Manticore Documentation

Release 0.1.0

Trail of Bits

Jan 14, 2019

Contents:

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

CHAPTER 1

API Reference

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

1.1 Core Helpers

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

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

Returns whether *value* is symbolic

Return type bool

manticore.**istainted**(*arg*, *taint=None*)
Helper to determine whether an object if tainted. :param arg: a value or Expression :param taint: a regular expression matching a taint value (eg. 'IMPORTANT.*'). If None, this function checks for any taint value.

1.2 Native Helpers

1.3 ManticoreBase

class manticore.core.manticore.**ManticoreBase**(*initial_state*, *workspace_url=None*, *policy='random'*, ***kwargs*)
Base class for the central analysis object.

Parameters

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

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

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 = context['visited']
    visited.append(state.cpu.PC)
    context['visited'] = visited
```

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

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

Parameters

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

run(*procs=1*, *timeout=None*, *should_profile=False*)
Runs analysis.

Parameters

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

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

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

1.4 native.Manticore

1.5 BaseState

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()`.

constrain(*constraint*)
Constrain state.

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

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

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

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

Returns Concrete value

Return type list[int]

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

Parameters

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

Returns Concrete value

Return type int

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

Parameters

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

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

1.6 native.State

1.7 SLinux

Symbolic Linux

1.8 Cpu

1.9 Models

1.10 EVM

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

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

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)

class manticore.ethereum.**ManticoreEVM**(*procs=10*, *workspace_url: str = None*, *policy: str = 'random'*)

Manticore EVM manager

Usage Ex:

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

all_states

Iterates over the all states (running and terminated)

See also *running_states*.

static compile(*source_code*, *contract_name=None*, *libraries=None*, *runtime=False*, *solc_bin=None*, *solc_remaps=[]*)

Get initialization bytecode from a Solidity source code

count_running_states()

Running states count

count_states()

Total states count

count_terminated_states()

Terminated states count

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

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

Parameters

- **balance** (*int or SValue*) – 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) (optional)
- **name** – a global account name eg. for use as reference in the reports (optional)

Returns an EVMAccount

create_contract(*owner*, *balance=0*, *address=None*, *init=None*, *name=None*, *gas=3000000*)

Creates a contract

Parameters

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

Return type EVMAccount

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

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

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

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

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

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

Parameters

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

Returns If a testcase was generated

Return type bool

get_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_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.

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

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

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

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

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

Parameters

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

Returns Symbolic address in form of a BitVecVariable.

make_symbolic_arguments(*types*)
Make a reasonable serialization of the symbolic argument types

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 )
```

new_address()
Create a fresh 160bit address

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.SoliditySolidityMetadata] = None*) → manticore.core.smtlib.expression.BoolOperation
Returns a constraint that excludes combinations of value and data that would cause an exception in the EVM contract dispatcher. :param address: address of the contract to call :param value: balance to be transferred (optional) :param data: symbolic transaction data :param contract_metadata: SolidityMetadata for the contract (optional)

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

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

running_states
Iterates over running states giving the possibility to change state data.

The state data change must be done in a loop, e.g. *for state in running_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.running_states)*.

save(*state*, *state_id=None*, *final=False*)
Save a state in secondary storage and add it to running or final lists

Parameters

- **state** – A manticore State
- **state_id** – if not None force state_id (overwrite)
- **final** – True if state is final

Returns a state id

solidity_create_contract(*source_code*, *owner*, *name=None*, *contract_name=None*, *libraries=None*, *balance=0*, *address=None*, *args=()*, *solc_bin=None*, *solc_remaps=[]*, *working_dir=None*, *gas=3000000*)
Creates a solidity contract and library dependencies

Parameters

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

- **solc_remaps** (*list of str*) – solc import remaps
- **working_dir** (*str*) – working directory for solc compilation (defaults to current)
- **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

terminated_states
Iterates over the terminated states.

See also *running_states*.

transaction(*caller*, *address*, *value*, *data*, *gas=21000*)
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 SValue*) – balance to be transfered on creation
- **data** – initial data
- **gas** – gas budget

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

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

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

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

CHAPTER 2

Symbolic Input

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

2.1 Wildcard byte

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

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

2.2 Symbolic arguments/environment

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

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

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

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

2.3 Symbolic stdin

Manticore by default is configured with 256 bytes of symbolic stdin data 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()`.

2.4 Symbolic file input

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

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

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

For API use, use the `add_symbolic_file()` interface to customize the initial execution state from an `init()` hook.:

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

2.5 Symbolic sockets

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

CHAPTER 3

Function Models

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

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

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

```
from manticore.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.

CHAPTER 4

Gotchas

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

4.1 Mutable context entries

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

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

4.2 Context locking

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

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

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

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

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

4.3 “Random” Policy

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

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index