HEVM, a Smart Contract Verification Tool

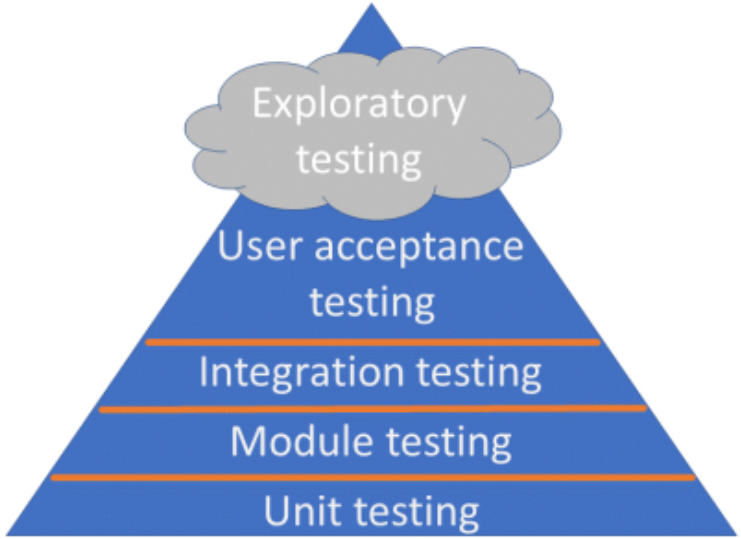
Mate Soos, Ethereum Foundation

25th of October 2023

Outline

A Quick Recap of Testing

Testing is a discipline. Reliable systems usually have a **testing strategy** encompassing at least parts of the test pyramid.



As part of the tests, there are usually (1) a set of tests against **known bad states** that should never be entered, and (2) a set of tests checking the system is currently in a known good state, also called **invariant tests**.

HEVM can help with finding ways to trigger *all* known bad states and validate that invariants *always* hold.

What is HEVM

Can run EVM bytecode **concretely** like geth, or can run EVM bytecode **symbolically**:

- Takes as input known bad state/invariant test
- Examines all execution paths
- Finds the set of requirements to reach all failing paths
- Runs external tool (SMT solver) to find input to reach them
- Displays call & state needed to trigger the bad state/invalidate the invariant

Symbolic Execution: Basics

Say your code looks like this:

```
function overflow(uint a) public pure {
        uint b;
        unchecked { b = a + 1;}
        assert(b > a);
}
```

HEVM can find the case where $a = 0xffffff \ldots$ to trigger the assert due to roll-around. HEVM gives you the **exact call to reproduce the bug**. This way of using HEVM can find a *known-bad state*.

Symbolic Execution vs Fuzzing

Say your code looks like this:

```
function tricky(uint a, uint b) public pure {
        // solution: a = 10000983843024
        //           b = 9877982748934

        if (a * 2 + b == 29879950434982 &&
            b / 2 == 4938991374467) {
                assert(false); // bad things happen
        }
}
```

Fuzzing never finds this edge-case. Symbolic execution always finds it.

In general, fuzzing is faster, but is incomplete. Symbolic execution is slower but complete.

Symbolic Execution: Invariant Checking

You can describe **invariants** of your contract and write them as functions
Then assert these functions every time the invariant must hold.

```
function my_invariant() private pure returns inv {
        inv = ...calculate invariant...
}

function transfer(uint a) public pure {
        require(my_invariant());
        ... your function code here ....
        assert(my_invariant());
}
```

Here, instead of a known-bad state, we validate that we are always in state that matches our expectations.

Symbolic Execution: Equivalence Checking

You can ask HEVM to check whether two implementations are equivalent

```
function (...) public {
  simple known good
  function
  /uses lots of gas/
}
```

```
function (...) public {
  complicated function

  /uses less gas/
}
```

HEVM can give you the exact **call to trigger the discrepancy** between the two functions. This way, you can safely improve the gas performance of your code.

Limitations of HEVM

While HEVM aims to be fully-featured, it has a number of inherent limitations:

- Loops can be challenging. The option `--max-iterations N` can raise the iteration limit until which loops are examined.
- Recursion, and parametric calls can cause HEVM to only partially explore the state
- Complicated mathematical expressions such as exponentiation can pose a challenge.
- It is important to set all known invariants at the start of the test via `require()` to speed up solving.
- HEVM itself is not verified, and neither are the solvers we use. However, some solvers will in the future be able to emit a proof, which can be verified.

How it works: The Intermediate Representation

Through the interpreter we create the global expression that captures all end-states. We then take each end-state out and filter it for things we are looking for, e.g. assertion failures. This is our intermediate representation, similar to what YUL is for Solidity. Let's take the previous example:

```
function overflow(uint a) public pure {
        uint b;
        unchecked { b = a + 1;}
        assert(b > a);
}
```

The expression to generate a counterexamle for this could look like:

```
PLEq (Add (Var "a") (Lit 1)) (Var "a")
```

Notice: we use less-or-equal, because we want a **counterexample**

How it Works: Passing the Expression to an SMT solver

Let's say the expression is:

```
PLEq (Add (Var "a") (Lit 1)) (Var "a")
```

The SMT expression for this could be:

```
(set-logic QF_AUFBV)
(define-sort Word () (_ BitVec 256))
(declare-const varA (Word))
(assert (bvule (bvadd varA (_ bv1 256)) varA))
(check-sat)
```

Z3 gives the answer:

```
sat
(
  (define-fun varA () (_ BitVec 256)
    #xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff)
)
```

Using HEVM

Install foundry [1]. Get static HEVM binary [2]. Install z3 [3]. Add foundry test cases, and prepend with ‘‘`prove_`’’ the ones you want HEVM to use:

```
function prove_add(uint x, uint y) public pure {
    unchecked { if (x + y < x) return; }
    assert(x + y >= x);
}
```

Then run with:

```
forge build
hevm test
```

[1] https://github.com/foundry-rs/foundry
[2] https://github.com/ethereum/hevm/releases
[3] https://github.com/Z3Prover/z3

Contributing Benchmarks

To test and improve the performance of HEVM, we use a benchmark repository [1]. Here, you can test your example problem against other full-featured symbolic execution engines such as Halmos [2]:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
git clone https://github.com/eth-sc-comp/benchmarks/
nix develop
./bench.py
./gen_graphs.py
cd graphs
```

We are keen on in real-world benchmarks that you feel could be interesting for the community.

[1] https://github.com/eth-sc-comp/benchmarks/
[2] https://github.com/a16z/halmos

Contributing Back to HEVM

The HEVM repository uses`nix` for ease of development:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
git clone https://github.com/ethereum/hevm/
nix-shell
cabal run exe:hevm -- test
```

You now have a full development environment, with all necessary tools installed, including Z3.

HEVM is written in Haskell, but there are many areas that can be contributed to without deep knowledge of Haskell. For example, expression simplification:

```
    go (Add a b)
      | b == (Lit 0) = a
      | a == (Lit 0) = b
      | otherwise = add a b
```

Conclusions

HEVM is a fully-featured, easy-to-use tool that can help find bugs in your code. It can be a useful **part of your testing strategy**.

HEVM can help you gain more confidence in the correctness of your system. It can be especially helpful to validate that certain catastrophic failure modes cannot be triggered. HEVM, however, can be incomplete: it can struggle with loops, complicated/recursive call chains, and complicated mathematical expressions.

Download HEVM from
`https://github.com/ethereum/hevm/releases`