---

title: How to use Manticore to find bugs in smart contracts description: How to use Manticore to automatically find bugs in smart contracts author: Trailofbits lang: en tags: ["solidity", "smart contracts", "security", "testing", "formal verification"] skill: advanced published: 2020-01-13 source: Building secure contracts sourceUrl: https://github.com/crytic/building-secure-contracts/tree/master/program-analysis/manticore

---

The aim of this tutorial is to show how to use Manticore to automatically find bugs in smart contracts.

Installation {#installation}

Manticore requires >= python 3.6. It can be installed through pip or using docker.

Manticore through docker {#manticore-through-docker}

```
bash docker pull trailofbits/eth-security-toolbox docker run -it -v "$PWD":/home/training trailofbits/eth-security-toolbox
```

The last command runs eth-security-toolbox in a docker that has access to your current directory. You can change the files from your host, and run the tools on the files from the docker

Inside docker, run:

```
bash solc-select 0.5.11 cd /home/trufflecon/
```

Manticore through pip {#manticore-through-pip}

```
bash pip3 install --user manticore
```

solc 0.5.11 is recommended.

Running a script {#running-a-script}

To run a python script with python 3:

```
bash python3 script.py
```

Introduction to dynamic symbolic execution {#introduction-to-dynamic-symbolic-execution}

Dynamic Symbolic Execution in a Nutshell {#dynamic-symbolic-execution-in-a-nutshell}

Dynamic symbolic execution (DSE) is a program analysis technique that explores a state space with a high degree of semantic awareness. This technique is based on the discovery of "program paths", represented as mathematical formulas called `path predicates`. Conceptually, this technique operates on path predicates in two steps:

1. They are constructed using constraints on the program's input.
2. They are used to generate program inputs that will cause the associated paths to execute.

This approach produces no false positives in the sense that all identified program states can be triggered during concrete execution. For example, if the analysis finds an integer overflow, it is guaranteed to be reproducible.

Path Predicate Example {#path-predicate-example}

To get an insight of how DSE works, consider the following example:

```solidity function f(uint a){

if (a == 65) { // A bug is present }

} ```
```

As `f()` contains two paths, a DSE will construct two different path predicates:

- Path 1: `a == 65`
- Path 2: `Not (a == 65)`

Each path predicate is a mathematical formula that can be given to a so-called SMT solver, which will try to solve the equation. For `Path 1`, the solver will say that the path can be explored with `a = 65`. For `Path 2`, the solver can give `a` any value other than 65, for example `a = 0`.

Verifying properties {#verifying-properties}

Manticore allows a full control over all the execution of each path. As a result, it allows you to add arbitrary constraints to almost anything. This control allows for the creation of properties on the contract.

Consider the following example:

```
solidity function unsafe_add(uint a, uint b) returns(uint c){ c = a + b; // no overflow protection return c; }
```

Here there is only one path to explore in the function:

- Path 1: `c = a + b`

Using Manticore, you can check for overflow, and add constraints to the path predicate:

- `c = a + b AND (c < a OR c < b)`

If it is possible to find a valuation of `a` and `b` for which the path predicate above is feasible, it means that you have found an overflow. For example the solver can generate the input `a = 10 , b = MAXUINT256.`

If you consider a fixed version:

```
solidity function safe_add(uint a, uint b) returns(uint c){ c = a + b; require(c>=a); require(c>=b); return c; }
```

The associated formula with overflow check would be:

- `c = a + b AND (c >= a) AND (c=>b) AND (c < a OR c < b)`

This formula cannot be solved; in other world this is a **proof** that in `safe_add`, `c` will always increase.

DSE is thereby a powerful tool, that can verify arbitrary constraints on your code.

Running under Manticore {#running-under-manticore}

We will see how to explore a smart contract with the Manticore API. The target is the following smart contract `example.sol`:

```solidity pragma solidity >=0.4.24 <0.6.0;

contract Simple { function f(uint a) payable public{ if (a == 65) { revert(); } } } ```

### Run a standalone exploration {#run-a-standalone-exploration}

You can run Manticore directly on the smart contract by the following command (`project` can be a Solidity File, or a project directory):

```
bash $ manticore project
```

You will get the output of testcases like this one (the order may change):

```
... ... m.c.manticore:INFO: Generated testcase No. 0 - STOP ... m.c.manticore:INFO: Generated testcase No. 1 -
REVERT ... m.c.manticore:INFO: Generated testcase No. 2 - RETURN ... m.c.manticore:INFO: Generated testcase No.
3 - REVERT ... m.c.manticore:INFO: Generated testcase No. 4 - STOP ... m.c.manticore:INFO: Generated testcase
No. 5 - REVERT ... m.c.manticore:INFO: Generated testcase No. 6 - REVERT ... m.c.manticore:INFO: Results in
/home/ethsec/workshops/Automated Smart Contracts Audit - TruffleCon 2018/manticore/examples/mcore_t6vi6ij3 ...
```
```

Without additional information, Manticore will explore the contract with new symbolic transactions until it does not explore new paths on the contract. Manticore does not run new transactions after a failing one (e.g: after a revert).

Manticore will output the information in a`mcore_*` directory. Among other, you will find in this directory:

- `global.summary`: coverage and compiler warnings
- `test_XXXXX.summary`: coverage, last instruction, account balances per test case
- `test_XXXXX.tx`: detailed list of transactions per test case

Here Manticore founds 7 test cases, which correspond to (the filename order may change):

| | Transaction 0 | Transaction 1 | Transaction 2 | Result | | :----------------: | :---------------: | :---------------: | ---------------- | :----: | | **test_00000000.tx** | Contract creation | f(!=65) | f(!=65) | STOP | | **test_00000001.tx** | Contract creation | fallback function | | REVERT | | **test_00000002.tx** | Contract creation | | | RETURN | | **test_00000003.tx** | Contract creation | f(65) | | REVERT | | **test_00000004.tx** | Contract creation | f(!=65) | | STOP | | **test_00000005.tx** | Contract creation | f(!=65) | f(65) | REVERT | | **test_00000006.tx** | Contract creation | f(!=65) | fallback function | REVERT |

Exploration summary f(!=65) denotes f called with any value different than 65.

As you can notice, Manticore generates an unique test case for every successful or reverted transaction.

Use the `--quick-mode` flag if you want fast code exploration (it disable bug detectors, gas computation, ...)

Manipulate a smart contract through the API {#manipulate-a-smart-contract-through-the-api}

This section describes details how to manipulate a smart contract through the Manticore Python API. You can create new file with python extension `*.py` and write the necessary code by adding the API commands (basics of which will be described below) into this file and then run it with the command `$ python3 *.py`. Also you can execute the commands below directly into the python console, to run the console use the command `$ python3`.

Creating Accounts {#creating-accounts}

The first thing you should do is to initiate a new blockchain with the following commands:

```python from manticore.ethereum import ManticoreEVM

m = ManticoreEVM() ```

A non-contract account is created using m.create_account:

```
python user_account = m.create_account(balance=1000)
```

A Solidity contract can be deployed using m.solidity_create_contract:

```solidity source_code = ''' pragma solidity >=0.4.24 <0.6.0; contract Simple { function f(uint a) payable public{ if (a == 65) { revert(); } } } '''

# Initiate the contract

contract_account = m.solidity_create_contract(source_code, owner=user_account) ```

#### Summary {#summary}

- You can create user and contract accounts withm.create_account and m.solidity_create_contract.

### Executing transactions {#executing-transactions}

Manticore supports two types of transaction:

- Raw transaction: all the functions are explored

- Named transaction: only one function is explored

#### Raw transaction {#raw-transaction}

A raw transaction is executed using m.transaction:

```
python m.transaction(caller=user_account, address=contract_account, data=data, value=value)
```

The caller, the address, the data, or the value of the transaction can be either concrete or symbolic:

- m.make_symbolic_value creates a symbolic value.
- m.make_symbolic_buffer(size) creates a symbolic byte array.

For example:

```
python symbolic_value = m.make_symbolic_value() symbolic_data = m.make_symbolic_buffer(320)
m.transaction(caller=user_account, address=contract_address, data=symbolic_data, value=symbolic_value)
```

If the data is symbolic, Manticore will explore all the functions of the contract during the transaction execution. It will be helpful to see the Fallback Function explanation in the Hands on the Ethernaut CTF article for understanding how the function selection works.

Named transaction {#named-transaction}

Functions can be executed through their name. To execute `f(uint var)` with a symbolic value, from user_account, and with 0 ether, use:

```
python symbolic_var = m.make_symbolic_value() contract_account.f(symbolic_var, caller=user_account, value=0)
```

If `value` of the transaction is not specified, it is 0 by default.

Summary {#summary-1}

- Arguments of a transaction can be concrete or symbolic
- A raw transaction will explore all the functions
- Function can be called by their name

Workspace {#workspace}

`m.workspace` is the directory used as output directory for all the files generated:

```
python print("Results are in {}".format(m.workspace))
```

Terminate the Exploration {#terminate-the-exploration}

To stop the exploration use m.finalize(). No further transactions should be sent once this method is called and Manticore generates test cases for each of the path explored.

Summary: Running under Manticore {#summary-running-under-manticore}

Putting all the previous steps together, we obtain:

```python from manticore.ethereum import ManticoreEVM

m = ManticoreEVM()

with open('example.sol') as f: source_code = f.read()

user_account = m.create_account(balance=1000) contract_account = m.solidity_create_contract(source_code, owner=user_account)

symbolic_var = m.make_symbolic_value() contract_account.f(symbolic_var)
```

print("Results are in {}".format(m.workspace)) m.finalize() # stop the exploration ```

All the code above you can find in the `example_run.py`

Getting throwing paths {#getting-throwing-paths}

We will now generate specific inputs for the paths raising an exception in `f()`. The target is still the following smart contract `example.sol`:

```
solidity pragma solidity >=0.4.24 <0.6.0; contract Simple { function f(uint a) payable public{ if (a == 65) { revert(); } } }
```

Using state information {#using-state-information}

Each path executed has its state of the blockchain. A state is either ready or it is killed, meaning that it reaches a THROW or REVERT instruction:

- m.ready_states: the list of states that are ready (they did not execute a REVERT/INVALID)
- m.killed_states: the list of states that are killed
- m.all_states: all the states

```
python for state in m.all_states: # do something with state
```

You can access state information. For example:

- `state.platform.get_balance(account.address)`: the balance of the account
- `state.platform.transactions`: the list of transactions
- `state.platform.transactions[-1].return_data`: the data returned by the last transaction

The data returned by the last transaction is an array, which can be converted to a value with ABI.deserialize, for example:

```
python data = state.platform.transactions[0].return_data data = ABI.deserialize("uint", data)
```

How to generate testcase {#how-to-generate-testcase}

Use m.generate_testcase(state, name) to generate testcase:

```
python m.generate_testcase(state, 'BugFound')
```

Summary {#summary-2}

- You can iterate over the state with m.all_states
- `state.platform.get_balance(account.address)` returns the account's balance
- `state.platform.transactions` returns the list of transactions
- `transaction.return_data` is the data returned
- `m.generate_testcase(state, name)` generate inputs for the state

Summary: Getting Throwing Path {#summary-getting-throwing-path}

```python from manticore.ethereum import ManticoreEVM

m = ManticoreEVM()

with open('example.sol') as f: source_code = f.read()

user_account = m.create_account(balance=1000) contract_account = m.solidity_create_contract(source_code, owner=user_account)

symbolic_var = m.make_symbolic_value() contract_account.f(symbolic_var)

# Check if an execution ends with a REVERT or INVALID
```

for state in m.terminated_states: last_tx = state.platform.transactions[-1] if last_tx.result in ['REVERT', 'INVALID']: print('Throw found {}'.format(m.workspace)) m.generate_testcase(state, 'ThrowFound') ```

All the code above you can find into the [example_run.py]

Note we could have generated a much simpler script, as all the states returned by terminated_state have REVERT or INVALID in their result: this example was only meant to demonstrate how to manipulate the API.

Adding constraints {#adding-constraints}

We will see how to constrain the exploration. We will make the assumption that the documentation of `f()` states that the function is never called with `a == 65`, so any bug with `a == 65` is not a real bug. The target is still the following smart contract [example.sol]:

```
solidity pragma solidity >=0.4.24 <0.6.0; contract Simple { function f(uint a) payable public{ if (a == 65) { revert(); } } }
```

Operators {#operators}

The Operators module facilitates the manipulation of constraints, among other it provides:

- Operators.AND,
- Operators.OR,
- Operators.UGT (unsigned greater than),
- Operators.UGE (unsigned greater than or equal to),
- Operators.ULT (unsigned lower than),
- Operators.ULE (unsigned lower than or equal to).

To import the module use the following:

```
python from manticore.core.smtlib import Operators
```

`Operators.CONCAT` is used to concatenate an array to a value. For example, the return_data of a transaction needs to be changed to a value to be checked against another value:

```
python last_return = Operators.CONCAT(256, *last_return)
```

Constraints {#state-constraint}

You can use constraints globally or for a specific state.

Global constraint {#state-constraint}

Use `m.constrain(constraint)` to add a global constraint. For example, you can call a contract from a symbolic address, and restraint this address to be specific values:

```
python symbolic_address = m.make_symbolic_value() m.constraint(Operators.OR(symbolic == 0x41, symbolic_address == 0x42)) m.transaction(caller=user_account, address=contract_account, data=m.make_symbolic_buffer(320), value=0)
```

State constraint {#state-constraint}

Use state.constrain(constraint) to add a constraint to a specific state It can be used to constrain the state after its exploration to check some property on it.

Checking Constraint {#checking-constraint}

Use `solver.check(state.constraints)` to know if a constraint is still feasible. For example, the following will constraint symbolic_value to be different from 65 and check if the state is still feasible:

```
python state.constrain(symbolic_var != 65) if solver.check(state.constraints): # state is feasible
```

Summary: Adding Constraints {#summary-adding-constraints}

Adding constraint to the previous code, we obtain:

```python from manticore.ethereum import ManticoreEVM from manticore.core.smtlib.solver import Z3Solver

solver = Z3Solver.instance()

m = ManticoreEVM()

with open("example.sol") as f: source_code = f.read()

user_account = m.create_account(balance=1000) contract_account = m.solidity_create_contract(source_code, owner=user_account)

symbolic_var = m.make_symbolic_value() contract_account.f(symbolic_var)

no_bug_found = True

## Check if an execution ends with a REVERT or INVALID

for state in m.terminated_states: last_tx = state.platform.transactions[-1] if last_tx.result in ['REVERT', 'INVALID']: # we do not consider the path were a == 65 condition = symbolic_var != 65 if m.generate_testcase(state, name="BugFound", only_if=condition): print(f'Bug found, results are in {m.workspace}') no_bug_found = False

if no_bug_found: print(f'No bug found') ```

All the code above you can find into the `example_run.py`
```