runtime verification

The K Framework

A tool kit for language semantics and verification

PM Session

Smart Contract Verification with kontrol

NAME

21st International Symposium on Automated Technology
for Verification and Analysis (ATVA 2023), DATE

PM Session Overview

runtime verification

- Quick Intro/Recap to Blockchain, Smart Contract and EVM
- Smart Contract Tooling and Testing
- Symbolic execution using **kontrol**
- **kontrol** Hands-on

Github repository for all materials

https://github.com/runtimeverification/k-tutorial-atva-2023

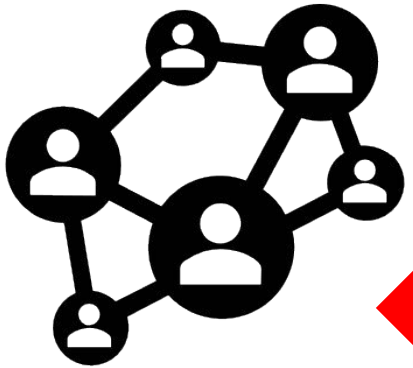
runtime verification

Quick Intro/Recap

Blockchain

runtime verification

Managed by **peer-to-peer network**

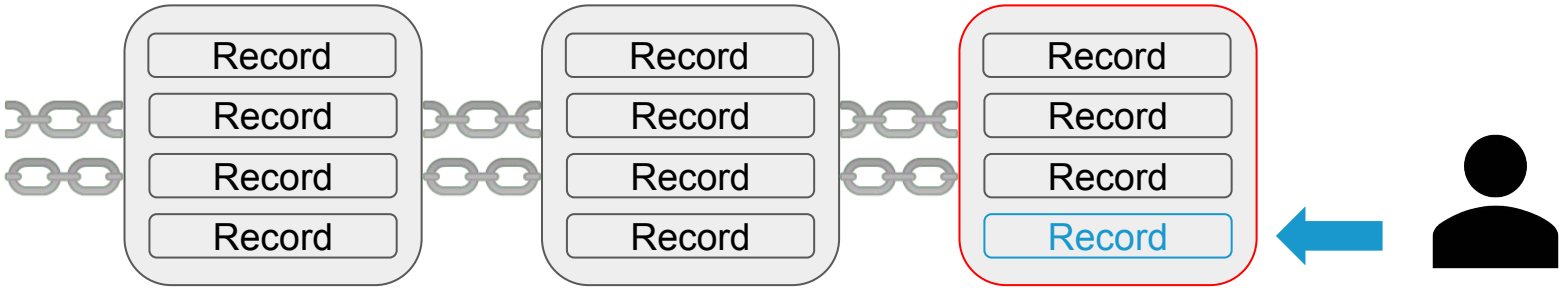


Which records to be included?

Consensus mechanism

Who can propose records?

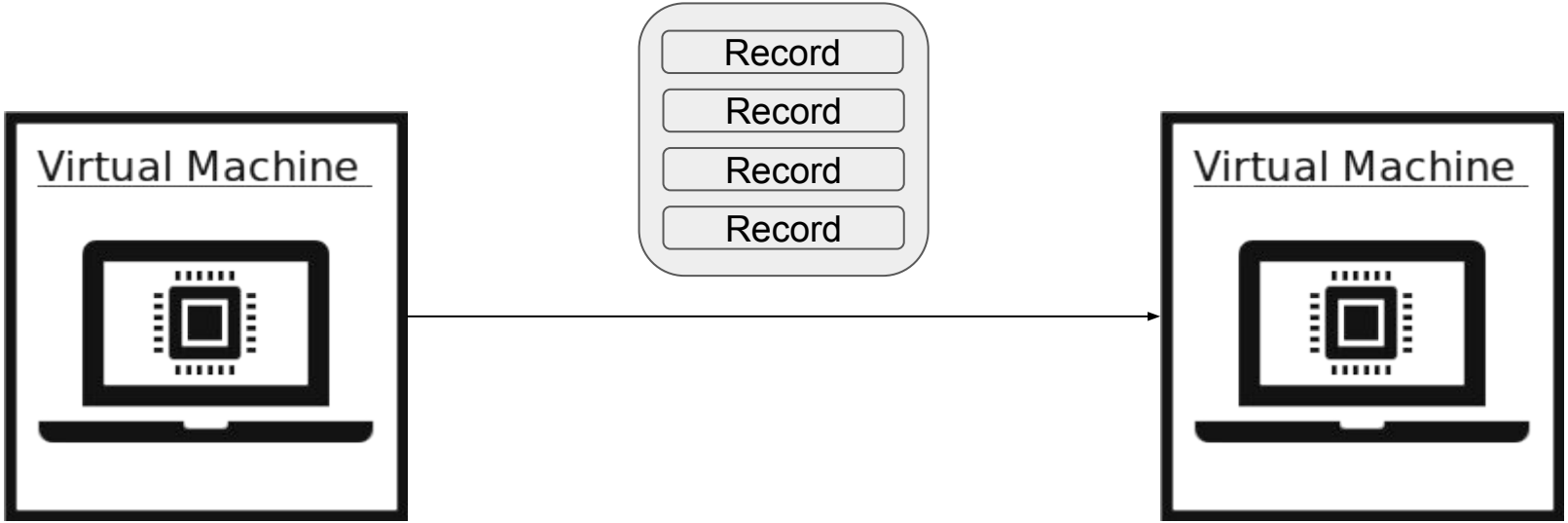
Permissionless => Anyone



From Wiki: A **blockchain** is a **distributed** ledger with growing lists of records (blocks) that are securely linked together via cryptographic hashes.

Blockchain updates VM

runtime verification



Record	Virtual Machine
Transaction	Universal Ledger (e.g., Bitcoin)
Operation from smart contract	Universal Virtual Machine (e.g., Ethereum)

EVM

Ethereum Smart Contract

runtime verification

Solidity

```
contract TetherToken is Pausable, StandardToken, BlackList {

    string public name;
    string public symbol;
    uint public decimals;
    address public upgradedAddress;
    bool public deprecated;

    //  The contract can be initialized with a number of tokens
    //  All the tokens are deposited to the owner address
    //
    // @param _balance Initial supply of the contract
    // @param _name Token Name
    // @param _symbol Token symbol
    // @param _decimals Token decimals
    function TetherToken(uint _initialSupply, string _name, string _symbol, uint _decimals) public {
        _totalSupply = _initialSupply;
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        balances[owner] = _initialSupply;
        deprecated = false;
    }

    // Forward ERC20 methods to upgraded contract if this one is deprecated
    function transfer(address _to, uint _value) public whenNotPaused {
        require(!isBlackListed[msg.sender]);
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).transferByLegacy(msg.sender, _to, _value);
        } else {
            return super.transfer(_to, _value);
        }
    }

    // Forward ERC20 methods to upgraded contract if this one is deprecated
    function transferFrom(address _from, address _to, uint _value) public whenNotPaused {
        require(!isBlackListed[_from]);
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).transferFromByLegacy(msg.sender, _from, _to, _value);
        } else {
            return super.transferFrom(_from, _to, _value);
        }
    }
```

Opcode

```
PUSH1 0x60
PUSH1 0x40
MSTORE
PUSH1 0x04
CALLDATASIZE
LT
PUSH2 0x0196
JUMPI
PUSH1 0x00
CALLDATALOAD
PUSH29 0x0100000000000000000000000000000000000000000000000000000000
SWAP1
DIV
PUSH4 0xffffffff
AND
DUP1
PUSH4 0x06fdde03
EQ
PUSH2 0x019b
JUMPI
DUP1
PUSH4 0x0753c30c
EQ
PUSH2 0x0229
JUMPI
DUP1
PUSH4 0x095ea7b3
EQ
PUSH2 0x0262
JUMPI
DUP1
PUSH4 0x0e136b19
EQ
PUSH2 0x02a4
JUMPI
DUP1
PUSH4 0x0ecb93c0
EQ
PUSH2 0x02d1
JUMPI
DUP1
```

Permissionless => Issues

runtime verification

Permissionless

Anyone can write a smart contract and deploy it to the network

⚠️ Security issues ⚠️

runtime verification

Smart contract Tooling and Testing

Property Testing Tools

runtime verification

Increase in coverage

Unit testing	TRUFFLE SUITE Hardhat
Fuzz testing	Foundry ERCX
Formal verification	K kontrol

The Foundry Toolkit

runtime verification

- Very **easy** to use - write property tests and test.
- Supports **fuzzing** over parametric tests at the Solidity level.
- Blazing **fast** execution, providing users with immediate feedback.
- There are no false positives[1], but it can produce false negatives[2].

[1]Tests that are supposed to pass will pass.

[2]Some tests that are supposed to fail may pass because the fuzzing test cannot find counterexamples within the limited number of runs.

Foundry Cheat codes

runtime verification

```
54      function startPrank(address) external;
55      // Sets the *next* call's msg.sender to be the input address, and the tx.origin to be the
        second input
56      function prank(address,address) external;
57      // Sets all subsequent calls' msg.sender to be the input address until `stopPrank` is
        called, and the tx.origin to be the second input
58      function startPrank(address,address) external;
59      // Resets subsequent calls' msg.sender to be `address(this)`
60      function stopPrank() external;
61      // Sets an address' balance, (who, newBalance)
62      function deal(address, uint256) external;
63      // Sets an address' code, (who, newCode)
64      function etch(address, bytes calldata) external;
65      // Expects an error on next call
66      function expectRevert(bytes calldata) external;
67      function expectRevert(bytes4) external;
68      function expectRevert() external;
```

- A Solidity Interface that contains function signatures
- Give developers the ability to alter the state of the EVM from their own Solidity tests

Foundry Fuzz Testing

runtime verification

```
function testBalanceOf(address addr, uint256 amount) public {
    ERC20 erc20 = new ERC20("Bucharest Workshop Token", "BWT");
    bytes32 storageLocation = getStorageLocationForKey(addr, BALANCES_STORAGE_INDEX);
    vm.assume(uint256(vm.load(address(erc20), storageLocation)) == amount);
    uint256 balance = erc20.balanceOf(addr);
    assertEq(balance, amount);
}
```

- Foundry would fuzz the values of the function parameters.

Foundry Fuzz Testing Output

runtime verification

```
anvacaru@desktop foundry-demo$ FOUNDRY_FUZZ_RUNS=300 forge test
[⠒] Compiling...
No files changed, compilation skipped

Running 8 tests for test/ERC20.t.sol:ERC20Test
[PASS] testBalanceOf(address,uint256) (runs: 300, μ: 510340, ~: 510340)
[PASS] testName() (gas: 508343)
[PASS] testSymbol() (gas: 508387)
[PASS] testTotalSupply(uint256) (runs: 300, μ: 509813, ~: 509813)
[PASS] testTransferFailure_0(address,uint256) (runs: 300, μ: 510167, ~: 510167)
[PASS] testTransferFailure_1(uint256) (runs: 300, μ: 509723, ~: 509723)
[PASS] testTransferFailure_2() (gas: 513545)
[FAIL. Reason: The `vm.assume` cheatcode rejected too many inputs (65536 allowed)]
,address,uint256,uint256,uint256) (runs: 9, μ: 516918, ~: 516918)
Test result: FAILED. 7 passed; 1 failed; finished in 3.59s
```

But recall that …

Some tests that are supposed to fail may pass because the fuzzing test cannot find counterexamples within the limited number of runs.

Symbolic execution using kontrol

What is KEVM?

runtime verification

- KEVM = Formal semantics of the Ethereum Virtual Machine in K Framework.
  - Passes same conformance test-suite as other clients.
  - Enables symbolic execution (and thus verification) of EVM bytecode.
  - Online: https://jellopaper.org or https://github.com/runtimeverification/evm-semantics

- Defined using a configuration and transition rules.
- Used by Runtime Verification in formal engagements.
- Large-scale proving with K and ACT (from Multi-Collateral Dai system - 1011 proofs)

KEVM Former Approach

runtime verification

```
6     // balanceOf
7     claim
8       <kevm>
9       <k> #execute => #halt </k>
10
11      <ethereum>
12        <output> _ => #buf(32, BAL) </output>
13        <statusCode> _ => EVMC_SUCCESS </statusCode>
14
15        <id> ACCT_ID </id> // contract owner
16        <caller> CALLER_ID </caller> // who called this contract; in the begining, origin // msg.sende
17
18        <callData> #abiCallData("balanceOf", #address(OWNER)) </callData>
19
20        <account>
21          <acctID> ACCT_ID </acctID>
22          <balance> _ </balance>
23          <code> #parseByteStack("0x6060604052600436106100d576000357c0100000000000000000000000000000000
24          <storage> #hashedLocation("Solidity", 1, OWNER) |-> BAL
25                    _:Map
26          </storage>
27        </account>
100     requires 0 <=Int ACCT_ID     andBool ACCT_ID     <Int (2 ^Int 160)
101      andBool 0 <=Int CALLER_ID   andBool CALLER_ID   <Int (2 ^Int 160)
102      andBool 0 <=Int ORIGIN_ID   andBool ORIGIN_ID   <Int (2 ^Int 160)
103      andBool 0 <=Int CALL_DEPTH  andBool CALL_DEPTH  <Int 1024
104      andBool 0 <=Int OWNER       andBool OWNER       <Int (2 ^Int 160)
105   andBool 0 <=Int BAL         andBool BAL         <Int (2 ^Int 256)
```

```
contract ERC20 {

    mapping(address => uint256) private _balances;

    function balanceOf(address account) external view returns (uint256) {
        return _balances[account];
    }
```

Provide formal verification
BUT
difficult to write proofs

KEVM Former Approach

```
requires "../verification.k"

module BALANCEOF-SPEC
  imports VERIFICATION

  // balanceOf
  claim
    <kevm>
    <k> #execute => #halt </k>
    <exit-code> 1 </exit-code>
    <mode> NORMAL </mode>
    <schedule> ISTANBUL </schedule>

    <ethereum>
      <evm>
        <output> _ => #buf(32, BAL) </output>
        <statusCode> _ => EVMC_SUCCESS </statusCode>
        <callStack> _ </callStack>
        <interimStates> _ </interimStates>
        <touchedAccounts> _ => ?_ </touchedAccounts>

        <callState>
          <program> #parseByteStack("0x6060604052600436106100 6d576000357c010000000000000000000000000000000000000000000000000000
          <jumpDests> #computeValidJumpDests(#parseByteStack("0x6060604052600436106100 6d576000357c010000000000000000000000000000000

          <id> ACCT_ID </id> // contract owner
          <caller> CALLER_ID </caller> // who called this contract; in the begining, origin // msg.sender

          <callData> #abiCallData("balanceOf", #address(OWNER)) </callData>

          <callValue> 0 </callValue>
          <wordStack> .WordStack => ?_ </wordStack>
          <localMem> .Bytes => ?_ </localMem>
          <pc> 0 => ?_ </pc>
          <gas> #gas(_VGAS) => ?_ </gas>
          <memoryUsed> 0 => ?_ </memoryUsed>
          <callGas> _ => ?_ </callGas>

          <static> false </static> // NOTE: non-static call
          <callDepth> CALL_DEPTH </callDepth>
        </callState>

        <substate>
          <selfDestruct> _ </selfDestruct>
          <log> _ </log>
          <refund> _ </refund> // TODO: more detail
          <accessedAccounts> _ => ?_ </accessedAccounts>
          <accessedStorage> _ => ?_ </accessedStorage>
        </substate>

        <gasPrice> _ </gasPrice>
        <origin> ORIGIN_ID </origin> // who fires tx

        <blockhashes> _ </blockhashes>
        <block>
          <previousHash> _ </previousHash>
          <ommersHash> _ </ommersHash>
          <coinbase> _ </coinbase>
          <stateRoot> _ </stateRoot>
          <transactionsRoot> _ </transactionsRoot>
          <receiptsRoot> _ </receiptsRoot>
          <logsBloom> _ </logsBloom>
          <difficulty> _ </difficulty>
          <number> _ </number>
          <gasLimit> _ </gasLimit>
          <gasUsed> _ </gasUsed>
          <timestamp> _ </timestamp>
          <extraData> _ </extraData>
          <mixHash> _ </mixHash>
          <blockNonce> _ </blockNonce>
          <baseFee> _ </baseFee>
          <ommerBlockHeaders> _ </ommerBlockHeaders>
        </block>
      </evm>

      <network>
        <chainID> _ </chainID>

        <activeAccounts> SetItem(ACCT_ID) _:Set </activeAccounts>

        <accounts>
          <account>
            <acctID> ACCT_ID </acctID>
            <balance> _ </balance>
            <code> #parseByteStack("0x6060604052600436106100 6d576000357c01000000000000000000000000000000000000000000000000000000000000
            <storage> #hashedLocation("Solidity", 1, OWNER) |-> BAL
_:Map </storage>
            <origStorage> _ </origStorage>
            <nonce> _ </nonce>
          </account>
         // ... // TODO: fix
        </accounts>

        <txOrder> _ </txOrder>
        <txPending> _ </txPending>
        <messages> _ </messages>
      </network>
    </ethereum>
    </kevm>
    requires 0 <=Int ACCT_ID    andBool ACCT_ID    <Int (2 ^Int 160)
     andBool 0 <=Int CALLER_ID  andBool CALLER_ID  <Int (2 ^Int 160)
     andBool 0 <=Int ORIGIN_ID  andBool ORIGIN_ID  <Int (2 ^Int 160)
     andBool 0 <=Int CALL_DEPTH andBool CALL_DEPTH <Int 1024
     andBool 0 <=Int OWNER      andBool OWNER      <Int (2 ^Int 160)
andBool 0 <=Int BAL        andBool BAL        <Int (2 ^Int 256)


endmodule
```

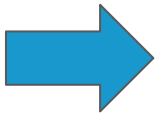
KEVM former approach might be too challenging

Property test function from Foundry

(Easy to write)

Symbolic execution of KEVM

(Formal verification)



kontrol

Aiming for Formal Verification

runtime verification

- Hoare Triples - `(∀ vars){pre-conditions} code {post-conditions}`
- We can use Foundry and KEVM to define Hoare Triples in Solidity parametric tests.

```
function testProperty(vars) external {
   assume pre;
   code;
   assert post;
}
```

- Using the symbolic execution capabilities of KEVM, it will formally verify the specifications.

runtime verification

kontrol Hands-on

Step-by-step tutorial

runtime verification

Github repository for all materials

https://github.com/runtimeverification/k-tutorial-atva-2023

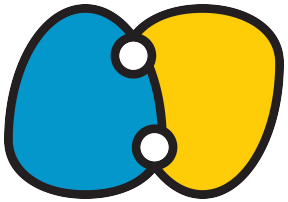
More on kontrol

runtime verification

Find out more at

https://docs.runtimeverification.com/kontrol/overview/readme

runtime verification



Questions?

https://runtimeverification.com/

@rv_inc

https://discord.com/invite/CurfmXNtbN

contact@runtimeverification.com