HEVM, a Smart Contract Verification Tool

Mate Soos, Ethereum Foundation

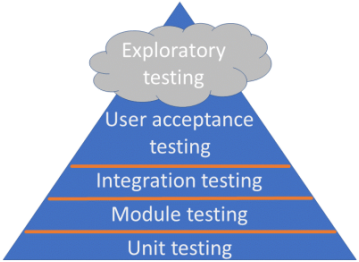
12th of December 2023

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. API specification that describes what should happen, checked by **positive tests**
2. Bad outcomes that should not happen, usually checked by **negative tests**
3. A set of known bad states that should never be entered, checked by **invariant tests**



Symbolic execution can help with finding ways to trigger *all* negative tests and validate that invariants *always* hold.

A Quick Recap of Ethereum

Ethereum is the second largest cryptocurrency in terms of value. Ethereum has:

- Stack-based VM, the EVM. Most popular executor of EVM: geth
- Contracts with code and storage
- ACID execution semantics
- EVM-specific programming language, Solidity and its IR, YUL
- Proof-of-stake as its consensus layer
- Soon a data availability layer ("proto-danksharding") for roll-ups

Downsides:

- Code and data are not clearly separated ("EOF" will solve)
- JUMP destinations are known but dynamic jumps are possible
- All EVM instructions operate on 256b values

Symbolic Execution – straight line program

Most execution works by running instructions concretely:

```
---              ax: 1    , bx: 2
mov %bx %ax      ax: 2    , bx: 2
add %ax $4       ax: 6    , bx: 2
```

Symbolic execution, with symbolic state:

```
---              ax: v1   , bx: v2
mov %bx %ax      ax: v2   , bx: v2
add %ax $4       ax: v2+4, bx: v2
```

Symbolic Execution – branching

Concrete execution:

```
-----------      ax: 1    bx: 1
cmp %ax %bx      ax: 1    bx: 1
je .if_true
; false
add %ax $4
jmp short .end
.if_true:
add %ax $5       ax: 6    bx: 1
.end:
```

Symbolic execution, with symbolic state:

```
-----------      ax: v1    bx: v2
cmp %ax %bx
je .if_true
; false
add %ax $4       ax: v1+4 bx: v2
jmp short .end
.if_true:
add %ax $5       ax: v1+5 bx: v2
.end:
---- v1==v2 ->   ax: v1+5 bx: v2
---- v1!=v2 ->   ax: v1+4 bx: v2
```

For symbolic execution, we end up having to follow two executions. This can become exponential.

HEVM

- Uses the Ethereum Virtual Machine (EVM) for execution
- Can do both concrete and symbolic execution
- Takes as input negative/invariant test or suspected-equivalent code
- Examines *all*[1] execution paths from the starting state
- Finds the set of requirements to reach all failing paths (i.e. `assert`-s)
- Runs internal fuzzer or external SMT solver to find input to reach them
- Displays call needed to trigger the bad state/invalidate the invariant

[1] loops/recursion is an issue, we have a loop/depth limit

Symbolic Execution: Negative Tests

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

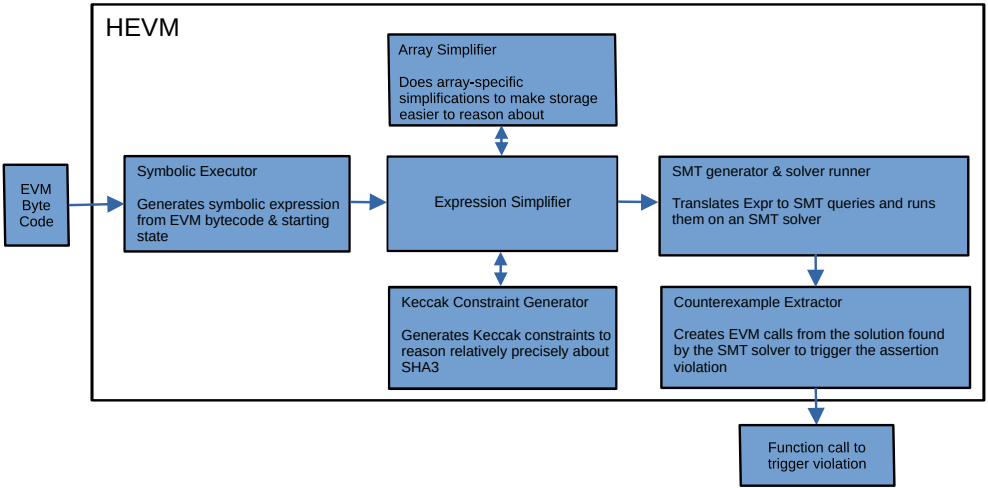
You can ask HEVM to check whether two implementations are equivalent

```
function (...) public returns x {
  x = /known good computation/
      /uses lots of gas/
}
```

```
function (...) public returns x {
  x = /complicated computation/
      /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.

How HEVM works



HEVM's Symbolic Executor

HEVM's symbolic executor is very powerful:

- Operates on *bytecode* so runs everything every deployed to the chain
- Understands all of the EVM: stack, call frames, memory, storage, calldata
- Can be set up to run at any point in time (blockchain history), through RPC
- Can chain any number of symbolic transactions
- Fuzzed against concrete execution (geth)

Limitations:

- Cannot deal with symbolic gas other than ignoring it
- Symbolic offset/size memory copy is not yet implemented
- Because it runs on bytecode, it needs to recover variable bit-widths, loops, etc.

Intermediate Representation

Through the symbolic execution engine we create an 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. Let's take the 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**

Intermediate Representation: Writes

Writing in a buffer is represented as a chain of writes:

```
AbstractBuf "a" // empty buffer
WriteWord (Lit 1) (Var "a") (AbstractBuf "a") //1st write
WriteWord (Lit 2) (Var "b") (WriteWord (Lit 1) (Var "a") (AbstractBuf "a")) //2nd
```

This can later be collapsed, e.g. in case the variables are concertized. We use `(AbstractBuf "var")` for symbolic, and `(ConcreteBuf "value")` for concrete values.

While this can be inefficient, it allows us perform symbolic execution as fast as possible, only prepending an operation for each instruction. We can (and do) later collapse & simplify these writes.

NOTE: Another way of doing this would be e.g. interval graphs

Intermediate Representation: Keccak

Keccak, i.e. SHA-3 is used extensively by all contracts in EVM. This is because e.g. storage of contracts is an *unstructured array* of uint256. Hence, to implement e.g. maps and an arrays, one needs to use Keccak to map to a "random" position in storage, so as not to clash.

It's very expensive to accurately represent Keccak in SMT. Hence, we represent Keccak as an *uninterpreted function* in SMT, with the following rules:

- We know the concrete value of the input $\rightarrow$ we add the axiom $keccak(input) = output$
- Size of value hash differs $\rightarrow$ hash differs. We assert this for all pairs
- Assert all pairs of keccak to be unequal if they don't match over *partial* concrete values
- $keccak(x) > 128$. Small slot values are used for non-map/array elements of contracts

Intermediate Representation: Maps

Storage of contracts is an unstructured array of uint256. Solidity uses: $keccak(bytes32(key)||bytes32(id))$ to map $mymap[key]$ where id is the map index:

```
contract C {
    mapping(uint => uint) mymap_id0;
    mapping(uint => uint) mymap_id1;
}
```

We have Solidity-specific rewrite rules to strip writes. So when writing two $keccak(bytes32(key)||bytes32(id))$-s on top of each other, and then reading, we traverse the list of writes to pick out the potentially matching one(s).

Notice that the SMT solver can use our Keccak rules to do this, too, but it's a lot slower

Intermediate Representation: Simplification

We do all the following rewrites to fixedpoint:

- Canonicalization of all commutative operators (concrete value first)
- Canonicalization of all less-than/greater-than/etc operators
- Canonicalization of all Keccak expressions to match Solidity patterns
- Stripping writes when they are not read
- Stripping reads when they are not used
- Constant folding
- (In)equality propagation
- All meaningful rewrites, such as min/max/add+sub/add+add/sub+sub/etc.
- Delayed simplification of Keccak expressions to keep structure as much as possible

Intermediate Representation: Haskell to the Rescue

Haskell supports algebraic data types (ADTs), so our intermediate representation (IR) can be fully typed. Furthermore, Haskell supports pattern matching, so our rewrite rules are easy to read & write:

```
-- syntactic Eq reduction
go (Eq (Lit a) (Lit b))
  | a == b = Lit 1
  | otherwise = Lit 0
go (Eq (Lit 0) (Sub a b)) = eq a b
go (Eq a b)
  | a == b = Lit 1
  | otherwise = eq a b
```

What can we use the IR for?

- Finding counterexamples with SMT solvers
- Constant extraction: helping other fuzzers with "magic numbers"
- Substitute constants and fold: we can build a fuzzer
- Not just failing branches: automated test-case generation

Solving the IR: Creating a Fuzzer

Fuzzing sounds strange given that HEVM is a symbolic execution engine. However:

- The IR is actually a very clean representation of the problem at hand
- Due to the extensive simplifications applied, it can contain specific constants that may be very hard to find otherwise
- IR *could* be transpiled to assembly (potentially through `C`), and executed as a program

Notice that geth, the normal EVM concrete executor is incredibly slow. Hence a transpiled IR could achieve serious speedup.

Solving the IR: using an SMT Solver

The IR is translated in a straightforward manner to SMT. 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);
}
```

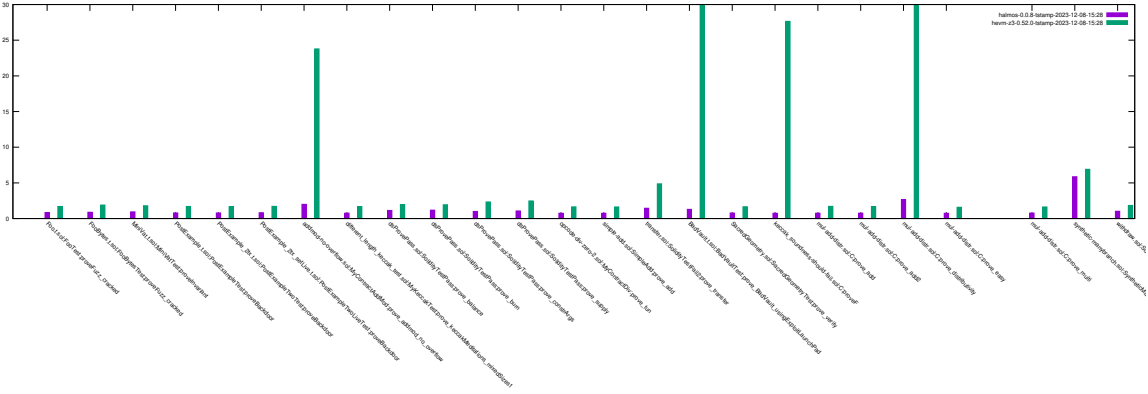
Then run with:

```
forge build
hevm test
```

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

Results

To test and improve the performance of HEVM, we use a benchmark repository [1] developed in conjunction with the Halmos team [2], and soon we will have the K framework contributing as well.



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

Limitations & Future Work

HEVM has a number of inherent limitations:

- Loops are challenging. We have an iteration limit until which loops are examined
- Recursion, and parametric calls can cause HEVM to only partially explore the state
- Complicated mathematical expressions (e.g. division, modulo) can cause a challenge
- HEVM is not verified, and neither are SMT solvers. Future: SMT solvers will emit a proof

Future work:

- CFG extraction to better handle recursion and loops
- Invariant computation for loops
- Time-limited, early-abort SMT solving and fuzzing
- Some pre-written invariants and negative tests, moving towards static analysis

Conclusions

HEVM is a fully-featured, easy-to-use tool that can help find bugs in EVM bytecode. Its sophisticated IR is tailored to the EVM via Keccak- and array/map-specific rewrites to lower the complexity of solving the final SMT expression.

Development environment:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
git clone https://github.com/ethereum/hevm/
nix-shell
cabal repl test
[...]
> :main
```