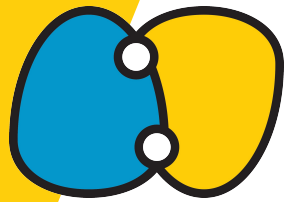
runtime verification



Getting started with Kontrol: A formal verification tool

Andrei Văcaru
Juan Conejero
Runtime Verification Inc.

About Us

runtime verification

Andrei Văcaru

Juan Conejero

Reproduction Steps

runtime verification

- Workshop repository:

  github.com/runtimeverification/kontrol-workshop-ethcc-7

- Docker image (recommended):

  ghcr.io/runtimeverification/kontrol/kontrol-workshop-ethcc-7

- Get the image:

```
docker pull ghcr.io/runtimeverification/kontrol/kontrol-workshop-ethcc-7
```

Workshop Overview

runtime verification

- Part 1: Kontrol Basics
  - Your first proof
  - Why Kontrol
  - Understanding Kontrol output
  - Debugging a proof

Workshop Overview

runtime verification

- Part 2: Kontrol on Roids
  - Compositional Symbolic Execution
  - External computation
  - NatSpec annotations
  - Proof and branch parallelization

Part 1:
Kontrol Basics.

Getting Kontrol

runtime verification

- Via `kup` (most user friendly):
  - `bash <(curl https://kframework.org/install)`
  - `kup install kontrol`
  - `kup` repo: https://github.com/runtimeverification/kup

Getting Kontrol

runtime verification

- Via `kup` (most user friendly):
  - `bash <(curl https://kframework.org/install)`
  - `kup install kontrol`
  - `kup` repo: https://github.com/runtimeverification/kup

```
➜  ~ kup list
```

Package name (alias)	Installed version	Status
kup	e78eda6a4b16183761e98fbc437e96c97c714709	🟢 installed
k.openssl.procps.secp256k1	a8dc6070e3d84d5ab009eb748b25a9aa2a51671f (v7.1.33)	🟠 newer version available
kontrol	*local checkout*	🟢 installed
kavm		🔵 available
kevm		🔵 available
kplutus		🔵 available
kmir		🔵 available
kmxwasm		🔵 available

Getting Kontrol

runtime verification

- Via `kup` (most user friendly):
  - `bash <(curl https://kframework.org/install)`
  - `kup install kontrol`
  - `kup` repo: https://github.com/runtimeverification/kup

- Documentation: https://docs.runtimeverification.com/kontrol/cheatsheets/kup-cheatsheet

Your First Kontrol Proof

runtime verification

test/Counter.t.sol

```
function testFuzz_SetNumber(uint256 x) public {
    counter.setNumber(x);
    assertEq(counter.number(), x);
}
```

forge build

```
[⠒] Compiling...
[⠢] Compiling 27 files with 0.8.25
[⠆] Solc 0.8.25 finished in 1.01s
Compiler run successful!
```

Your First Kontrol Proof

runtime verification

test/Counter.t.sol

```
function testFuzz_SetNumber(uint256 x) public {
    counter.setNumber(x);
    assertEq(counter.number(), x);
}
```

```
forge test --match-test testFuzz_SetNumber
```

```
[PASS] testFuzz_SetNumber(uint256) (runs: 256, μ: 30765, ~: 31310)
```

Your First Kontrol Proof

runtime verification

test/Counter.t.sol

```
function testFuzz_SetNumber(uint256 x) public {
    counter.setNumber(x);
    assertEq(counter.number(), x);
}
```

kontrol build

```
🔨 Building Kontrol project 🔨
 Add `--verbose` to `kontrol build` for more details!
✅ Success! Kontrol project built 💪
```

Your First Kontrol Proof

runtime verification

test/Counter.t.sol

```
function testFuzz_SetNumber(uint256 x) public {
    counter.setNumber(x);
    assertEq(counter.number(), x);
}
```

```
kontrol prove --match-test testFuzz_SetNumber
```

```
✨ PROOF PASSED ✨ test%CounterTest.testFuzz_SetNumber(uint256):0
```

What Did Just Happen?

runtime verification

- What
  - did
    - just
      - happen?
        - ??

- ???

Symbolic Execution!

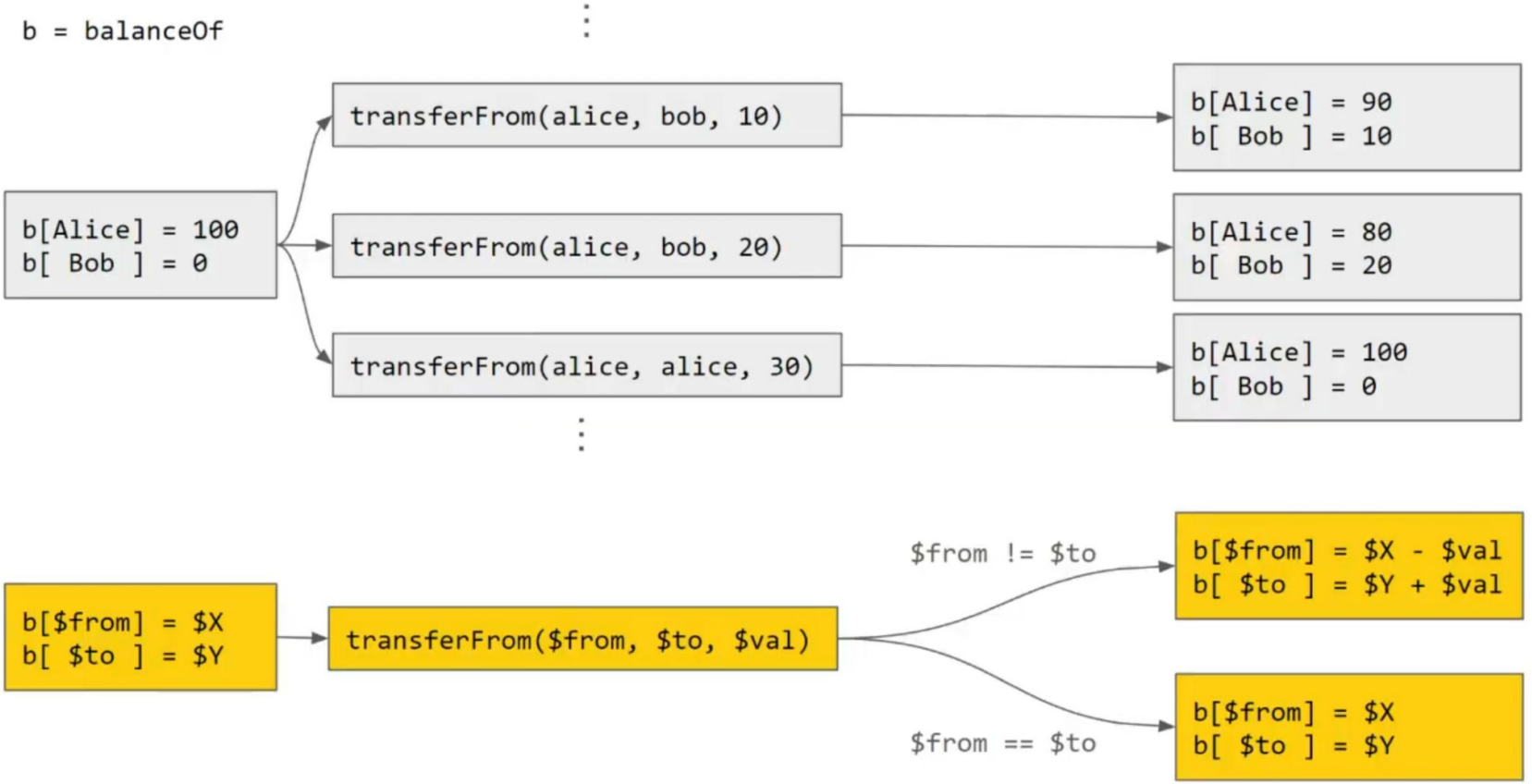
runtime verification

- What is symbolic execution?
  - Execution
  - Symbolic?
    - Not concrete :)

- Great. Let's elaborate:
  - `transferFrom(address from, address to, uint256 val)`
  - Concrete execution =
    - Execute with concrete `from`, `to` and `val`
    - Obtain concrete state update
  - Symbolic execution =
    - Execute with symbolic `from`, `to` or `val`
    - Obtain symbolic state update

Symbolic Execution!

runtime verification



A Word on Formal Verification

runtime verification

- What is Formal Verification?
  - Assert properties of your code (i.e. **specify** its behavior)
  - Prove that your code satisfies the specification
- There are a lot of different techniques to achieve this
- Symbolic execution is one of them
  - Execute your code with symbolic parameters
  - Get all possible execution traces
  - Check that each execution trace satisfies the specification

What is Kontrol

runtime verification

- Symbolic execution engine
- Specs as Foundry property tests
- Verifies EVM bytecode
- Open Source (PRs welcome!)
- github.com/runtimeverification/kontrol

Why So Formal?

runtime verification

When Fuzzing Fails

runtime verification

```
function setNumber(uint256 newNumber) public {
    number = newNumber;
}
```

When Fuzzing Fails

runtime verification

```
function setNumberCaffeinated(uint256 newNumber,
                              bool isTime) public {
    if (newNumber == 0xC0FFEE && isTime) {
        revert CoffeeBreak();
    }
    number = newNumber;
}
```

```
forge test --match-test testFuzz_SetNumberCaffeinated
```

```
[PASS] testFuzz_SetNumberCaffeinated(uint256,bool) (runs: 256, μ: 31189, ~: 31500)
```

When Fuzzing Fails

runtime verification

```
function setNumberCaffeinated(uint256 newNumber,
                              bool isTime) public {
    if (newNumber == 0xC0FFEE && isTime) {
        revert CoffeeBreak();
    }
    number = newNumber;
}
```

```
kontrol prove --match-test testFuzz_SetNumberCaffeinated
```

❌ **PROOF FAILED** ❌ test%CounterTest.testFuzz_SetNumberCaffeinated(uint256,bool):0

Visual Control Flow

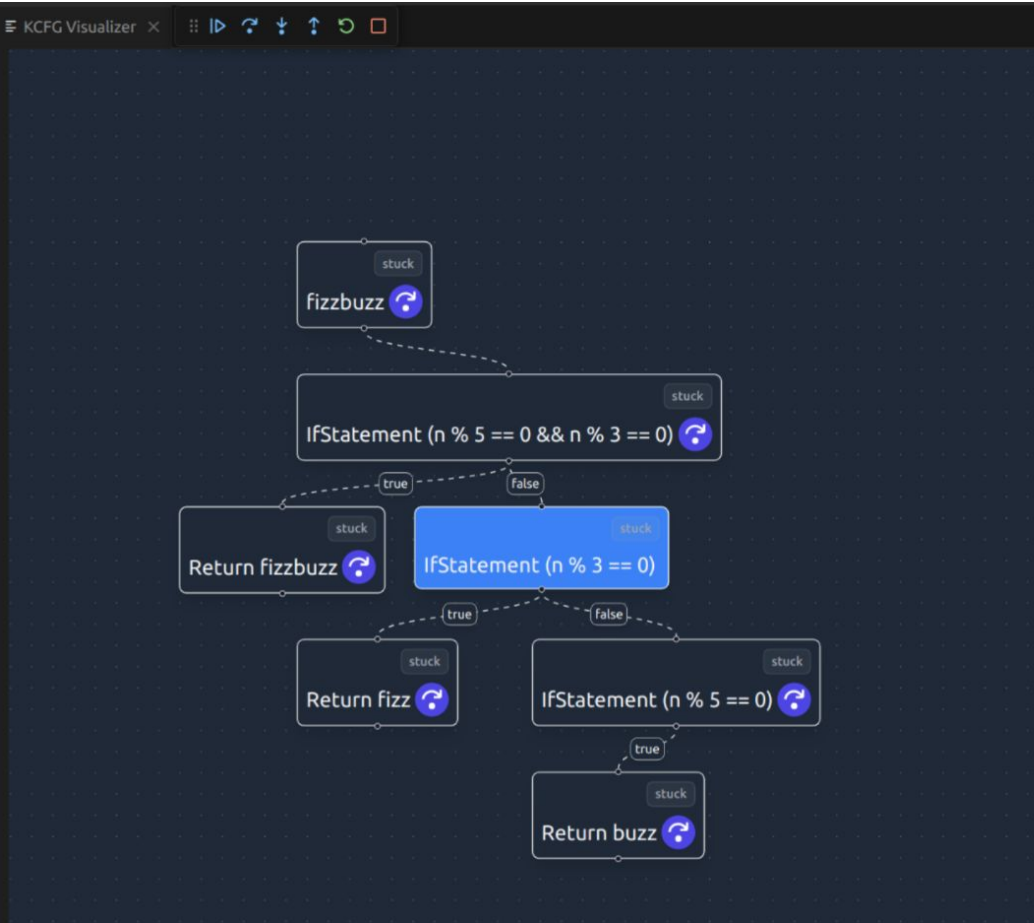
runtime verification

Demo.sol 9+

src > Demo.sol

```
7   uint256 public number;
8
    ▷ Debug
9   function setNumber(uint256 newNumber) public {
10      number = newNumber;
11  }
12
    ▷ Debug
13  function increment() public {
14      number++;
15  }
16
    ▷ Debug
17  function fizzbuzz(uint256 n) public returns (string memory) {
18      if (n % 5 == 0 && n % 3 == 0) {
19          return "fizzbuzz";
20      } else if (n % 3 == 0) {
21          return "fizz";
22      } else if (n % 5 == 0) {
23          return "buzz";
24      }
25  }
26
    ▷ Debug
27  function sequenceOfAssignments() public {
28      uint256 a = 1;
29      uint256 b = 2;
30      uint256 c = 3;
31      uint256 d = 4;
32      uint256 e = 5;
33      uint256 f = 6;
34  }
35
36
    ▷ Debug
37  function sumToN(uint256 n) public {
```

KCFG Visualizer



Proof Debugging - Feat. Solady

runtime verification

Let's verify `mulWad`

```
/// @dev Equivalent to `(x * y) / WAD` rounded down.
function mulWad(uint256 x, uint256 y) internal pure returns (uint256 z) {
    assembly {
        if mul(y, gt(x, div(not(0), y))) {
            mstore(0x00, 0xbac65e5b)
            revert(0x1c, 0x04)
        }
        z := div(mul(x, y), WAD)
    }
}
```

Proof Debugging - Feat. Solady

runtime verification

`mulWad` specs:

```
function testMulWad(uint256 x, uint256 y) public {
    // No overflow case
    if (y == 0 || x <= type(uint256).max / y) {
        uint256 zSpec = (x * y) / WAD;
        uint256 zImpl = mulWad(x, y);
        // mulWad behaves as specified
        assert(zImpl == zSpec);
    } else {
        // If overflow, it should revert
        vm.expectRevert(MulWadFailed.selector);
        // External call needed for expectRevert to kick in
        this.mulWad(x, y);
    }
}
```

Proof Debugging - Feat. Solady

runtime verification

`mulWad` specs:

```
function testMulWad(uint256 x, uint256 y) public {
    // No overflow case
    if (y == 0 || x <= type(uint256).max / y) {
        uint256 zSpec = (x * y) / WAD;
        uint256 zImpl = mulWad(x, y);
        // mulWad behaves as specified
        assert(zImpl == zSpec);
    } else {
        // If overflow, it should revert
        vm.expectRevert(MulWadFailed.selector);
        // External call needed for expectRevert to kick in
        this.mulWad(x, y);
    }
}
```

Proof Debugging - Feat. Solady

runtime verification

`mulWad` specs:

```
function testMulWad(uint256 x, uint256 y) public {
    // No overflow case
    if (y == 0 || x <= type(uint256).max / y) {
        uint256 zSpec = (x * y) / WAD;
        uint256 zImpl = mulWad(x, y);
        // mulWad behaves as specified
        assert(zImpl == zSpec);
    } else {
        // If overflow, it should revert
        vm.expectRevert(MulWadFailed.selector);
        // External call needed for expectRevert to kick in
        this.mulWad(x, y);
    }
}
```

Proof Debugging - Feat. Solady

runtime verification

- Let’s run the proof!
  - `kontrol build`
  - `kontrol prove --mt testMulWad`

…

❌ **PROOF FAILED** ❌ test%MulWadTest.testMulWad(uint256,uint256):0

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - #Not ({ VV1_y_114b9705:Int #Equals 0 })
  - #And { true #Equals VV0_x_114b9705:Int <=Int maxUInt256 /Int VV1_y_114b9705:Int }
  - #And #Not ({ chop (VV1_y_114b9705:Int *Int bool2Word (maxUInt256 /Int VV1_y_114b9705:Int <Int VV0_x_114b9705:Int)) #Equals 0 })

Proof Debugging - Feat. Solady

runtime verification

- Translating Kontrol fanciness:
  - VV0_x_114b9705 → x
  - VV1_y_114b9705 → y

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - #Not ({ VV1_y_114b9705:Int #Equals 0 })
  - #And { true #Equals VV0_x_114b9705:Int <=Int maxUInt256 /Int VV1_y_114b9705:Int }
  - #And #Not ({ chop (VV1_y_114b9705:Int *Int bool2Word (maxUInt256 /Int VV1_y_114b9705:Int <Int VV0_x_114b9705:Int)) #Equals 0 })

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - #Not ({ y:Int #Equals 0 })
  - #And { true #Equals x:Int <=Int maxUInt256 /Int y:Int }
  - #And #Not ({ chop (y:Int *Int bool2Word (maxUInt256 /Int y:Int <Int x:Int)) #Equals 0 })

Proof Debugging - Feat. Solady

runtime verification

- Translating Kontrol fanciness:
  - VV0_x_114b9705 → x
  - VV1_y_114b9705 → y
  - Remove #And, { true #Equals _ }

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - #Not ({ y:Int #Equals 0 })
  - #And { true #Equals x:Int <=Int maxUInt256 /Int y:Int }
  - #And #Not ({ chop (y:Int *Int bool2Word (maxUInt256 /Int y:Int <Int x:Int)) #Equals 0 })

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - #Not ({ y:Int #Equals 0 })
  - x:Int <=Int maxUInt256 /Int y:Int
  - #Not ({ chop (y:Int *Int bool2Word (maxUInt256 /Int y:Int <Int x:Int)) #Equals 0 })

Proof Debugging - Feat. Solady

runtime verification

- Translating Kontrol fanciness:
  - VV0_x_114b9705 → x
  - VV1_y_114b9705 → y
  - Remove #And, { true #Equals _ }
  - Substitute #Not ({ _ #Equals 0) } with _ != 0

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - #Not ({ y:Int #Equals 0 })
  - x:Int <=Int maxUInt256 /Int y:Int
  - #Not ({ chop (y:Int *Int bool2Word (maxUInt256 /Int y:Int <Int x:Int)) #Equals 0 })

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - y:Int != 0
  - x:Int <=Int maxUInt256 /Int y:Int
  - chop (y:Int *Int bool2Word (maxUInt256 /Int y:Int <Int x:Int)) != 0

Proof Debugging - Feat. Solady

runtime verification

- Translating Kontrol fanciness:
  - VV0_x_114b9705 → x
  - VV1_y_114b9705 → y
  - Remove #And, { true #Equals _ }
  - Substitute #Not ({ _ #Equals 0) } with _ != 0
  - Remove :Int typing

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - y:Int != 0
  - x:Int <=Int maxUInt256 /Int y:Int
  - chop (y:Int *Int bool2Word (maxUInt256 /Int y:Int <Int x:Int)) != 0

Proof Debugging - Feat. Solady

runtime verification

- Let’s read the path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0

Proof Debugging - Feat. Solady

runtime verification

- Don’t be intimidated!!
  - #Not ({ VV1_y_114b9705:Int #Equals 0 })
  - #And { true #Equals VV0_x_114b9705:Int <=Int maxUInt256 /Int VV1_y_114b9705:Int }
  - #And #Not ({ chop (VV1_y_114b9705:Int *Int bool2Word (maxUInt256 /Int VV1_y_114b9705:Int <Int VV0_x_114b9705:Int)) #Equals 0 })

Proof Debugging - Feat. Solady

runtime verification

- Nice path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0

Proof Debugging - Feat. Solady

runtime verification

- KEVM definitions:
  - chop (X) = X % maxUInt256
  - bool2Word → Casting booleans to integers

Proof Debugging - Feat. Solady

runtime verification

- Legible path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0

Proof Debugging - Feat. Solady

runtime verification

- Legible path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0
  - y * bool2Word (maxUInt256 / y < x) != 0

Proof Debugging - Feat. Solady

runtime verification

- Legible path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0
  - bool2Word (maxUInt256 / y < x) != 0

Proof Debugging - Feat. Solady

runtime verification

- Legible path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0
  - maxUInt256 / y < x != false

Proof Debugging - Feat. Solady

runtime verification

- Legible path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0
  - maxUInt256 / y < x == true

Proof Debugging - Feat. Solady

runtime verification

- Legible path condition:
  - y != 0
  - x <= maxUInt256 / y
  - chop (y * bool2Word (maxUInt256 / y < x)) != 0
  - maxUInt256 / y < x == true

Proof Debugging - Feat. Solady

runtime verification

Let's look at the definition of bool2Word

- Go to github.com/runtimeverification/evm-semantics/
- `cd kevm-pyk/src/kevm_pyk/kproj/evm-semantics` (where KEVM lives)
- `grep bool2Word`

```
evm-types.md:    rule bool2Word( true  ) => 1
evm-types.md:    rule bool2Word( false ) => 0
lemmas/lemmas.k:    rule bool2Word(A) |Int bool2Word(B) => bool2Word(A  orBool B) [simplification]
lemmas/lemmas.k:    rule bool2Word(A) &Int bool2Word(B) => bool2Word(A andBool B) [simplification]
lemmas/lemmas.k:    rule bool2Word(_B) |Int 1 => 1           [simplification]
lemmas/lemmas.k:    rule bool2Word( B) &Int 1 => bool2Word(B) [simplification]
lemmas/lemmas.k:    rule bool2Word(X ==Int 1) => X requires #rangeBool(X) [simplification]
lemmas/lemmas.k:    rule bool2Word( B:Bool ) ==Int I => B ==K word2Bool(I) [simplification, concrete(I)]
```

Proof Debugging - Feat. Solady

runtime verification

No rule explicitly simplifies argument of bool2Word

So… The next step is to add one!

But… How does a rule that triggers argument simplification look like?

Well, very simple!

Proof Debugging - Feat. Solady

runtime verification

```
rule bool2Word ( X ) => 1 requires X              [simplification]

rule bool2Word ( X ) => 0 requires notBool X [simplification]
```

Proof Debugging - Feat. Solady

runtime verification

We save the rule in a `.k` file, say `lemmas.k`

And tell Kontrol to take them into account!

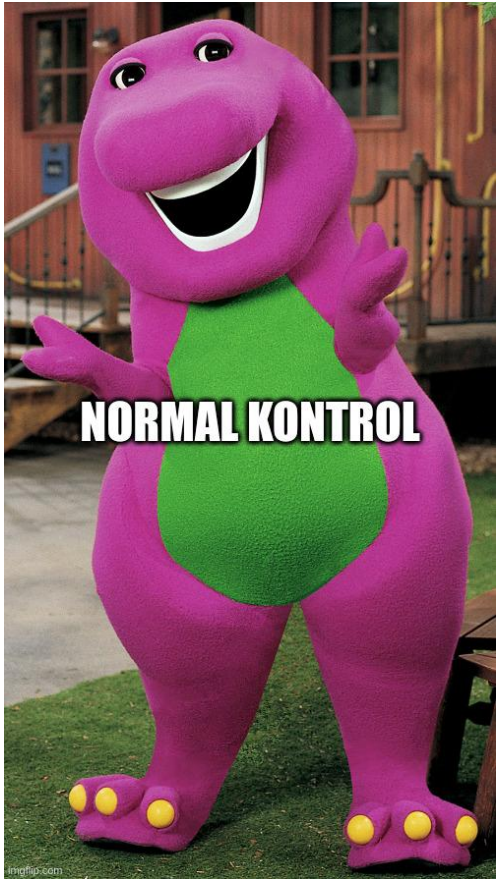
```
kontrol build --requires lemmas.k --module MulWadTest:KONTROL-LEMMAS
```

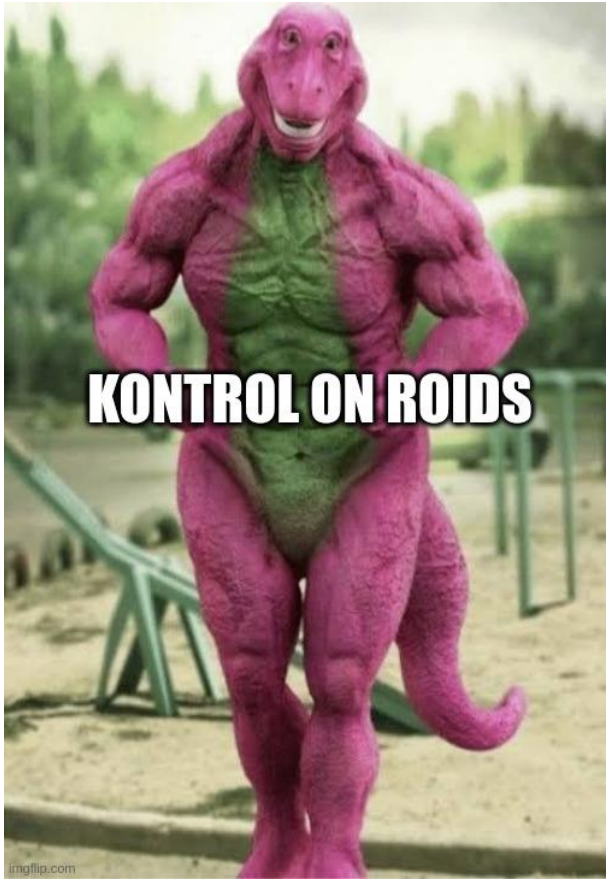
runtime verification

Part 2:
Kontrol on Roids.

Kontrol on Roids

runtime verification





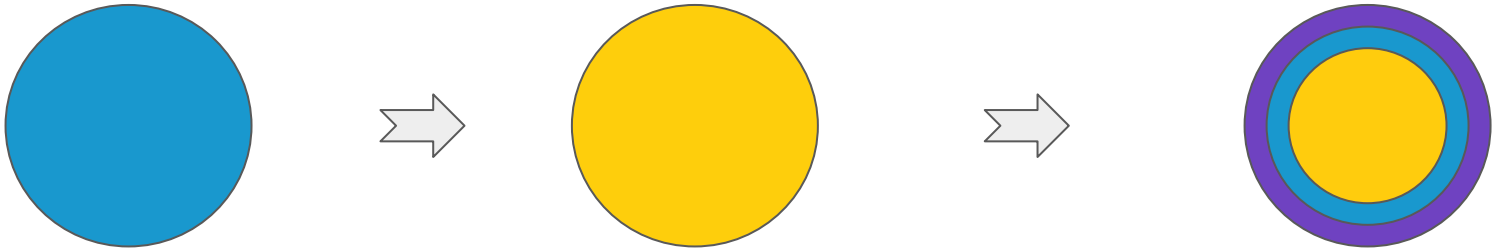
Kontrol on Roids

runtime verification

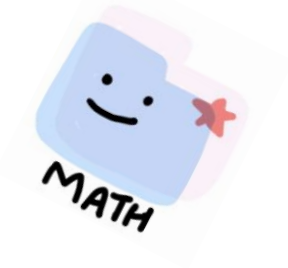
- Compositional Symbolic Execution
- External computation
- NatSpec annotations
- Proof and branch parallelization

Compositional Symbolic Execution (CSE)

runtime verification



CSE - Function Summary



runtime verification

`GLDToken.transfer(address to, uint256 amount)`

`msg.sender == 0`			`ERC20InvalidSender`
`msg.sender =/= 0`	`to == 0`		`ERC20InvalidReceiver`
`msg.sender =/= 0`	`to =/= 0`	`_balances[msg.sender] < amount`	`ERC20InsufficientBalance`

`msg.sender =/= 0`	`to =/= 0`	`_balances[msg.sender] >= amount`	`STORAGE[keccak(msg.sender, 0)] -= amount`
			`STORAGE[keccak(to, 0)] += amount`
			`emit Transfer event`
			`output TRUE`

CSE - Function Summary

runtime verification

```
function testTransferConsecutive(address alice, address bob, uint256 amount, bytes32
storageSlot) public unchangedStorage(storageSlot)
{
    vm.assume(alice != address(0));
    vm.assume(bob != address(0));
    _notBuiltinOrPrecompiledAddress(alice);
    _notBuiltinOrPrecompiledAddress(bob);
    vm.assume(gld.balanceOf(alice) >= amount);
    vm.assume(gld.balanceOf(bob) <= MAX_INT - amount);
    vm.prank(alice);
    gld.transfer(bob, amount);
    vm.prank(bob);
    gld.transfer(alice, amount);
    }
}
```

CSE - Function Summary

runtime verification

```
function testTransferConsecutive(address alice, address bob, uint256 amount, bytes32
storageSlot) public unchangedStorage(storageSlot)
{
    vm.assume(alice != address(0));
    vm.assume(bob != address(0));
    _notBuiltinOrPrecompiledAddress(alice);
    _notBuiltinOrPrecompiledAddress(bob);
    vm.assume(gld.balanceOf(alice) >= amount);
    vm.assume(gld.balanceOf(bob) <= MAX_INT - amount);
    vm.prank(alice);
    gld.transfer(bob, amount);
    vm.prank(bob);
    gld.transfer(alice, amount);
    }
}
```

CSE - Function Summary

runtime verification

```
function testTransferConsecutive(address alice, address bob, uint256 amount, bytes32 storageSlot) public unchangedStorage(storageSlot)
{
    vm.assume(alice != address(0));
    vm.assume(bob != address(0));
    _notBuiltinOrPrecompiledAddress(alice);
    _notBuiltinOrPrecompiledAddress(bob);
    vm.assume(gld.balanceOf(alice) >= amount);
    vm.assume(gld.balanceOf(bob) <= MAX_INT - amount);
    vm.prank(alice);
    gld.transfer(bob, amount);
    vm.prank(bob);
    gld.transfer(alice, amount);
    }
}
```

CSE - Function Summary

runtime verification

```
function testTransferConsecutive(address alice, address bob, uint256 amount, bytes32
storageSlot) public unchangedStorage(storageSlot)
{
    vm.assume(alice != address(0));
    vm.assume(bob != address(0));
    _notBuiltinOrPrecompiledAddress(alice);
    _notBuiltinOrPrecompiledAddress(bob);
    vm.assume(gld.balanceOf(alice) >= amount);
    vm.assume(gld.balanceOf(bob) <= MAX_INT - amount);
    vm.prank(alice);
    gld.transfer(bob, amount);
    vm.prank(bob);
    gld.transfer(alice, amount);
    }
}
```

CSE - How It’s Done

runtime verification

Given a test proof:

1. Fetch all the external functions (dependencies) that are called in the test

2. Execute them in an abstract state of EVM

3. Generate a summary for each dependency

4. Whenever a dependency is called, apply the summary instead of executing the function once more.

CSE - Key Features

runtime verification

- Modularity
  - Subproofs that can be reasoned about in an abstract context
- Efficiency
  - Avoiding repeated analysis of the same dependency in different parts of the program.
- Scalability
  - New proofs can be executed without reasoning about known dependencies

CSE - Function Summary

runtime verification

```
function testTransferConsecutive(address alice, address bob, uint256 amount, bytes32
storageSlot) public unchangedStorage(storageSlot)
{
    vm.assume(alice != address(0));
    vm.assume(bob != address(0));
    _notBuiltinOrPrecompiledAddress(alice);
    _notBuiltinOrPrecompiledAddress(bob);
    vm.assume(gld.balanceOf(alice) >= amount);
    vm.assume(gld.balanceOf(bob) <= MAX_INT - amount);
    vm.prank(alice);
    gld.transfer(bob, amount);
    vm.prank(bob);
    gld.transfer(alice, amount);
    }
}
```

CSE - Function Summary

runtime verification

```
APRProof: test%GLDTokenTest.testTransferConsecutive(address,address,uint256,bytes32):0
    status: ProofStatus.PASSED
    admitted: False
    nodes: 7
    pending: 0
    failing: 0
    vacuous: 0
    stuck: 0
    terminal: 3
    refuted: 0
    bounded: 0
    execution time: 3m 36s
Subproofs: 2
                APRProof: test%GLDTokenTest.testTransferConsecutive(address,address,uint256,bytes32):1
                    status: ProofStatus.PASSED
                    admitted: False
                    nodes: 14
                    pending: 0
                    failing: 0
                    vacuous: 0
                    stuck: 0
                    terminal: 4
                    refuted: 0
                    bounded: 0
                    execution time: 4m 52s
                Subproofs: 0
```

External Computation

runtime verification

- What is external computation?
  - Compute with Foundry, import results to Kontrol
  - Only works for concrete execution
- Why do we need it?
  - Kontrol executes in math, which is not the fastest way
  - Test suite initialization can be *very* long and complex

External Computation - How does it work?

runtime verification

Foundry-Kontrol approach: outsource concrete execution to Foundry

- Execute any function with Foundry
- Record all state updates with `vm.dumpState` or `vm.stopAndReturnStateDiff`
- Recreate the recorded state with `vm.store` and `vm.etch`

[Foundry]
Execute and produce
produce JSON state update

[Kontrol]
From JSON state update,
create a summary contract
containing only `etch` and
`store` cheatcodes

External Computation - How does it work?

runtime verification

We go from:

- Long execution script
- Lots of dependencies
- Nontrivial deployment logic

External Computation - How does it work?

runtime verification

We go from:

- Long execution script
- Lots of dependencies
- Nontrivial deployment logic

To:

- Comparatively short summary contract
- Effectively no project dependencies
- Trivial recreation logic

```
slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
value = hex'0000000000000000000000000000000000000000000000000000000000000001';
vm.store(ProtocolVersionsProxyAddress, slot, value);
vm.etch(OptimismPortalProxyAddress, OptimismPortalProxyCode);
slot = hex'b53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103';
value = hex'000000000000000000000000db8cff278adccf9e9b5da745b44e754fc4ee3c76';
vm.store(OptimismPortalProxyAddress, slot, value);
vm.etch(L2OutputOracleProxyAddress, L2OutputOracleProxyCode);
slot = hex'b53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103';
value = hex'000000000000000000000000db8cff278adccf9e9b5da745b44e754fc4ee3c76';
vm.store(L2OutputOracleProxyAddress, slot, value);
vm.etch(SystemConfigProxyAddress, SystemConfigProxyCode);
slot = hex'b53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103';
value = hex'000000000000000000000000db8cff278adccf9e9b5da745b44e754fc4ee3c76';
vm.store(SystemConfigProxyAddress, slot, value);
```

External Computation

runtime verification

- Nice simplification!
  - Now Kontrol only has to execute a bunch of cheatcodes!
- Is this all performance improvement we can get?
  - Certainly **not** :)
  - Kontrol can directly include the state updates into a proof
  - No cheatcode recreation needed!

External Computation - Example

runtime verification

```
function severalCountersDump() public {
    for (uint256 i; i <= 9; ++i) {
        counter = new Counter();
        counter.setNumber(i);
        vm.deal(address(counter), i * (1 ether));
        string memory addressName = string.concat("counter", vm.toString(i));
        save_address(address(counter), addressName);
    }
    vm.dumpState(check_file(folder, dumpStateFile));
}
```

External Computation - Example

runtime verification

```
function severalCountersDump() public {
    for (uint256 i; i <= 9; ++i) {
        counter = new Counter();
        counter.setNumber(i);
        vm.deal(address(counter), i * (1 ether));
        string memory addressName = string.concat("counter", vm.toString(i));
        save_address(address(counter), addressName);
    }
    vm.dumpState(check_file(folder, dumpStateFile));
}
```

External Computation - Example

runtime verification

```
function severalCountersDump() public {
    for (uint256 i; i <= 9; ++i) {
        counter = new Counter();
        counter.setNumber(i);
        vm.deal(address(counter), i * (1 ether));
        string memory addressName = string.concat("counter", vm.toString(i));
        save_address(address(counter), addressName);
    }
    vm.dumpState(check_file(folder, dumpStateFile));
}
```

External Computation - Example

runtime verification

```
function severalCountersDump() public {
    for (uint256 i; i <= 9; ++i) {
        counter = new Counter();
        counter.setNumber(i);
        vm.deal(address(counter), i * (1 ether));
        string memory addressName = string.concat("counter", vm.toString(i));
        save_address(address(counter), addressName);
    }
    vm.dumpState(check_file(folder, dumpStateFile));
}
```

External Computation - Example

runtime verification

```
forge script src/ExternalComputation.sol:ExternalComputation
--sig severalCountersDump --ffi
```

State recording JSON:


```
  "0x2e234dae75c793f67a35089c9d99245e1c58470b": {
    "nonce": "0x1",
    "balance": "0xde0b6b3a7640000",
    "code": "0x6080604052348015600f57600080fd5b5060043610603c5760003560e01c80633fb5c1cb1460415\
780638381f58a146053578063d09de08a14606d575b600080fd5b6051604c3660046083565b600055565b005b605b6\
0005481565b60405190815260200160405180910390f35b6051600080549080607c83609b565b9190505550565b600\
0602082840312156094576000080fd5b5035919050565b60006001820160ba57634e487b7160e01b600052601160045\
260246000fd5b506001019056fea2646970667358221220037d3e0197c9d08f161dbb1697fcd490e178cceed9f68884\
6b9eca0fb960fecdc64736f6c63430008190033",
    "storage": {
      "0x0000000000000000000000000000000000000000000000000000000000000000": "0x000000000000000\
0000000000000000000000000000000000000000000000001"
    }
  },
```

External Computation - Example

runtime verification

```
forge script src/ExternalComputation.sol:ExternalComputation
--sig severalCountersDump --ffi
```

Address naming JSON:


```
{
  "0x03A6a84cD762D9707A21605b548aaaB891562aAb": "counter8",
  "0x1d1499e622D69689cdf9004d05Ec547d650Ff211": "counter6",
  "0x2e234DAe75C793f67A35089C9d99245E1C58470b": "counter1",
  "0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f": "counter0",
  "0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9": "counter3",
  "0xA4AD4f68d0b91CFD19687c881e50f3A00242828c": "counter7",
  "0xD6BbDE9174b1CdAa358d2Cf4D57D1a9F7178FBfF": "counter9",
  "0xF62849F9A0B5Bf2913b396098F7c7019b51A820a": "counter2",
  "0xa0Cb889707d426A7A386870A03bc70d1b0697598": "counter5",
  "0xc7183455a4C133Ae270771860664b6B7ec320bB1": "counter4"
}
```

External Computation - Example

runtime verification

```
kontrol load-state RecordedState state-dump/StateDump.json     \
       --contract-names state-dump/AddressNames.json           \
       --output-dir test
```

```
contract RecordedStateCode {
  bytes constant internal counter1Code = hex'6080604052348015600f57600080fd5b5\
060043610603c5760003560e01c80633fb5c1cb1460415780638381f58a146053578063d09de08\
a14606d575b600080fd5b6051604c3660046083565b600055565b005b605b60005481565b60405\
190815260200160405180910390f35b60516000805490806007c83609b565b9190505550565b600\
060208284031215609457600080fd5b5035919050565b60006001820160ba57634e487b7160e01\
b600052601160045260246000fd5b506001019056fea2646970667358221220037d3e0197c9d08f\
161dbb1697fcd490e178cceed9f688846b9eca0fb960fecdc64736f6c63430008190033';
  bytes constant internal counter0Code = hex'6080604052348015600f57600080fd5b5\
060043610603c5760003560e01c80633fb5c1cb1460415780638381f58a146053578063d09de08\
a14606d575b600080fd5b6051604c3660046083565b600055565b005b605b60005481565b60405\
190815260200160405180910390f35b60516000805490806007c83609b565b9190505550565b600\
060208284031215609457600080fd5b5035919050565b60006001820160ba57634e487b7160e01\
b600052601160045260246000fd5b506001019056fea2646970667358221220037d3e0197c9d08f\
161dbb1697fcd490e178cceed9f688846b9eca0fb960fecdc64736f6c63430008190033';
```

External Computation - Example

runtime verification

```
kontrol load-state RecordedState state-dump/StateDump.json     \
       --contract-names state-dump/AddressNames.json           \
       --output-dir test
```

```
contract RecordedStateCode {
  bytes constant internal counter1Code = hex'6080604052348015600f57600080fd5b5\
060043610603c5760003560e01c80633fb5c1cb1460415780638381f58a146053578063d09de08\
a14606d575b600080fd5b6051604c3660046083565b600055565b005b605b60005481565b60405\
190815260200160405180910390f35b60516000805490806007c83609b565b9190505550565b600\
060208284031215609457600080fd5b5035919050565b60006001820160ba57634e487b7160e01\
b600052601160045260246000fd5b506001019056fea2646970667358221220037d3e0197c9d08f\
161dbb1697fcd490e178cceed9f688846b9eca0fb960fecdc64736f6c63430008190033';
  bytes constant internal counter0Code = hex'6080604052348015600f57600080fd5b5\
060043610603c5760003560e01c80633fb5c1cb1460415780638381f58a146053578063d09de08\
a14606d575b600080fd5b6051604c3660046083565b600055565b005b605b60005481565b60405\
190815260200160405180910390f35b60516000805490806007c83609b565b9190505550565b600\
060208284031215609457600080fd5b5035919050565b60006001820160ba57634e487b7160e01\
b600052601160045260246000fd5b506001019056fea2646970667358221220037d3e0197c9d08f\
161dbb1697fcd490e178cceed9f688846b9eca0fb960fecdc64736f6c63430008190033';
```

External Computation - Example

runtime verification

```
contract RecordedState is RecordedStateCode {
  address internal constant counter8Address = 0x03A6a84cD762D9707A21605b548aaaB891562aAb;
  address internal constant counter6Address = 0x1d1499e622D69689cdf9004d05Ec547d650Ff211;
  address internal constant counter1Address = 0x2e234DAe75C793f67A35089C9d99245E1C58470b;
  address internal constant counter0Address = 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f;
  address internal constant counter3Address = 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9;
  address internal constant counter7Address = 0xA4AD4f68d0b91CFD19687c881e50f3A00242828c;
  address internal constant counter9Address = 0xD6BbDE9174b1CdAa358d2Cf4D57D1a9F7178FBfF;
  address internal constant counter2Address = 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a;
  address internal constant counter5Address = 0xa0Cb889707d426A7A386870A03bc70d1b0697598;
  address internal constant counter4Address = 0xc7183455a4C133Ae270771860664b6B7ec320bB1;


  function recreateState() public {
    bytes32 slot;
    bytes32 value;
    vm.etch(counter8Address, counter8Code);
    vm.deal(counter8Address, 8000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000008';
    vm.store(counter8Address, slot, value);
    vm.etch(counter5Address, counter5Code);
    vm.deal(counter5Address, 5000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000005';
    vm.store(counter5Address, slot, value);
```

External Computation - Example

runtime verification

```
contract RecordedState is RecordedStateCode {
  address internal constant counter8Address = 0x03A6a84cD762D9707A21605b548aaaB891562aAb;
  address internal constant counter6Address = 0x1d1499e622D69689cdf9004d05Ec547d650Ff211;
  address internal constant counter1Address = 0x2e234DAe75C793f67A35089C9d99245E1C58470b;
  address internal constant counter0Address = 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f;
  address internal constant counter3Address = 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9;
  address internal constant counter7Address = 0xA4AD4f68d0b91CFD19687c881e50f3A00242828c;
  address internal constant counter9Address = 0xD6BbDE9174b1CdAa358d2Cf4D57D1a9F7178FBfF;
  address internal constant counter2Address = 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a;
  address internal constant counter5Address = 0xa0Cb889707d426A7A386870A03bc70d1b0697598;
  address internal constant counter4Address = 0xc7183455a4C133Ae270771860664b6B7ec320bB1;
```

```
  function recreateState() public {
    bytes32 slot;
    bytes32 value;
    vm.etch(counter8Address, counter8Code);
    vm.deal(counter8Address, 8000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000008';
    vm.store(counter8Address, slot, value);
    vm.etch(counter5Address, counter5Code);
    vm.deal(counter5Address, 5000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000005';
    vm.store(counter5Address, slot, value);
```

External Computation - Example

runtime verification

```
contract RecordedState is RecordedStateCode {
  address internal constant counter8Address = 0x03A6a84cD762D9707A21605b548aaaB891562aAb;
  address internal constant counter6Address = 0x1d1499e622D69689cdf9004d05Ec547d650Ff211;
  address internal constant counter1Address = 0x2e234DAe75C793f67A35089C9d99245E1C58470b;
  address internal constant counter0Address = 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f;
  address internal constant counter3Address = 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9;
  address internal constant counter7Address = 0xA4AD4f68d0b91CFD19687c881e50f3A00242828c;
  address internal constant counter9Address = 0xD6BbDE9174b1CdAa358d2Cf4D57D1a9F7178FBfF;
  address internal constant counter2Address = 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a;
  address internal constant counter5Address = 0xa0Cb889707d426A7A386870A03bc70d1b0697598;
  address internal constant counter4Address = 0xc7183455a4C133Ae270771860664b6B7ec320bB1;
```

```
  function recreateState() public {
    bytes32 slot;
    bytes32 value;
    vm.etch(counter8Address, counter8Code);
    vm.deal(counter8Address, 8000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000008';
    vm.store(counter8Address, slot, value);
    vm.etch(counter5Address, counter5Code);
    vm.deal(counter5Address, 5000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000005';
    vm.store(counter5Address, slot, value);
```

External Computation - Example

runtime verification

```
contract RecordedState is RecordedStateCode {
  address internal constant counter8Address = 0x03A6a84cD762D9707A21605b548aaaB891562aAb;
  address internal constant counter6Address = 0x1d1499e622D69689cdf9004d05Ec547d650Ff211;
  address internal constant counter1Address = 0x2e234DAe75C793f67A35089C9d99245E1C58470b;
  address internal constant counter0Address = 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f;
  address internal constant counter3Address = 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9;
  address internal constant counter7Address = 0xA4AD4f68d0b91CFD19687c881e50f3A00242828c;
  address internal constant counter9Address = 0xD6BbDE9174b1CdAa358d2Cf4D57D1a9F7178FBfF;
  address internal constant counter2Address = 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a;
  address internal constant counter5Address = 0xa0Cb889707d426A7A386870A03bc70d1b0697598;
  address internal constant counter4Address = 0xc7183455a4C133Ae270771860664b6B7ec320bB1;


  function recreateState() public {
    bytes32 slot;
    bytes32 value;
    vm.etch(counter8Address, counter8Code);
    vm.deal(counter8Address, 8000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000008';
    vm.store(counter8Address, slot, value);
    vm.etch(counter5Address, counter5Code);
    vm.deal(counter5Address, 5000000000000000000);
    slot = hex'0000000000000000000000000000000000000000000000000000000000000000';
    value = hex'0000000000000000000000000000000000000000000000000000000000000005';
    vm.store(counter5Address, slot, value);
```

External Computation - Example

runtime verification

Why all of this??

External Computation - Example

runtime verification

```
contract ExternalComputationTest is RecordedState {
    function prove_multiple_counters() public view {
        Counter[] memory counters = new Counter[](10);

        // Load all counters
        counters[0] = (Counter(address(counter0Address)));
        counters[1] = (Counter(address(counter1Address)));
        counters[2] = (Counter(address(counter2Address)));
        counters[3] = (Counter(address(counter3Address)));
        counters[4] = (Counter(address(counter4Address)));
        counters[5] = (Counter(address(counter5Address)));
        counters[6] = (Counter(address(counter6Address)));
        counters[7] = (Counter(address(counter7Address)));
        counters[8] = (Counter(address(counter8Address)));
        counters[9] = (Counter(address(counter9Address)));

        for (uint256 i; i <= 9; ++i) {
            require(counters[i].number() == i, "Bad number");
            require(address(counters[i]).balance == i * (1 ether), "Ill funded");
        }
    }
}
```

External Computation - Example

runtime verification

```
contract ExternalComputationTest is RecordedState {
    function prove_multiple_counters() public view {
        Counter[] memory counters = new Counter[](10);

        // Load all counters
        counters[0] = (Counter(address(counter0Address)));
        counters[1] = (Counter(address(counter1Address)));
        counters[2] = (Counter(address(counter2Address)));
        counters[3] = (Counter(address(counter3Address)));
        counters[4] = (Counter(address(counter4Address)));
        counters[5] = (Counter(address(counter5Address)));
        counters[6] = (Counter(address(counter6Address)));
        counters[7] = (Counter(address(counter7Address)));
        counters[8] = (Counter(address(counter8Address)));
        counters[9] = (Counter(address(counter9Address)));

        for (uint256 i; i <= 9; ++i) {
            require(counters[i].number() == i, "Bad number");
            require(address(counters[i]).balance == i * (1 ether), "Ill funded");
        }
    }
}
```

External Computation - Example

runtime verification

```
contract ExternalComputationTest is RecordedState {
    function prove_multiple_counters() public view {
        Counter[] memory counters = new Counter[](10);

        // Load all counters
        counters[0] = (Counter(address(counter0Address)));
        counters[1] = (Counter(address(counter1Address)));
        counters[2] = (Counter(address(counter2Address)));
        counters[3] = (Counter(address(counter3Address)));
        counters[4] = (Counter(address(counter4Address)));
        counters[5] = (Counter(address(counter5Address)));
        counters[6] = (Counter(address(counter6Address)));
        counters[7] = (Counter(address(counter7Address)));
        counters[8] = (Counter(address(counter8Address)));
        counters[9] = (Counter(address(counter9Address)));

        for (uint256 i; i <= 9; ++i) {
            require(counters[i].number() == i, "Bad number");
            require(address(counters[i]).balance == i * (1 ether), "Ill funded");
        }
    }
}
```

External Computation - Example

runtime verification

```
contract ExternalComputationTest is RecordedState {
    function prove_multiple_counters() public view {
        Counter[] memory counters = new Counter[](10);

        // Load all counters
        counters[0] = (Counter(address(counter0Address)));
        counters[1] = (Counter(address(counter1Address)));
        counters[2] = (Counter(address(counter2Address)));
        counters[3] = (Counter(address(counter3Address)));
        counters[4] = (Counter(address(counter4Address)));
        counters[5] = (Counter(address(counter5Address)));
        counters[6] = (Counter(address(counter6Address)));
        counters[7] = (Counter(address(counter7Address)));
        counters[8] = (Counter(address(counter8Address)));
        counters[9] = (Counter(address(counter9Address)));

        for (uint256 i; i <= 9; ++i) {
            require(counters[i].number() == i, "Bad number");
            require(address(counters[i]).balance == i * (1 ether), "Ill funded");
        }
    }
}
```

External Computation - Example

runtime verification

Now we `prove` with `--init-node-from-dump`:

```
kontrol prove --mt prove_multiple_counters --init-node-from-dump
state-dump/StateDump.json
```

NatSpec Magic

runtime verification

- NatSpec has a `@custom:` tag, tailored at third party applications
  - It allows us to use Kontrol in a much powerful way
  - We are actively expanding its applications for Kontrol
- What applications does it currently have?
  - Remove branching from symbolic calldata and dynamic types
  - Instantiating interfaces with code (CSE related)
  - [In progress] Assumptions for code outside of tests (CSE related)

NatSpec Magic

runtime verification

- Remove branching from symbolic calldata and dynamic types
  - Kontrol (powered by KEVM) can have symbolic calldata.
  - E.g., `bytes calldata someSymbolicBytes`
  - But having unconstrained `someSymbolicBytes` will result in a lot of branching
  - This is due for all the different compiler checks of `someSymbolicBytes` hidden in plain sight
  - We can pass the desired restrictions via `/// @custom:kontrol-bytes-length-equals someSymbolicBytes: n`

NatSpec Magic - Optimism Example

runtime verification

```
function prove_proveWithdrawalTransaction_paused10(
    Types.WithdrawalTransaction memory _tx,
    uint256 _l2OutputIndex,
    Types.OutputRootProof calldata _outputRootProof,
    bytes[] calldata _withdrawalProof
)
```

- Will branch like crazy on `_withdrawalProof`
- Kontrol is unhappy:

NatSpec Magic - Optimism Example

runtime verification

```
/// @custom:kontrol-array-length-equals _withdrawalProof: 10,
/// @custom:kontrol-bytes-length-equals _withdrawalProof: 600,
function prove_proveWithdrawalTransaction_paused10(
    Types.WithdrawalTransaction memory _tx,
    uint256 _l2OutputIndex,
    Types.OutputRootProof calldata _outputRootProof,
    bytes[] calldata _withdrawalProof
)
```

- Will **not** branch on `_withdrawalProof`
- Kontrol is happy:

Parallelization with Kontrol

runtime verification

- Two ways of parallelizing proofs
  - Run multiple proofs simultaneously
  - Run multiple branches simultaneously

Parallelization with Kontrol

runtime verification

- Run multiple proofs simultaneously
  - Add `--workers n` to `kontrol prove`
  - `kontrol prove --mt test1 --mt test2 --workers 2`
  - Kontrol can be very resource intensive! Use this rule of thumb:
  - Max number of processes is `(RAM - 8) / 8` for a machine of `RAM` GB of RAM

Parallelization with Kontrol

runtime verification

- Run multiple branches simultaneously

Parallelization with Kontrol

runtime verification

- Run multiple branches simultaneously
  - Build with `GHCRTS='' kontrol build`

Parallelization with Kontrol

runtime verification

- Run multiple branches simultaneously
  - Build with `GHCRTS='' kontrol build`
  - Add `--max-frontier-parallel n` to `kontrol prove` with `GHCRTS`:

Parallelization with Kontrol

runtime verification

- Run multiple branches simultaneously
  - Build with `GHCRTS='' kontrol build`
  - Add `--max-frontier-parallel n` to `kontrol prove` with `GHCRTS`:
  - ```
    GHCRTS='-Nn' kontrol prove --mt testFoo
    --max-frontier-parallel n
    ```
    ```

Parallelization with Kontrol

runtime verification

- Run multiple branches simultaneously
  - Build with `GHCRTS='' kontrol build`
  - Add `--max-frontier-parallel n` to `kontrol prove` with `GHCRTS`:
  - ```
    GHCRTS='-Nn' kontrol prove --mt testFoo
    --max-frontier-parallel n
    ```
  - Mind the `(RAM - 8) / 8` rule of thumb!
    ```

More stuff!

runtime verification

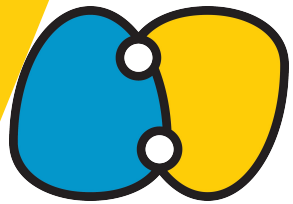
- Kontrol's own cheatcodes
  - github.com/runtimeverification/kontrol-cheatcodes/
- Proof manipulation
  - Remove nodes, merge nodes, …
- KaaS and CI integration
  - Share proofs between KaaS users
  - Use cached proofs for better CI runtimes

Summary

runtime verification

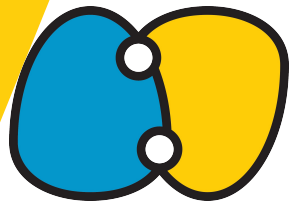
- How Kontrol works
- Usual proving workflow
- Reuse symbolic execution for performance improvements
- Include large computation chunks into Kontrol
- Use Natspec for the greater good
- Parallelizatiooon

runtime verification



Thank You!

runtime verification



Stay in touch!

@rv_inc
@anvacaru
@BatisteFormal

https://discord.com/invite/CurfmXNtbN
https://docs.runtimeverification.com/kontrol
https://github.com/runtimeverification/kontrol
https://t.me/rv_kontrol