*)
https://www.w3.org/TR/wasm-core-1/#syntax-globaltype

mut = None
Whether or not this global is mutable

value = None
The value of the global

class manticore.wasm.types.**GlobalVarXsImm**(*global_index: manticore.wasm.types.U32*)

class manticore.wasm.types.**I32**
Subclass of int that's restricted to 32-bit values

classmethod cast(*other*)

Parameters **other** – Value to convert to I32

Returns If other is symbolic, other. Otherwise, I32(other)

static to_unsigned(*val*)
Reinterprets the argument from a signed integer to an unsigned 32-bit integer

Parameters **val** – Signed integer to reinterpret

Returns The unsigned equivalent

class manticore.wasm.types.**I32ConstImm**(*value: manticore.wasm.types.I32*)

class manticore.wasm.types.**I64**
Subclass of int that's restricted to 64-bit values

classmethod cast(*other*)

Parameters **other** – Value to convert to I64

Returns If other is symbolic, other. Otherwise, I64(other)

static to_unsigned(*val*)
Reinterprets the argument from a signed integer to an unsigned 64-bit integer

Parameters **val** – Signed integer to reinterpret

Returns The unsigned equivalent

class manticore.wasm.types.**I64ConstImm**(*value: manticore.wasm.types.I64*)

manticore.wasm.types.**ImmType = typing.Union[manticore.wasm.types.BlockImm, manticore.wasm.t**
Types of all immediates

class manticore.wasm.types.**Instruction**(*inst: wasm.decode.Instruction*, *imm=None*)
Internal instruction class that's pickle-friendly and works with the type system

imm
A class with the immediate data for this instruction

mnemonic
Used for debugging

opcode
Opcode, used for dispatching instructions

exception manticore.wasm.types.**InvalidConversionTrap**(*ty*, *val*)

class manticore.wasm.types.**LabelIdx**

class manticore.wasm.types.**LimitType**(*min: manticore.wasm.types.U32, max: Optional[manticore.wasm.types.U32]*)
https://www.w3.org/TR/wasm-core-1/#syntax-limits

class manticore.wasm.types.**LocalIdx**

class manticore.wasm.types.**LocalVarXsImm**(*local_index: manticore.wasm.types.U32*)

class manticore.wasm.types.**MemIdx**

class manticore.wasm.types.**MemoryImm**(*flags: manticore.wasm.types.U32, offset: manticore.wasm.types.U32*)

manticore.wasm.types.**MemoryType**
https://www.w3.org/TR/wasm-core-1/#syntax-memtype

alias of *manticore.wasm.types.LimitType*

exception manticore.wasm.types.**MissingExportException**(*name*)

class manticore.wasm.types.**Name**

exception manticore.wasm.types.**NonExistentFunctionCallTrap**

exception manticore.wasm.types.**OutOfBoundsMemoryTrap**(*addr*)

exception manticore.wasm.types.**OverflowDivisionTrap**

class manticore.wasm.types.**TableIdx**

class manticore.wasm.types.**TableType**(*limits: manticore.wasm.types.LimitType*, *elemtype: type*)
https://www.w3.org/TR/wasm-core-1/#syntax-tabletype

elemtype = None
the type ot the element. Currently, the only element type is *funcref*

limits = None
Minimum and maximum size of the table

exception manticore.wasm.types.**Trap**
Subclass of Exception, used for WASM errors

class manticore.wasm.types.**TypeIdx**

exception manticore.wasm.types.**TypeMismatchTrap**(*ty1*, *ty2*)

class manticore.wasm.types.**U32**

class manticore.wasm.types.**U64**

exception manticore.wasm.types.**UnreachableInstructionTrap**

manticore.wasm.types.**ValType**
alias of builtins.type

manticore.wasm.types.**Value = typing.Union[manticore.wasm.types.I32, manticore.wasm.types.I**
https://www.w3.org/TR/wasm-core-1/#syntax-val

exception manticore.wasm.types.**ZeroDivisionTrap**

manticore.wasm.types.**convert_instructions**(*inst_seq*) → List[manticore.wasm.types.Instruction]
Converts instructions output from the parser into full-fledged Python objects that will work with Manticore. This is necessary because the pywasm module uses lots of reflection to generate structures on the fly, which doesn't play nicely with Pickle or the type system. That's why we need the *debug* method above to print out immediates, and also why we've created a separate class for every different type of immediate.

Parameters **inst_seq** – Sequence of raw instructions to process

Returns The properly-typed instruction sequence in a format Manticore can use

manticore.wasm.types.**debug**(*imm*)
Attempts to pull meaningful data out of an immediate, which has a dynamic GeneratedStructure type

Parameters **imm** – the instruction immediate

Returns a printable representation of the immediate, or the immediate itself

CHAPTER 11

Plugins

11.1 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*)

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*, *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*)

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

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 12

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 13

Utilities

13.1 Logging

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

CHAPTER 14

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

Z