A gentle introduction to formal verification of Ethereum smart contracts

Brussels 4th Summer School on Security Testing and Verification

Mate Soos

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

9th of July 2025

A bit about myself

- PhD from INRIA Grenoble, France
- Part-time research at Kuldeep Meel's group, authored::
  - **CryptoMiniSat** – SAT solver with XORs
  - **Ganak&ApproxMC** – CNF counters
  - **Arjun** – CNF simplifier
  - **Bosphorus** – ANF simplifier
  - **Pepin** – DNF counter
- Worked in IT Security for 10+ years: hacking, threat modelling, risk management
- Working at Ethereum Foundation, now Argot Collective for the past $\approx$ 3 years: **hevm**

Outline

What is a Blockchain – Transaction

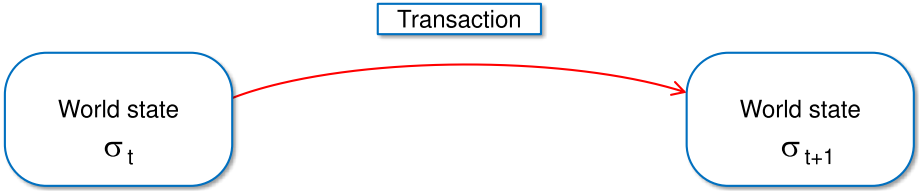


Illustration from: `takenobu-hs/ethereum-evm-illustrated`, BSD License

What is a Blockchain – Block

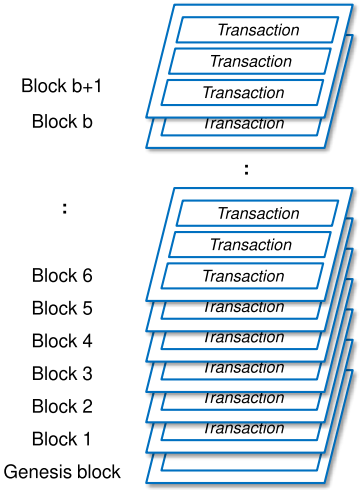


Illustration from: `takenobu-hs/ethereum-evm-illustrated`, BSD License

What is a Blockchain – Consensus

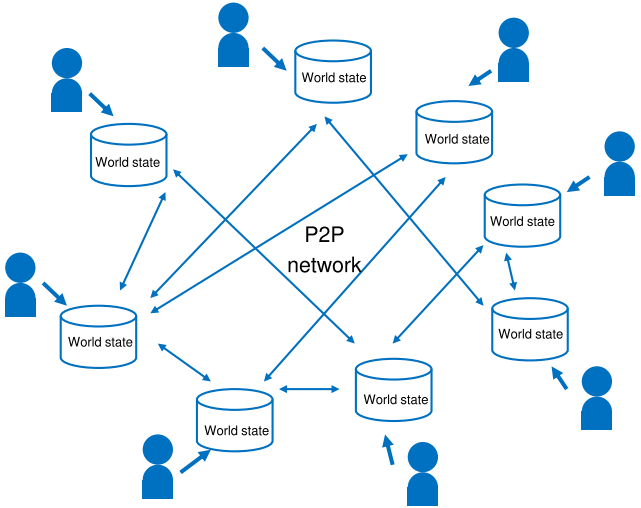


Illustration from: `takenobu-hs/ethereum-evm-illustrated`, BSD License

What is a Blockchain – Overview

A blockchain is a **distributed ledger** that is replicated across many nodes. It is:

- **Immutable** – once written, it cannot be changed
- **Append-only** – once written, it cannot be deleted
- **Transparent** – anyone can see the contents of the blockchain
- **Decentralized** – no single entity controls the blockchain
- **Consensus-based** – nodes agree on the state of the blockchain: PoW/PoS
- **Permissionless** – anyone can participate in the network (censorship resistant)
- **Trustless** – no need to trust any single entity, the system works as a whole

Immutable caveat: except when everyone agrees it should be undone... (see the DAO hack)

Ethereum Blockhain

You can think of it as a very fancy transition system that is:

- **Programmable** – it can execute code, i.e. smart contracts (via EVM)
- **Open-source** – the code is available for anyone to inspect and contribute (via GitHub)
- **Community-driven** – the development is driven by the community (e.g. ACD meetings)
- **Interoperable** – it can interact with other blockchains and systems (via bridges)
- **Ever-evolving** – it changes every 6mo via "hard-forks" driven by EIPs
- **Scaling via L2s** – used as a settlement layer for L2s: ZK, optimistic rollups

Currently, 30TPS on mainnet. Layer 2s can handle much more, but with lower guarantees.

What is Ethereum used for?

Decentralized:

- **Finance** – enables new financial systems and applications
- **IAM** – enables new ways of managing identity and access management
- **Governance** – enables new ways of managing and governing systems
- **Marketplaces** – enables new ways of buying and selling goods and services
- **Gaming** – enables new ways of playing and interacting with games

Some of these are simply re-packaging of existing concepts in a in a decentralized context. Here, the **permissionless** and **ACID** properties can sometimes come very handy.

Decentralized Applications (dApps) on Ethereum

- **MakerDAO** – stablecoin protocol
- **Uniswap, Suhiswap** – exchange for trading tokens
- **ENS** – domain name system for Ethereum addresses
- **Gitcoin** – platform for funding open-source projects
- **Aave, Compund** – lending and borrowing protocol
- **Tornado Cash** – ZK-based privacy mixer for transactions

Let's Explore

- UniSwap: `https://app.uniswap.org/`
- UniSwap Fees Collected: **dashboard** Dune analytics
- AAVE: `https://app.aave.com/markets/`
- ENS: `https://app.ens.domains/`
- Gitcoin: `https://impact.gitcoin.co/`
- MEV: `https://libmev.com/`
- Betting: `https://polymarket.com/`
- L2s: `https://l2beat.com/scaling/summary`
- dApps: `https://www.growthepie.com/applications`

Use-case: Flashloans

Flashloans are a type of loan that allows you to borrow assets without collateral, as long as you return the assets within the same transaction.

Remember, it's ACID, so:

- User asks contract for a flashloan + provides functions to call
- Contract blindly executes the functions, with 1B+ USD in assets
- Contract checks that the loan is repaid at the end of the transaction
- If the loan is not repaid, the transaction is reverted

Can be used for:

- Arbitrage – taking advantage of price difference between exchanges
- Liquidation – borrow, liquidate collateral, immediately sell, repay loan
- Collateral swapping – swapping collateral in a loan

Use-case: Flashloans – Example

Example flashloan transaction via **BlockSec Explorer**

Notice:

- Flash loan function is `FlashLoanLogic.executeFlashLoanSimple`
- User-defined action is executed in the `executeOperation`
- Ends with `handleRepayment` which checks that the loan is repaid
- Notice: `transferUnderlyingTo` was ran with 47352823905004708422332 wei – currently around 140M USD
- Repayment was 47376500316957210776543 wei – difference is 70K USD called the `premium`

Etherscan **view of the transaction.** Net "profit" for the "attacker": 7M USD

Use-case: Tornado Cash

Tornado Cash is a privacy-focused smart contract that allows users to send ETH and ERC20 tokens anonymously. It uses zero-knowledge proofs to ensure that the sender and receiver addresses are not linked, while still allowing the network to verify that the transaction is valid.

- Tornado Cash 0.1 ETH **etherscan view**
- Example **transaction** → Click "decode input data"
- ZK proof: verifier can check that prover knows the secret without revealing it
- Think of it as RSA: you can verify that someone knows the private key without them revealing it.

The Layers of Ethereum

- **Execution layer** – executes the code of smart contracts as triggered by transactions. Performed by the EVM
- **Consensus layer** – ensures all nodes agree on the state of the blockchain. Currently managed via proof-of-stake.
- **Data Availability Layer** – ensures that transaction data is available (e.g. via data availability sampling)
- ****Application layer** – provides the user interface and interaction with the blockchain (dApps)

The execution layer is the most important for us, as it is where the smart contracts are executed.

EVM

EVM executes the code of smart contracts and updates the state of the blockchain:

- Stack machine – instead of registers, it uses a stack
- 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
- Can call other contracts, allowing for composability
- ACID-compliant – provides atomicity, consistency, isolation, and durability

Note: **Solana uses eBPF**, which was mentioned during the static code analysis seminar

EVM Overview

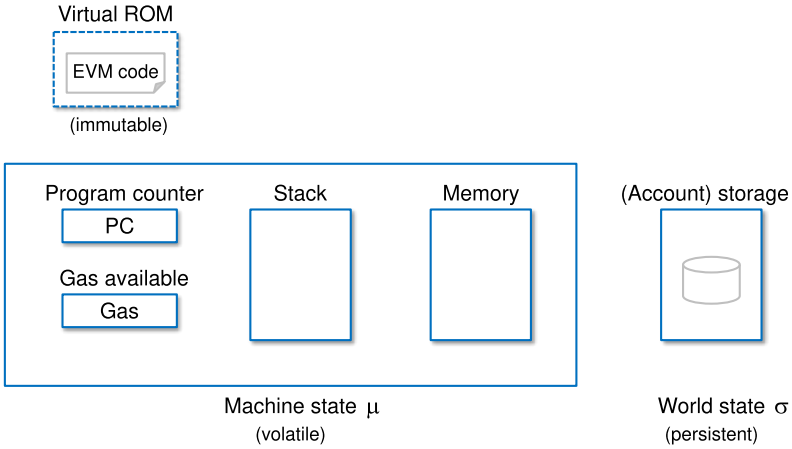


Illustration from: `takenobu-hs/ethereum-evm-illustrated`, BSD License

EVM Instructions Overview

The EVM has a set of instructions that can be used to manipulate the state of the blockchain.

Instruction categories:

- **Arithmetic/Logic Ops** – basic operations like add, multiplication, div, ...
- **Control Flow Ops** – jumps, program counter, halting, revert, ...
- **Stack/Memory/Storage Ops** – stack manipulation, memory&storage access, ...
- **External Calls and Contracts** – calling contracts, transferring value, creating contracts
- **Context Ops** – accessing block context, gas information, contract information, logging

Let's check out `https://www.evm.codes/`

Let's try 8/8 and then 2/0 (i.e. division by zero), in the **EVM Playground**.
Load in Bytecode: 60086008046002600004

EVM vs x86 ADD

EVM **add instruction**:

- Takes two numbers from the stack, adds them mod 2^{256}, pushes the result to the stack
- If it runs out of gas, reverts the frame

The OF, SF, ZF, AF, CF, and PF flags are set according to the result. x86 **add instruction**:

- Depending on the variation of the opcode, it'll add different registers together
- Depending on the variation of the opcode, it will add them over different bit sizes
- The OF, SF, ZF, AF, CF, and PF flags are set according to the result
- It might throw an exception

Q: Which one is easier to reason about?

EVM Instructions Part 1

Arithmetic/Logic (ALU) Operations

- Basic arithmetic: `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `EXP`, `SIGNEXTEND`
- Bitwise operations: `AND`, `OR`, `XOR`, `NOT`
- Comparisons: `LT`, `GT`, `EQ`, `ISZERO`, `SLT`, `SGT`
- Shift operations: `SHL`, `SHR`, `SAR`
- Special operations: `KECCAK256`

Control Flow Operations

- Jumps: `JUMP`, `JUMPI`
- Program counter: `PC`
- Halting: `STOP`, `RETURN`, `REVERT`

Control Flow, Code and Data Separation

- You can only jump to already deployed code – no dynamic code generation
- Code points that can be jumped to are marked with `JUMPDEST`
- Code is immutable, but we can `JUMP`/`JUMPI` to a dynamically computed position
- So computing a Control Flow Graph (CFG) can be hard/impossible
- It's possible to jump into the middle of a PUSH instruction, if the PUSH-ed data happens to match the JUMPDEST opcode
- So it's data and code separation is not as clear-cut

EVM Instructions Part 2

Stack/Memory/Storage Operations

- Stack manipulation: `PUSH`, `POP`, `DUP`, `SWAP`
- Memory access: `MLOAD`, `MSTORE`
- Storage: `SLOAD`, `SSTORE`, `TSLOAD`, `TSSTORE`
- Calldata: `CALLDATALOAD`, `CALLDATASIZE`, `CALLDATACOPY`
- Return data: `RETURNDATASIZE`, `RETURNDATACOPY`
- Code access: `CODECOPY`, `EXTCODECOPY`

External Calls and Contracts

- Call ops: `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`
- Value transfer: `CALLVALUE`, `TRANSFER`
- Contract creation: `CREATE`, `CREATE2`, `SELFDESTRUCT`

EVM Instructions Part 3

Context

- Block Context: `COINBASE`, `TIMESTAMP`, `NUMBER`, `PREVRANDAO`, `ADDRESS`, `BALANCE`, `ORIGIN`, `CALLER`, `BLOCKHASH`, `DIFFICULTY`, `BLOBHASH`, `BLOBBASEFEE`, `CHAINID`
- Gas: `GAS`, `GASLIMIT`, `GASPRICE`
- Contract information: `EXTCODESIZE`, `EXTCODEHASH`, `MSIZE`
- Logging: `LOG`

ABI

The Application Binary Interface (ABI) is the interface between smart contracts and external applications. It defines how to encode and decode data for function calls, events, and errors.

The ABI is used to:

- Encode function calls and parameters
- Decode return values
- Handle errors

ABI function call

- Function signature is a 4-byte hash of the function name and its parameter types
- Function parameters follow, with packed encoding (e.g. 4 uint64-s will be packed into 32 bytes)

Example:

```
$ cast keccak "get()"
0x6d4ce63caa65600744ac797760560da39ebd16e8240936b51f53368ef9e0e01f
```

Hence, the function signature for `get()` is `0x6d4ce63c`. So:

```
(bool success, bytes memory data) = address(myContract).call("get()");
```

will generate the calldata `0x6d4ce63c` as the first 4 bytes, followed by 0 bytes for the empty parameters.

ABI error/panic

```
(bool success, bytes memory data) = address(myContract).call(
    abi.encodeWithSignature("myfun(uint256)", 42));
if (!success) {
    bytes4 errorSelector;
    assembly { errorSelector := mload(add(data, 0x20)) }
    if (errorSelector == bytes4(keccak256("Error(string)"))) {
        string memory reason = abi.decode(data, (string));
        // reason is a string, e.g. "revert"
    } else if (errorSelector == bytes4(keccak256("Panic(uint256)"))) {
        uint256 code = abi.decode(data, (uint256));
        // Panic code is a uint256, e.g. 0x11
    }
}
```

EVM: Panic Codes

- 0x00: Generic compiler inserted panics.
- 0x01: Call assert with an argument that evaluates to false.
- 0x11: Arithmetic operation results in underflow or overflow
- 0x12; Divide or modulo by zero (e.g. 5 / 0 or 23 % 0).
- 0x21: Convert a value that is too big or negative into an enum type.
- 0x22: Access a storage byte array that is incorrectly encoded.
- 0x31: Call .pop() on an empty array.
- 0x32: Access an array, bytesN or an array slice at an out-of-bounds or negative index (i.e. $x[i]$ where $i >= x.length$ or $i < 0$).
- 0x41: Allocate too much memory or create an array that is too large.
- 0x51: Call a zero-initialized variable of internal function type.

Programming the EVM

We use Solidity to program the EVM, with YUL inline assembly:

- **Imperative**, high-level language -– Code is executed step-by-step (like C++).
- **Statically typed** – Variable types must be declared at compile time (e.g., uint, string).
- **Object-oriented** – Supports contracts (≈classes), inheritance, interfaces, and libraries.
- **Smart contract focus** – Designed for writing decentralized apps (dApps)
- **Special features** – Built-in globals (e.g., msg.sender), ABI en&decoding

`https://docs.soliditylang.org/en/v0.8.30/`

Solidity vs EVM bytecode

- Solidity is a high-level language that compiles to EVM bytecode
- Provides a more readable and maintainable way to write contracts
- EVM bytecode is what is actually executed by the EVM
- The two are **connected** via a hash that's added at the end of the bytecode
- Hash, along with the contract code can be submitted to e.g. sourcify.eth
- Click on **sourcify.eth** and look up `0x1F98431c8aD98523631AE4a59f267346ea31F984` – the UniSwapv3 contract
- Click `View in Sourcify Repository` to see the source code

Online development environment: Remix

Built-in:

- Solidity compiler
- Test deployment environment
- Debugger
- Unit testing
- GitHub integration

Remix Online IDE: Load a **simple Remix GitHub project**
Check tests: `a.complicated(10,9)` could be made to fail.

Solidity: Special Functions: constructor, receive, fallback

You can define special functions in Solidity that will be called when the contract is **created**, when the contract **receives Ether** without a function call, or when a **function is not found**,

```
interface RemoteContract {
    function withdraw() external;
}
contract Attacker{
    RemoteContract remote;
    uint receives = 0;
    uint fallbacks = 0;

    constructor(address addr) { remote = RemoteContract(addr); }
    receive() external payable { receives++; }
    fallback() external { fallbacks++; }
}
```

Solidity: Special Functions: constructor, receive, fallback – part 2

Who sees the issues with this?

Solidity: Special Functions: constructor, receive, fallback – part 2

Who sees the issues with this?

Can't just send Ether to the contract, it might take control of execution

- It might call us back! Reentrancy attack.
- It might call other contracts we rely on, changing their state
- It might manipulate the cost of some asset whose price we already calculated/saved

All calls&transfers to external contracts require **careful consideration**.

The DAO Hack

The DAO hack (in 2016) was a major event in Ethereum's history, where an attacker exploited a **reentrancy vulnerability** in the DAO contract to steal $50 million worth of Ether.
The hack led to a hard fork of Ethereum, creating Ethereum Classic (ETC) and Ethereum (ETH).

The DAO Hack – Let's do it!

Let's recreate the DAO hack in Remix!

1. Load **this GitHub repo** into Remix
2. Compile & Deploy **FundRaiser**
3. Compile & Deploy **Attacker**, with **FundRaiser** address as parameter
4. Set **Value** to 10000 and call **deposit** on Fundraiser. This will deposit money on YOUR balance
5. Call **getTotalFunds** on Fundraiser. Should show 10000.
6. Set **Value** to 400 and call **depositFunds** on Attacker. This will deposit money on the ATTACKER's balance
7. Call **getTotalFunds** on Fundraiser. Should show 10400.
8. Call **getFunds** on Attacker. Should show 0. The attacker has 0 funds currently.
9. Call **withdrawFunds** on Attacker. This is the attack.
10. Call **getFunds** on Attacker. Should show 1600 – 4x what the Attacker deposited!

Example: Tornado Cash's Reentrancy Guard

Uses a decorator **ReentracyGuard** to prevent reentrancy attacks:

The actual implementation of the guard is by **openzeppelin**

Simple solution: `_status` variable set to $1/2$ depending on whether we have entered the function or not. If we try to enter the function again, we fail the `_status == ENTERED` check, and revert the transaction.

What is a bug anyway?

EVM:

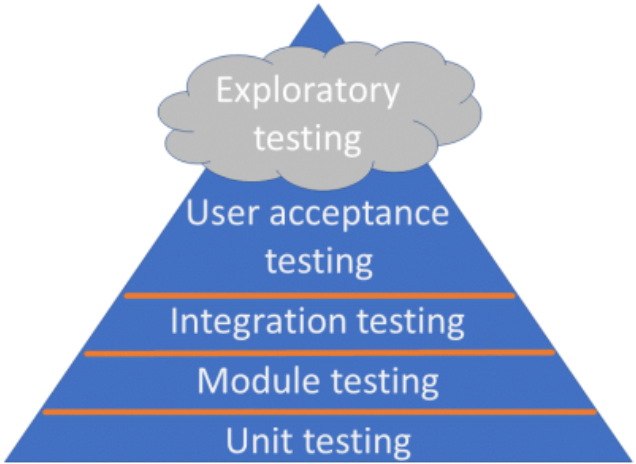
- Steal the money due to e.g. overflow/reentrancy/logical error/etc bug
- Lock in the money due to liveness issues
- Exploit incorrect rounding in online exchanges to drain contract
- Borrow money to buy some asset, dump it in an automated market maker pushing its price to zero, then buy everything for zero.

Consensus:

- **Frontrunning**: observe a money-making transaction, replace the recipient of the transaction with my own address
- **Sandwiching**: see 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. Block is **here**, observe the `MEV Bot` before and after the `0xee9fc` transaction. (**X thread**, **another** example)

Some of these are "clearly" bugs. Some of them are abuse of systems(frontrunning). Some are games played according to the rules, but perhaps not according to the spirit.

The Testing Pyramid



The Testing Continued

- Things get more expensive as you go up the pyramid
- Test types
  - Positive test: correct password – should log in
  - Negative test: incorrect password – should not log in
  - Invariant test: total balance of all accounts should not change
- Exploratory testing:
  - manual review: auditing by a human
  - fuzz testing: "randomly" generate inputs to find bugs
  - symbolic execution: compute all possible inputs to find bugs

Fuzzing

You just had a long lecture so I won't go into details :) Tools in Ethereum:

- **Echidna**: Coverage-guided fuzzer for Ethereum smart contracts, via **hevm**
- **Forge**: Built-in fauzzer as part of Foundry, only recently coverage-guided, via **reth**
- **Medusa**: Fuzzer for Ethereum smart contracts via fork of **geth**

Why specialized execution engines? Because we need:

- **Cheat Codes** – special functions to **manipulate the world state**
  - **deal** – gives ETH to an address
  - **prank** – changes the sender of the transaction
  - **roll** – changes the block number
  - **...**
- **Coverage Metrics** – we need to know which parts of the code were executed

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.

Path Explosion

hevm struggling with path explosion

Potential solutions:

- **Path pruning** – dispatch partial path conditions, stop exploring paths that are already known to be infeasible
- **Path merging** – merge paths that have the same state
- **Lookahead** – check if any assert failure is reachable from this path at all

Just some insight: **halmos' todo list**

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, uses CVL specification language
- **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

Satisfiability Modulo Theories (SMT) Solvers

What is an SMT solver?

- Extends SAT solvers by supporting **first-order logic with theories** (e.g., arithmetic, arrays, bit-vectors)
- Decides the satisfiability of logical formulas over **background theories**

Key Applications:

- Formal verification (hardware/software)
- Program analysis (symbolic execution)
- Automated theorem proving

Example Theory Combinations:

- Bit-vectors: $x \ll 2 == 0b1000$
- Arrays: $A[i] = A[j]$
- Uninterpreted functions: $f(x) = f(y)$

SMT Solving in Practice

Example Query (Using Z3 Syntax):

```
(declare-const x Int)
(declare-const y Int)
(assert (>= x 0))
(assert (>= y 0))
(assert (= (+ (* 2 x) y) 10))
(check-sat)
(get-model)
```

Result: Solver finds a satisfying model, e.g., $x = 5$, $y = 0$

Why SMT?

- Combines **SAT** efficiency with **domain-specific reasoning**
- Theories to model: bitvectors, arrays, arithmetic, and uninterpreted functions (AUFBV)

SMT solver hands-on: z3 in the browser

You can play with **z3 in your browser**

Example bitvector query:

```
(set-logic QF_BV)
(declare-const x (_ BitVec 8))
(declare-const y (_ BitVec 8))
(assert (= (bvadd x y) (_ bv7 8)))
(check-sat)
(get-model)
```

Let's require that one is > 7:

```
(assert (bvugt x (_ bv7 8)))
```

hevm: How to define what's a bug?

- We decided NOT to enforce any "opinions" on what a bug is
- We don't have a specification language, unlike e.g. Certora Prover's CVL
- The spec has to be expressed in-line code via require/assert
- If used as a library, interface exists to specify pre/post-conditions

This way we allow others to be opinionated, but we ourselves remain unopinionated.
But: it misses the ability to express some invariants, e.g. "total balance of all accounts should not change".

Certora Verification Language (CVL)

Certora Prover uses a specification language called CVL to express properties of the contract. CVL is a domain-specific language for writing specifications of smart contracts. It allows you to express properties of the contract in a declarative way, and then prove that the contract satisfies these properties.

Example **CVL specification** for SQRT approximation over fixed-point numbers.

Symbolic Execution: Example

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*.

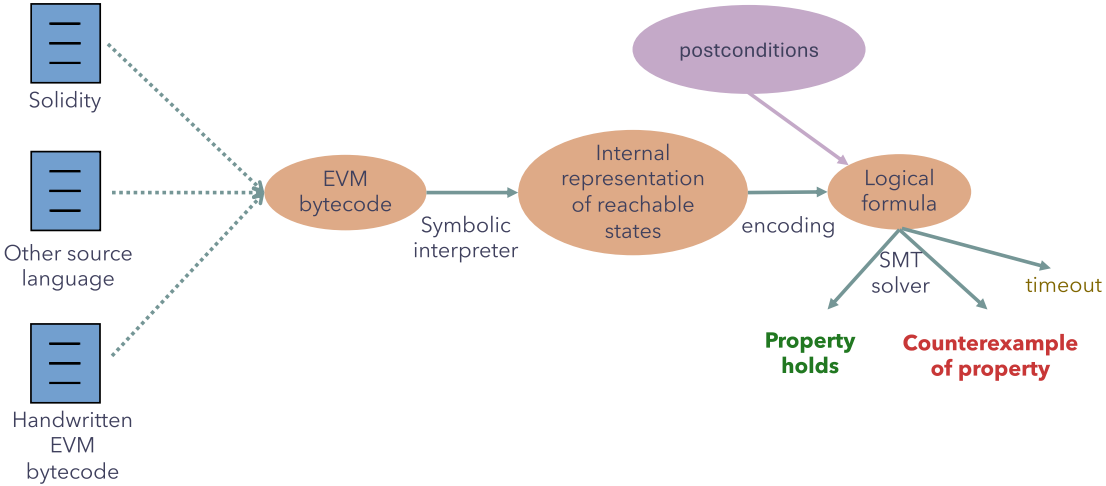
Symbolic Execution: Invariant Checking

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

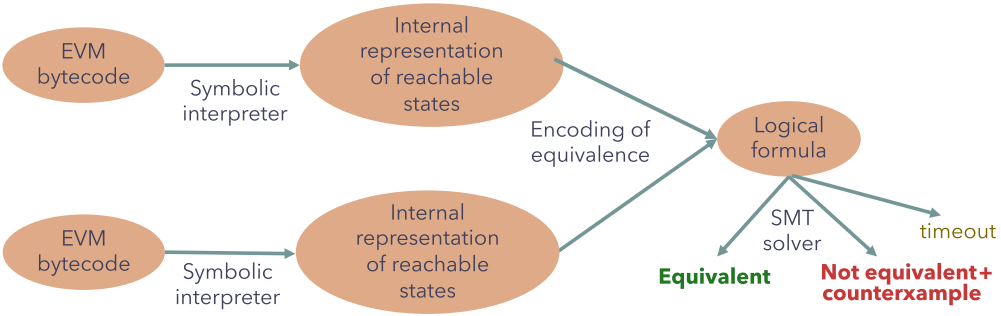
```
function my_invariant() private pure returns inv {
        inv = ...calculate invariant...
}

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.

hevm: Symbolic Execution for Counterexample Generation



hevm: Symbolic Execution for Equivalence Checking



hevm: Symbolic Execution for 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 the exact **call to trigger the discrepancy** between the two functions. This way, you can safely improve the gas performance of your code.

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

Under- and Overapproximation

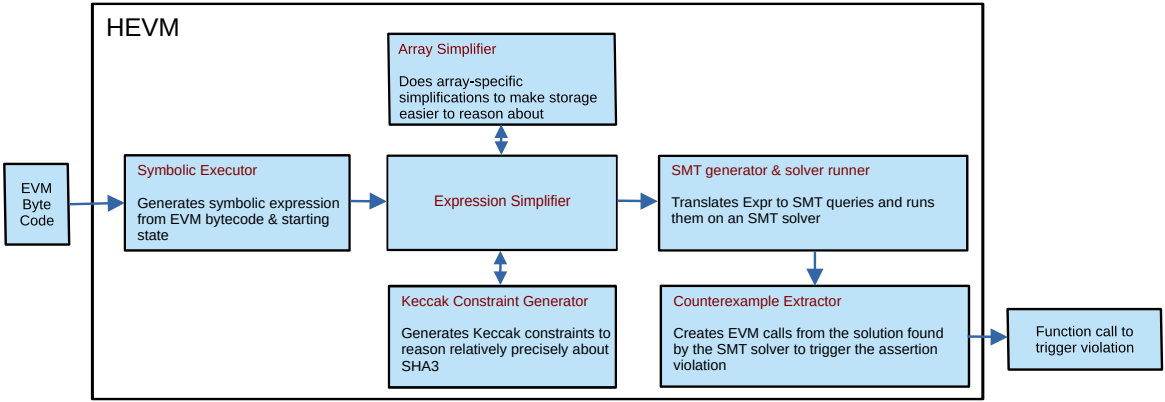
Underapproximation:

- We may **miss a counterexample** that is reachable in the contract.
- Example: `CALL` to unknown code gives up exploration, **prints warning**
- This would be a **soundness** issue, hence we **always warn**

Overapproximation:

- We may return a counterexample that is **not actually reachable** in the contract.
- Example: `STATICCALL` to unknown code returns a fresh buffer and return value.
- `STATICCALL` cannot change state, so this is a valid overapproximation.
- Note: computing an **invariant of a loop** using CHC would also be an overapproximation

hevm internals



Intermediate Representation: Code to Expression

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:

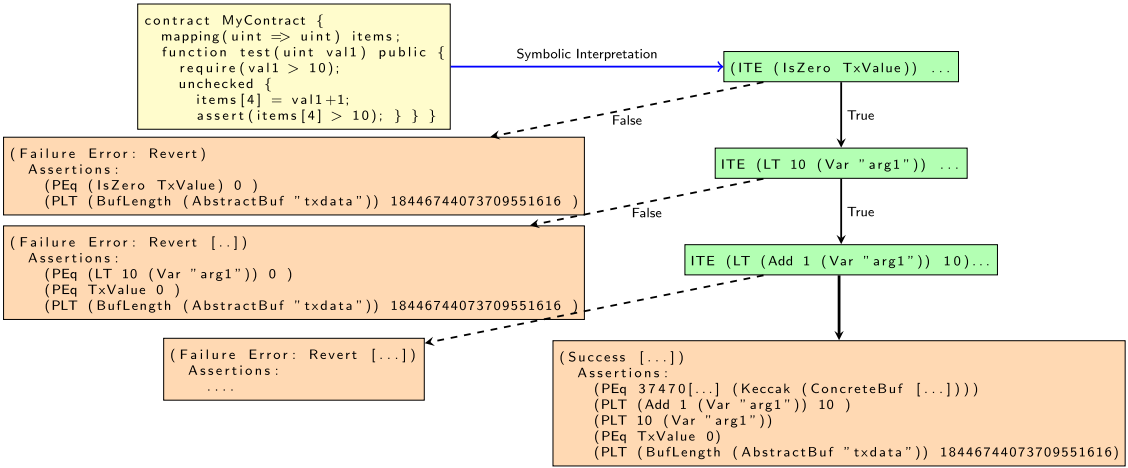
```
function overflow(uint a) public pure {
    uint b;
    unchecked { b = a + 1;}
    assert(b > a);
}
```

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: Graph of Expressions



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: $keccak(bytes32(key)||bytes32(id))$ to map $mymap[key]$ 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 $keccak(bytes32(key)||bytes32(id))$-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:

- 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
```

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
- Review by humans: the IR is a clean representation of the problem

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)
(get-model)
```

Z3 gives the answer:

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

Example Solidity Test

```
import {Test} from "forge-std/Test.sol";
contract MyContractOverflow is Test {
  uint balance = 100;
  function prove_overflow(uint amt) public {
    unchecked { balance += amt; }
    assert(balance >= amt);
  } }
```

```
Checking 1 function(s) in contract src/overflow-test.sol:MyContractOverflow
[RUNNING] prove_overflow(uint256)
   Exploring call prefix 0x7a400a2b
   Exploration finished, 3 branch(es) to check in call prefix 0x7a400a2b
   SMT result: Cex SMTCex {vars = fromList [(Var "arg1",0xfffffffffff...
   Found 1 potential counterexample(s) in call prefix 0x7a400a2b
   [FAIL] prove_overflow
   Counterexample:
     calldata: prove_overflow(115792089237316195423570985008687907853269984665640564039457584
```

Example Solidity Test: IR + SMT

IR for the counterexample is:

```
(PLT
  (Add
    100
    (Var "arg1")
  )
  (Var "arg1")
)
```

SMT expression:

```
(set-logic QF_AUFBV)
(define-sort Word () (_ BitVec 256))
(declare-fun arg1 () (_ BitVec 256))
(assert (bvult (bvadd (_ bv100 256) arg1) arg1))
(check-sat)
(get-model)
```

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:

Contributing Back to hevm

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

All PRs are welcome. Haskell can be a bit intimidating, but it's a very expressive language

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
- Better handling of loops and recursion: include into IR, and solve for invariant via CHC

Quick Start Guide

Development environment:

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

You now have in the shell:

- hevm via cabal: `cabal run exe:hevm -- test/symbolic/equivalance`
- SMT solvers: Z3, bitwuzla, cvc5
- Ethereum development environment: solc, geth, evm, forge

```
cabal run exe:hevm -- symbolic --create --code "60806040523480..."
```

Thank you for your time!

Any questions?