hevm, a flexible symbolic execution framework to verify EVM bytecode

dxo, *Mate Soos*, Zoe Paraskevopoulou

Argot Collective (`https://argot.org`)

7th of March 2025, BSA Conference, Switzerland

Outline

Symbolic Execution vs Fuzzing

Say your code is:

```
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: 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.

Related Work

Symbolic execution is used in two major ways. One is to **validate static code analysis** results, the other is **pure symbolic execution**. The first approach is followed by Oyente, sCompile, Mythril, etc. These are typically incomplete, and false positives are allowed.

Purely symbolic execution-based systems:

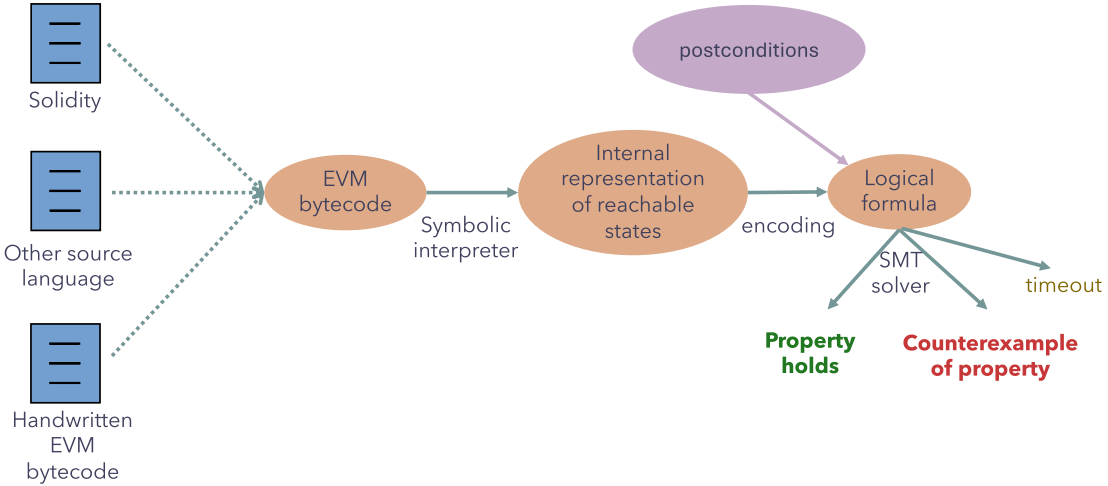
- **halmos**: Written in python, with its own IR and internal rewrite engine
- **Certora Prover**: Based on backwards exploration and weakest precondition computation
- **KEVM**: K-framework based, allows to "break out" into K to prove lemmas
- **EthBMC**: Bounded model checking-based exploration of contracts

Overview of hevm

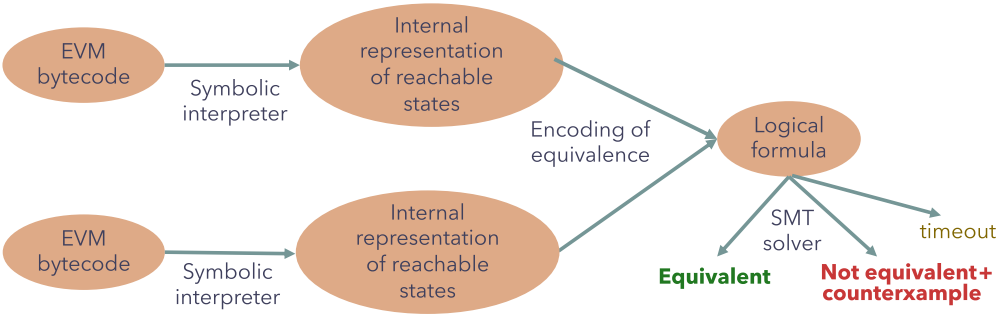
- Started ≈ 7 years ago as part of dapptools, but is now a standalone tool
- Implements EVM semantics for concrete and symbolic execution
- Examines *all*[1] execution paths from the starting state
- Finds the set of requirements to reach all failing paths
- Runs external SMT solver(s) to find input to reach them
- Displays call needed to trigger:
  - the fault: `hevm test`
  - the discrepancy: `hevm equivalence`

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

hevm: Symbolic Execution for Counterexample Generation



hevm: Symbolic Execution for Equivalence Checking



hevm's Symbolic Executor

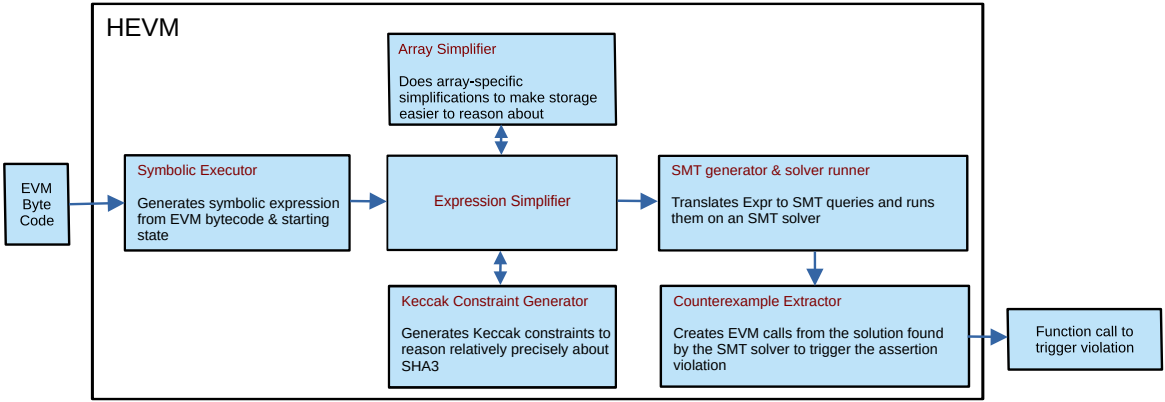
hevm's symbolic executor is very powerful:

- **Operates on bytecode** so runs everything deployed to the chain
- Understands **all of EVM**: stack, call frames, memory, storage, calldata
- Can run on **any point in blockchain history** via RPC to an archive node
- Pull all required contracts from the chain via RPC to a full node
- Overapproximates unknown code
- Fuzzed against concrete execution (geth) for correctness

Limitations:

- Cannot deal with **symbolic gas** other than ignoring it
- **Symbolic offset/size memcopy** is not implemented, but is often unneeded
- **Loops and recursion** are explored only to a fixed depth

hevm internals



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**

Using "hevm test"

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 check:

```
import {Test} from "forge-std/Test.sol";
contract MyContract is Test {
  function prove_add_fail(uint x, uint y) public pure {
      unchecked {
          uint256 z = x + y;
          assert(z >= x);
      } } }
```

```
forge build --ast && hevm test
[RUNNING] prove_add_fail(uint256,uint256)
   [FAIL] prove_add_fail(uint256,uint256)
   Counterexample:
     calldata: prove_add_fail(1,1157920892373161954235 70...)
```

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

Using "hevm equivalence"

```
$ time --verbose hevm equivalence --code-a "0x...." --code-b "0x..."
[WARNING] hevm was only able to partially explore the contract due to the following
issue(s):
  - Unexpected Symbolic Arguments to Opcode
    msg: "call target has unknown code"
    opcode: STATICCALL
    program counter: 14137
    arguments: 0x0000000000000000000000000000000000000000
    [...]
Found 1729225 total pairs of endstates
No discrepancies found
But the following issues occurred:
      93x -> CopySlice with a symbolically sized region not currently implemented
      2x -> SMT result timeout/unknown
Command exited with non-zero status 1
        User time (seconds): 45762.27
        Elapsed (wall clock) time (h:mm:ss or m:ss): 34:32.70
```

References & Pro Tips Using hevm

References:

- hevm repository: `https://github.com/ethereum/hevm/`
- hevm user guide: `https://hevm.dev/`
- Forge testing guide: `https://book.getfoundry.sh/forge/writing-tests`

Pro tips:

- Equivalence testing can be a useful way to check if a refactoring is correct
- The spec of your contract is effectively its set of test cases
- Write **postive, negative, and invariant** tests. Ex:
  - Positive test: required number of signatures are met, transfer allowed
  - Negative test: required number of signatures are not met, transfer not allowed
  - Invariant test: after transfer, sum of balances is the same
- Even if "hevm test" emits warnings, it still adds a level of assurance beyond pure fuzzing

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 work:

- Symbolic CopySlice handling – I finally have an idea :)
- Better handling of loops and recursion: include into IR, and solve for invariant via CHC

`https://github.com/ethereum/hevm/releases`