Manticore Documentation

Release 0.1.0

Trail of Bits

Sep 06, 2017

Contents:

Manticore is a prototyping tool for dynamic binary analysis, with support for symbolic execution, taint analysis, and binary instrumentation.

CHAPTER 1

API Reference

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

Helpers

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

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

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

Returns whether *value* is symbolic

Return type bool

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

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

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

Manticore

class `manticore.`**`Manticore`**(*binary_path*, *args=None*, *disasm='capstone'*)

The central analysis object.

Parameters

- **`binary_path`** (*`str`*) – Path to binary to analyze
- **`args`** (*`list[str]`*) – Arguments to provide to binary

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

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

Parameters

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

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

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

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

Example use:

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

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

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

Parameters

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

run (*procs=1*, *timeout=0*)
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()`.

verbosity
Convenience interface for setting logging verbosity to one of several predefined logging presets. Valid values: 0-5.

State

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

Parameters

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

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

abandon()

Abandon the currently-active state.

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

constrain(*constraint*)

Constrain state.

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

generate_testcase(*name*, *message='State generated testcase'*)

Generate a testcase for this state and place in the analysis workspace.

Parameters

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

invoke_model(*model*)

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

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

new_symbolic_buffer(*nbytes*, ***options*)

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

Parameters

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

Returns `Expression` representing the buffer.

new_symbolic_value(*nbits*, *label='val'*, *taint=frozenset([])*)

Create and return a symbolic value that is *nbits* bits wide. Assign the value to a register or write it into the address space to introduce it into the program state.

Parameters

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

Returns `Expression` representing the value

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

Parameters

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

Returns Concrete contents of buffer

Return type list[int]

solve_n(*expr*, *nsolves*, *policy='minmax'*)
Concretize a symbolic `Expression` into *nsolves* solutions.

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

Returns Concrete value

Return type list[int]

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

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

Returns Concrete value

Return type int

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

Parameters

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

Cpu

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

The following attributes need to be defined in any derived class

•arch

•mode

- max_instr_width
- address_bit_size
- pc_alias
- stack_alias

all_registers

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

Returns valid register names

Return type tuple[str]

read_bytes(*where*, *size*)

Read from memory.

Parameters

- **where** (*int*) – address to read data from
- **size** (*int*) – number of bytes

Returns data

Return type list[int or Expression]

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

Reads int from memory

Parameters

- **where** (*int*) – address to read from
- **size** – number of bits to read

Returns the value read

Return type int or BitVec

read_register(*register*)

Dynamic interface for reading cpu registers

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

Returns register value

Return type int or long or Expression

write_bytes(*where*, *data*)

Write a concrete or symbolic (or mixed) buffer to memory

Parameters

- **where** (*int*) – address to write to
- **data** (*str or list*) – data to write

write_int(*where*, *expression*, *size=None*)

Writes int to memory

Parameters

- **where** (*int*) – address to write to
- **expr** (*int or BitVec*) – value to write

- **size** – bit size of *expr*

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

Parameters

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

Models

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

manticore.models.**strlen**()

manticore.models.**strcmp**()

CHAPTER 2

Function Models

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

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

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

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

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

CHAPTER 3

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.

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"

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']
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index