hevm, a flexible symbolic execution framework to verify EVM bytecode

dxo, *Mate Soos*, Zoe Paraskevopoulou

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

11th of June 2025, Dagstuhl Seminar Testing Program Analyzers and Verifiers
`https://www.dagstuhl.de/25242`

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.

EVM

- Stack machine
- Everything is 256b by default, addresses are 160b
- No undefined behaviour
- No IO. No printing, disk, networking, etc.
- Only has a few regions for data: returndata, calldata, memory, storage, (stack)
- No pointers. One contiguous memory, calldata, and returndata region
- No such thing as infinite loops: gas is a unit of execution that is always limited
- No such thing as huge memory/storage usage: gas also limits memory and storage
- There are calls to other code already deployed, sometimes not yet deployed (i.e. call to unknown code)

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:

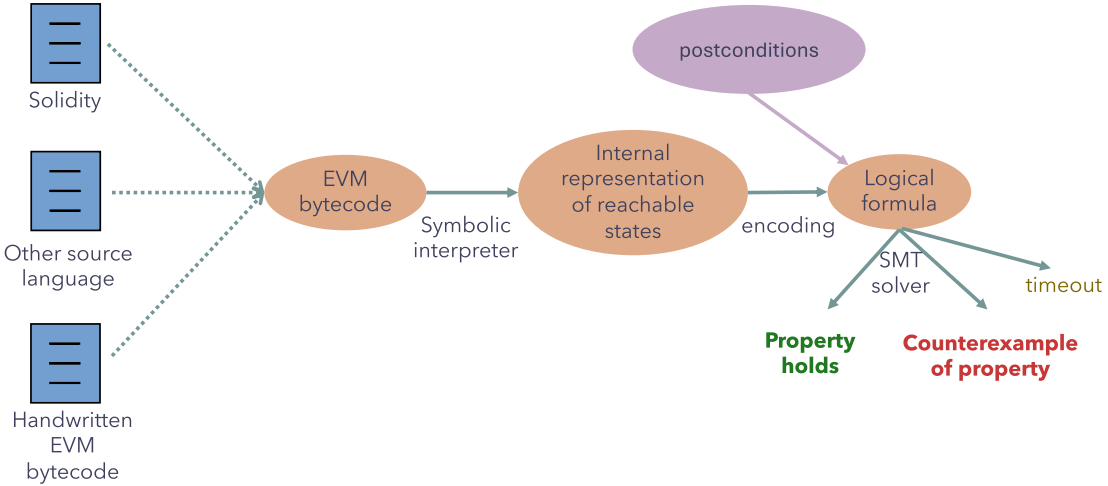
- **halmos**: Written in python, with its own IR and internal rewrite engine
- **Certora Prover**: Based on backwards exploration and weakest precondition computation
- **KEVM**: K-framework based, allows to "break out" into K to prove lemmas
- **EthBMC**: Bounded model checking-based exploration of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

Overview of hevm

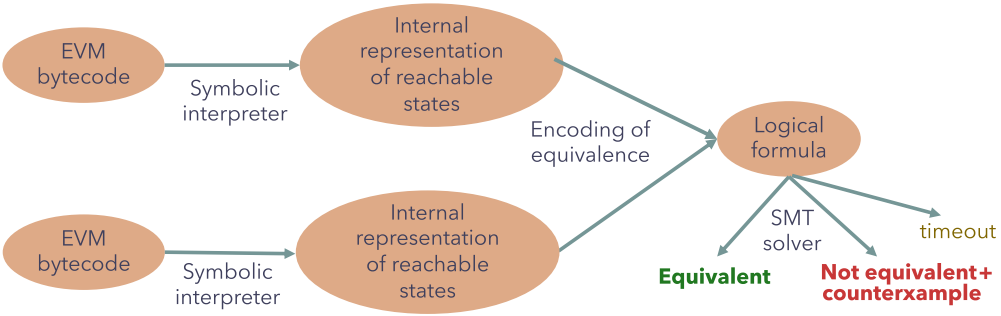
- Started ≈ 7 years ago as part of dapptools, but is now a standalone tool
- Implements EVM semantics for concrete and symbolic execution
- Examines *all*[1] execution paths from the starting state
- Finds the set of requirements to reach all failing paths
- Runs external SMT solver(s) to find input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

[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

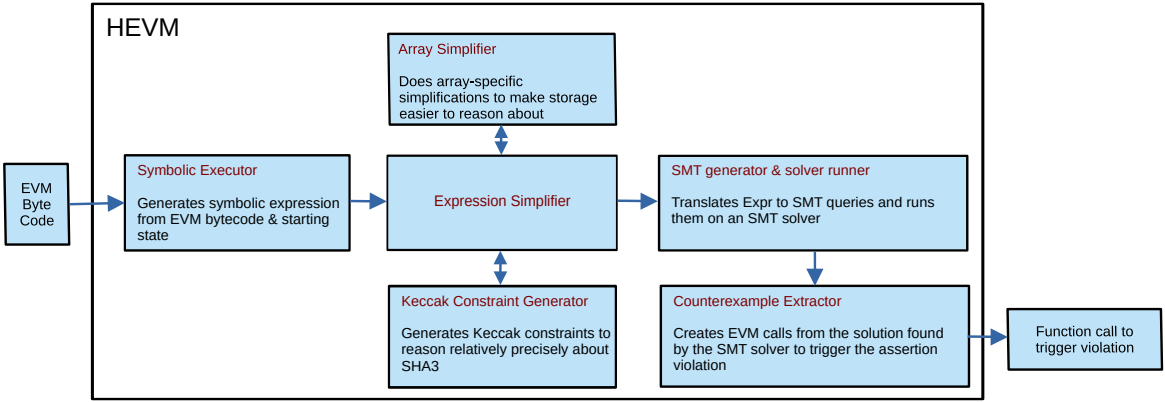
hevm's symbolic executor is very powerful:

- **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

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



What is a bug anyway?

- I can steal the money due to e.g. overflow/reentrancy/logical error/etc bug
- I can lock in the money due to liveness issues
- I can exploit incorrect rounding in online exchanges to get better prices
- I can borrow money to buy some asset, dump it in an automated market maker pushing its price to zero, then buy everything for zero.
- Frontrunning: I can observe a transaction, see that it makes money, and replace the recipient of the transaction with my own address, so I get the money instead.
- Sandwiching: I can see if someone is buying an asset, buy a lot of the asset before their transaction is executed, then sell it all off later for a higher price.

And the list goes on. Some of these are "clearly" bugs. Some of them are abuse of systems (frontrunning). Most interesting oners are games played according to the rules, but not according to the spirit.

What is a bug anyway? (cont.)

- We decided NOT to enforce any bug specification
- Everything has to be specified by the user via an assertion
- So we don't have a specification language
- The spec has to be expressed in-line code via require/assert
- We provide an interface where it's possible to specify pre/post-conditions
- Separate tool is in the works that adds pre/post conditions to prevent the above

This way we allow others to be opinionated, but we ourselves remain unopinionated.

How we ensure correctness of hevm

- Concrete execution checked via eth test suite
- Over 100 unit test for internal components (e.g. IR simplifications)
- Over 100 end-to-end tests for symbolic execution
- IR simplification checked via: generate IR via quickcheck $\rightarrow$ simplify IR $\rightarrow$ translate original IR and simplified IR to SMT, equivalence check $\rightarrow$ must be UNSAT
- Symbolic execution checked via: random conctract generator $\rightarrow$ symbolic execution $\rightarrow$ concrete input $\rightarrow$ constant folding vs geth + concrete input $\rightarrow$ final state
- Curated set of symbolic test cases checked against kontrol (KEVM) and halmos

Bugs, issues, and inconveniences

- The IR $\rightarrow$ SMT translation was incorrect, it sometimes produced trivially UNSAT queries
- Now we check if a != a' — if it's also UNSAT, we know our translation is buggy
- IR was not canonical, so two identical expressions could be different in IR, due to missing commutativity and associativity rewrites
- geth behaviour is undefined for large ($> 2^{64}$) memory usage, so memory overflow simplifications can be unsound
- ... so in fact some behaviour is undefined, but this is only a problem for compliers and symbolic execution systems
- Since storage is limited, we have to limit counterexample sizes – certain optimizations exploit limited memory
- Once our tool started to be used by a larger tool, Echidna, in symbolic execution mode, a lot of edge-cases had to be dealt with via soft errors

Where to find hevm

hvm repository: `https://github.com/ethereum/hevm/`
hevm user guide: `https://hevm.dev/`

Code:

Paper:

Thank you for your time!

Thanks for listening!