hevm, a Fast Symbolic Execution Framework for EVM Bytecode

dxo[1], *Mate Soos*[1], Zoe Paraskevopoulou[1], Martin Lundfall[2], Mikael Brockman[2]

[1] Ethereum Foundation, [2] Independent Researcher

19th of August 2024, Stanford

Outline

A Quick Recap of Ethereum

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

- Contracts with **code and storage**, unlike e.g. Bitcoin
- Stack-based VM, the EVM. Most popular executor of EVM: **geth**
- ACID execution semantics, like a **database**
- Proof-of-stake as its **consensus layer** – ordering transactions

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 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:

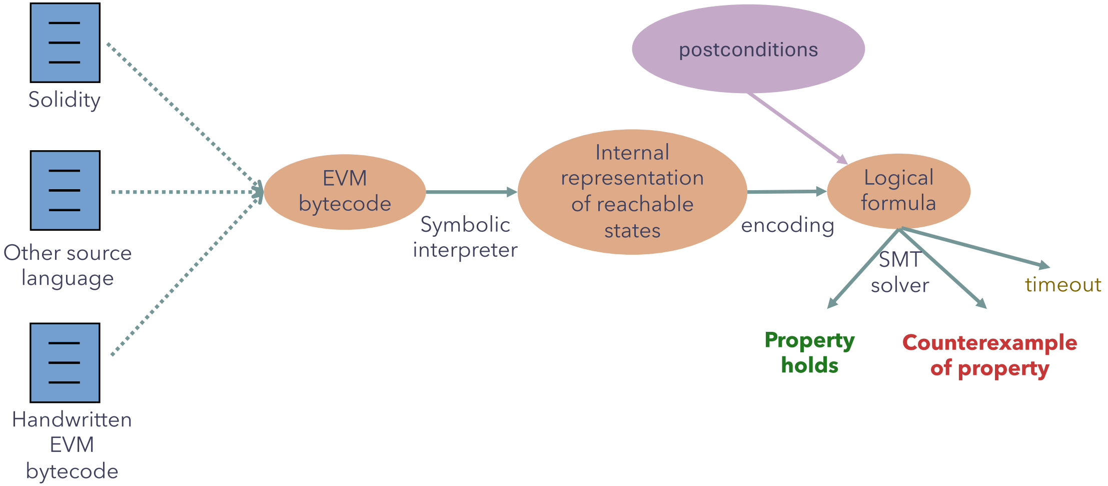
- **Certora Prover**: Based on backwards exploration and weakest precondition computation
- **EthBMC**: Bounded model checking-based exploration of contracts
- **halmos**: Written in python, with its own IR and internal rewrite engine
- **KEVM**: K-framework based, allows to "break out" into K to prove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

hevm

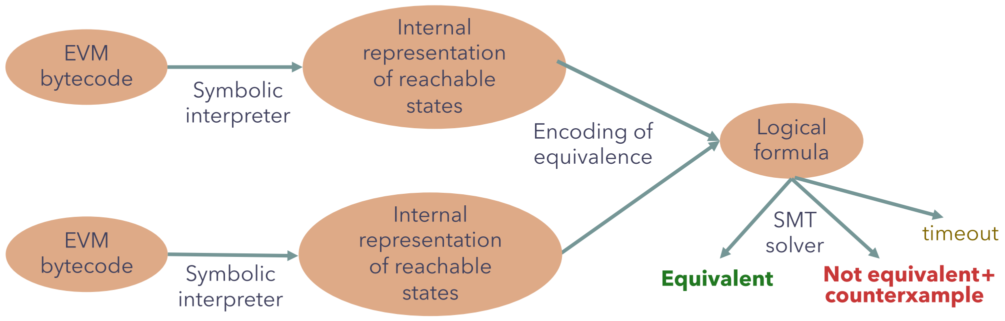
- Implements EVM semantics for concrete and symbolic execution
- Can do both concrete and symbolic execution
- Examines *all*[1] execution paths from the starting state
- Finds the set of requirements to reach all failing paths (i.e. `assert`-s)
- Runs external SMT solver to find input to reach them
- Displays call needed to trigger the bad state/invalidate the invariant

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

hevm: Symbolic Execution for Counterexample Generation



hevm: Symbolic Execution for Equivalence Checking



hevm's Symbolic Executor

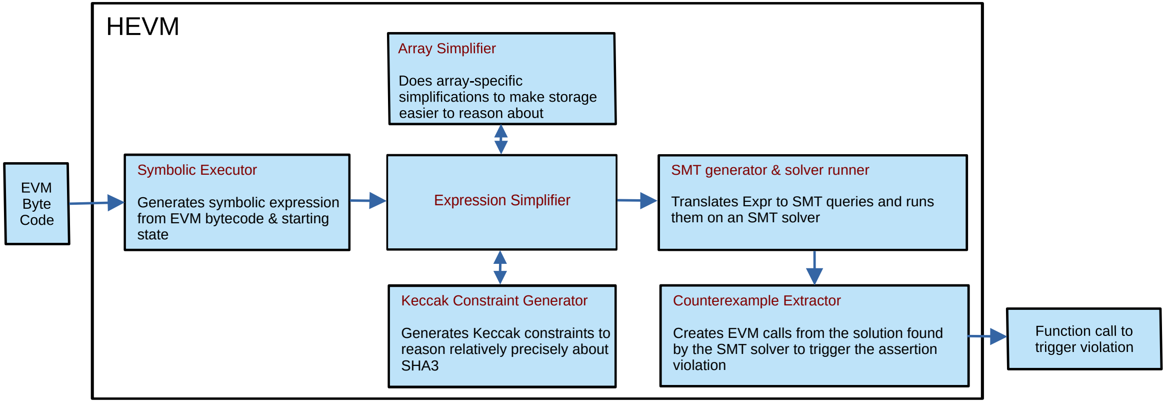
hevm's symbolic executor is very powerful:

- **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**, through RPC
- Can be used to **chain together symbolic transactions**
- Fuzzed against concrete execution (geth) for correctness

Limitations:

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

hevm internals


HEVM
EVM Byte Code
Symbolic Executor
Generates symbolic expression from EVM bytecode & starting state
Array Simplifier
Does array-specific simplifications to make storage easier to reason about
Expression Simplifier
Keccak Constraint Generator
Generates Keccak constraints to reason relatively precisely about SHA3
SMT generator & solver runner
Translates Expr to SMT queries and runs them on an SMT solver
Counterexample Extractor
Creates EVM calls from the solution found by the SMT solver to trigger the assertion violation
Function call to trigger violation

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:

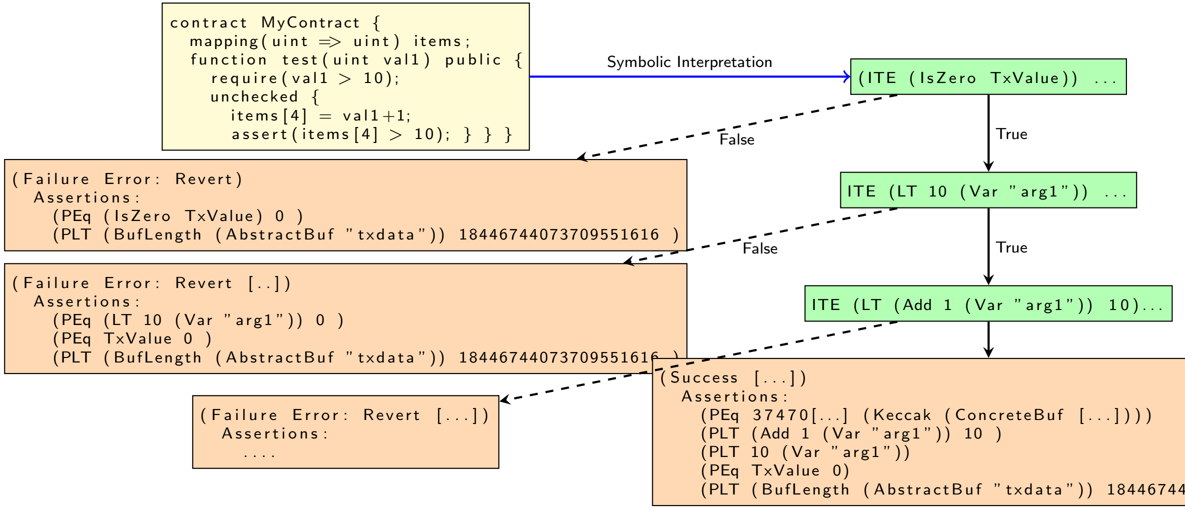
```
function overflow(uint a) public pure {
        uint b;
        unchecked { b = a + 1;}
        assert(b > a);
}
```

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



```
contract MyContract {
  mapping(uint => uint) items;
  function test(uint val1) public {
    require(val1 > 10);
    unchecked {
      items[4] = val1+1;
      assert(items[4] > 10); } } }
```

```
(ITE (IsZero TxValue)) ...
```

```
ITE (LT 10 (Var "arg1")) ...
```

```
ITE (LT (Add 1 (Var "arg1")) 10)...
```

```
(Failure Error: Revert)
  Assertions:
    (PEq (IsZero TxValue) 0 )
    (PLT (BufLength (AbstractBuf "txdata")) 18446744073709551616 )
```

```
(Failure Error: Revert [..])
  Assertions:
    (PEq (LT 10 (Var "arg1")) 0 )
    (PEq TxValue 0 )
    (PLT (BufLength (AbstractBuf "txdata")) 18446744073709551616 )
```

```
(Failure Error: Revert [...])
  Assertions:
    ....
```

```
(Success [...])
  Assertions:
    (PEq 37470[...] (Keccak (ConcreteBuf [...])))
    (PLT (Add 1 (Var "arg1")) 10 )
    (PLT 10 (Var "arg1"))
    (PEq TxValue 0)
    (PLT (BufLength (AbstractBuf "txdata")) 18446744
```

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.

Allows us to simply prepend an operation for each instruction.
We later collapse & simplify these writes.

Intermediate Representation: Keccak

Keccak, i.e. SHA-3 is used extensively by all contracts. This is because **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 input 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: $\mathit{keccak}(\mathit{bytes}32(\mathit{key})||\mathit{bytes}32(\mathit{id}))$ to map $\mathit{mymap}[\mathit{key}]$ where id is the map index:

```
contract C {
    mapping(uint => uint) map_id_0;
    mapping(uint => uint) map_id_1;
}
```

We have Solidity-specific rewrite rules to strip writes. So when writing two $\mathit{keccak}(\mathit{bytes}32(\mathit{key})||\mathit{bytes}32(\mathit{id}))$-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:

- Constant folding
- 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
- (In)equality propagation
- All meaningful rewrites, such as min/max/add+sub/add+add/sub+sub/etc.
- Special rewrite rules for array/map lookups and other Solidity patterns

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
```

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:

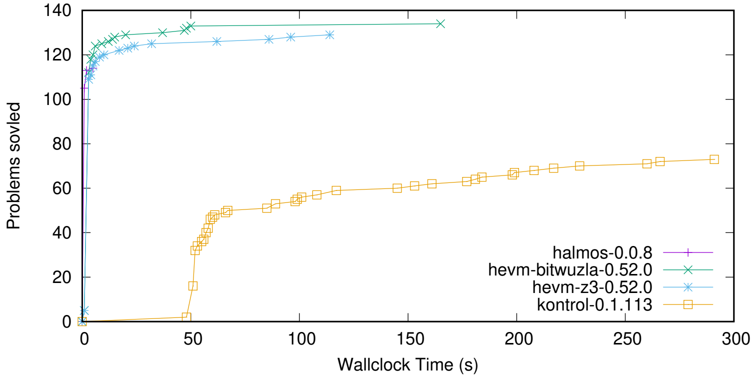
```
(set-logic QF_AUFBV)
(define-sort Word () (_ BitVec 256))
(declare-const varA (Word))
(assert (bvule (bvadd varA (_ bv1 256)) varA))
(check-sat)
```

Z3 gives the answer:

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

Results

We ran latest hevm, halmos, and kontrol (KEVM) on the Eth-SC Ethereum benchmarks [1], on an AMD 5950x with 5min timeout, and 128GB of RAM. For more details: $\rightarrow$ see paper!



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

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

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