Advanced Scribble

Scribble Bootcamp

Dimitar Bounov

DILIGENCE Fuzzing

Review

Review

Function Annotations (#if_succeeds)

Review

Function Annotations (#if_succeeds)
Contract Invariants (#invariant)

Review

Function Annotations (#if_succeeds)
Contract Invariants (#invariant)
Scribble Language Features (old, let, sum, ==>)

Plan

Plan

State Variable Annotations

Plan

State Variable Annotations
Universal Quantification

Plan

State Variable Annotations
Universal Quantification
Statement Annotations

Plan

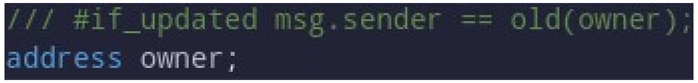
State Variable Annotations
Universal Quantification
Statement Annotations
User Functions

Plan

State Variable Annotations
Universal Quantification
Statement Annotations
User Functions

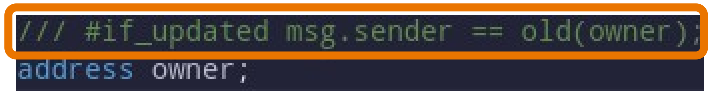
State Variable Annotations

State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```

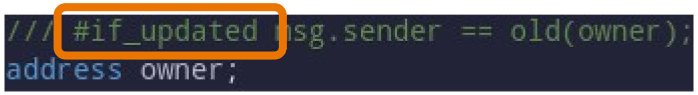
State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```

Docstring annotations above state vars

State Variable Annotations

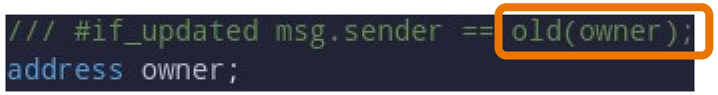


```
/// #if_updated msg.sender == old(owner);
address owner;
```

Docstring annotations above state vars

Checked every time variable is modified

State Variable Annotations



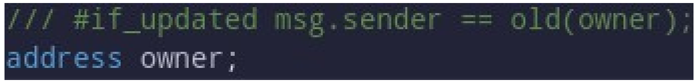
```
/// #if_updated msg.sender == old(owner);
address owner;
```

Docstring annotations above state vars

Checked every time variable is modified

Allows using old()

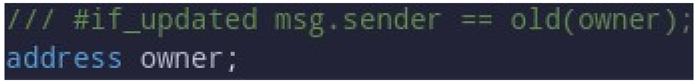
State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```

Predicates checked every time
a state variable is updated

State Variable Annotations

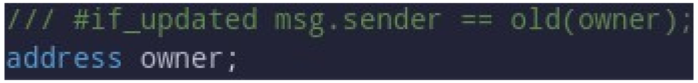


```
/// #if_updated msg.sender == old(owner);
address owner;
```

Predicates checked every time
a state variable is updated

Useful for checking permissions

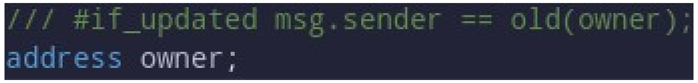
State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```

How do we check state variable invariants?

State Variable Annotations

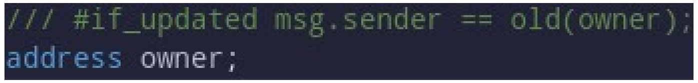


```
/// #if_updated msg.sender == old(owner);
address owner;
```

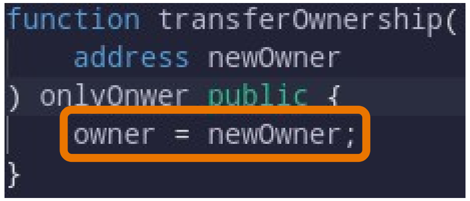
How do we check state variable invariants?

1. Identify all updates to state variable

State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```

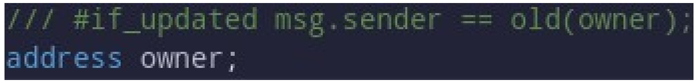


```
function transferOwnership(
    address newOwner
) onlyOnwer public {
    owner = newOwner;
}
```

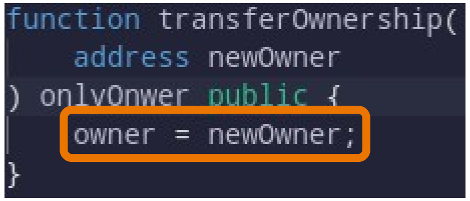
How do we check state variable invariants?

1. Identify all updates to state variable

State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```



```
function transferOwnership(
    address newOwner
) onlyOnwer public {
    owner = newOwner;
}
```

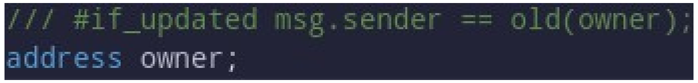
How do we check state variable invariants?

1. Identify all updates to state variable

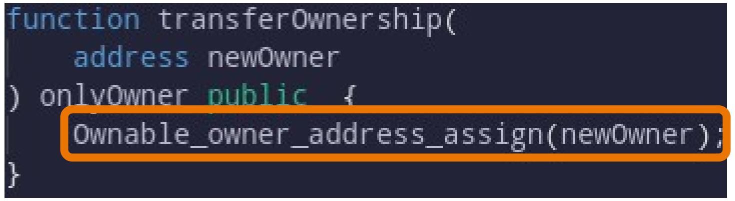
2. Interpose on updates

DILIGENCE Fuzzing

State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```



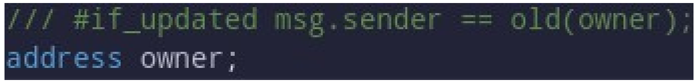
```
function transferOwnership(
    address newOwner
) onlyOwner public  {
    Ownable_owner_address_assign(newOwner);
}
```

How do we check state variable invariants?

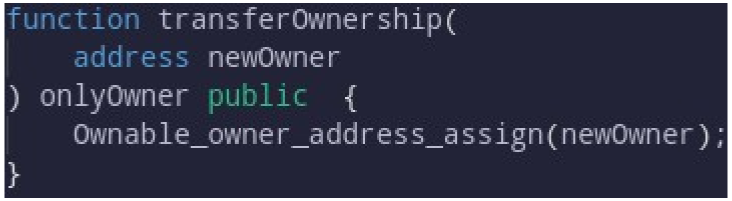
1. Identify all updates to state variable

2. Interpose on updates

State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```



```
function transferOwnership(
    address newOwner
) onlyOwner public  {
    Ownable_owner_address_assign(newOwner);
}
```

How do we check state variable invariants?

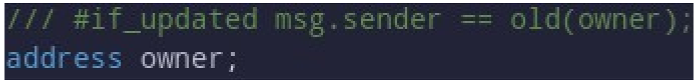
1. Identify all updates to state variable

2. Interpose on updates

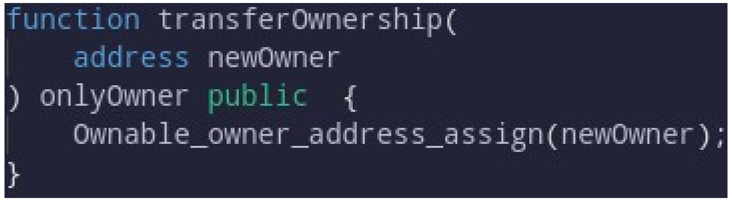
3. Compile annotations in wrapper

DILIGENCE Fuzzing

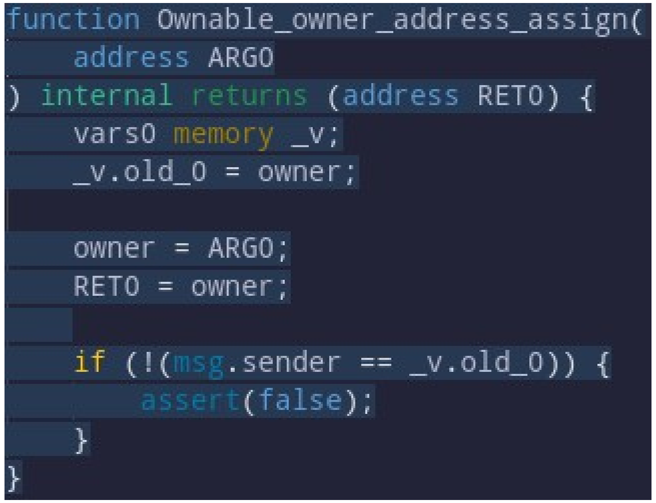
State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```



```
function transferOwnership(
    address newOwner
) onlyOwner public  {
    Ownable_owner_address_assign(newOwner);
}
```



```
function Ownable_owner_address_assign(
    address ARG0
) internal returns (address RET0) {
    vars0 memory _v;
    _v.old_0 = owner;

    owner = ARG0;
    RET0 = owner;

    if (!(msg.sender == _v.old_0)) {
        assert(false);
    }
}
```

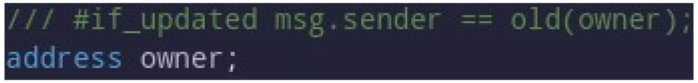
How do we check state variable invariants?

1. Identify all updates to state variable

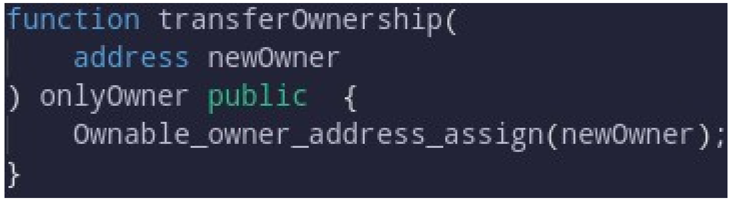
2. Interpose on updates

3. Compile annotations in wrapper

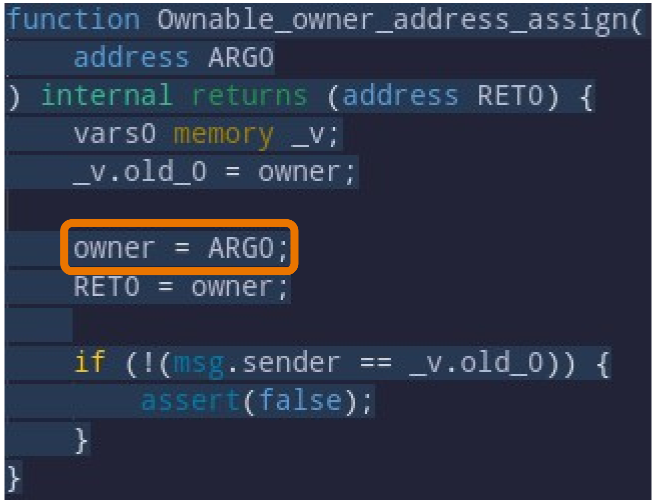
State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```



```
function transferOwnership(
    address newOwner
) onlyOwner public  {
    Ownable_owner_address_assign(newOwner);
}
```



```
function Ownable_owner_address_assign(
    address ARG0
) internal returns (address RET0) {
    vars0 memory _v;
    _v.old_0 = owner;

    owner = ARG0;
    RET0 = owner;

    if (!(msg.sender == _v.old_0)) {
        assert(false);
    }
}
```

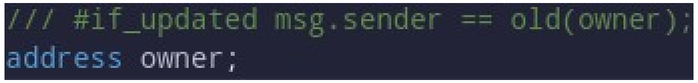
How do we check state variable invariants?

1. Identify all updates to state variable

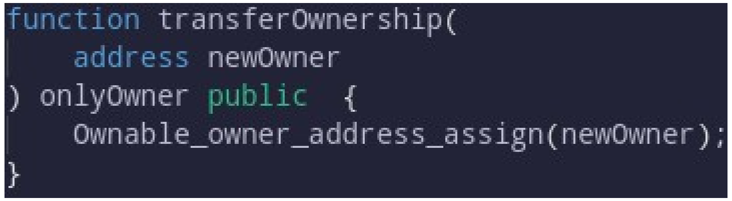
2. Interpose on updates

3. Compile annotations in wrapper

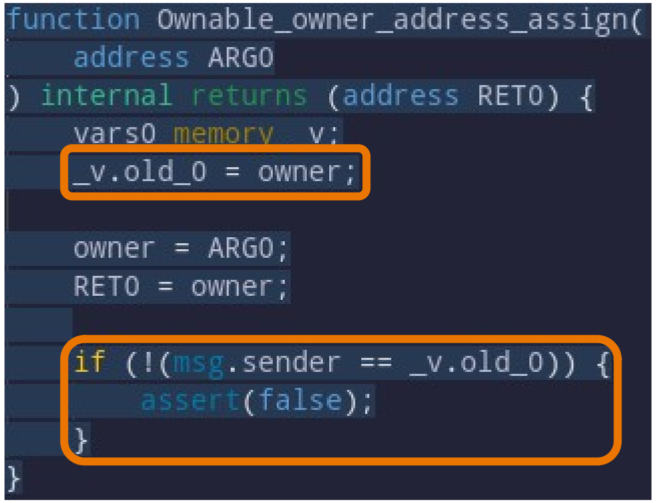
State Variable Annotations



```
/// #if_updated msg.sender == old(owner);
address owner;
```



```
function transferOwnership(
    address newOwner
) onlyOwner public  {
    Ownable_owner_address_assign(newOwner);
}
```



```
function Ownable_owner_address_assign(
    address ARG0
) internal returns (address RET0) {
    vars0 memory _v;
    _v.old_0 = owner;

    owner = ARG0;
    RET0 = owner;

    if (!(msg.sender == _v.old_0)) {
        assert(false);
    }
}
```

How do we check state variable invariants?

1. Identify all updates to state variable

2. Interpose on updates

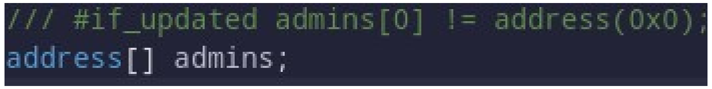
3. Compile annotations in wrapper

DILIGENCE Fuzzing

State Variable Annotations

What about aliasing?

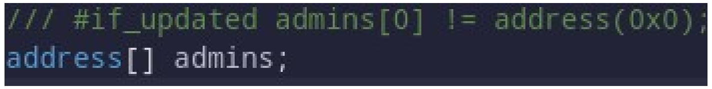
State Variable Annotations



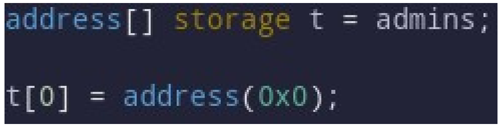
```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```

What about aliasing?

State Variable Annotations



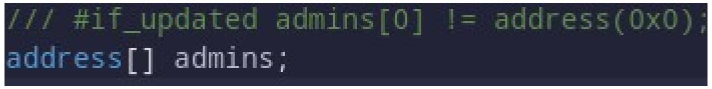
```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```



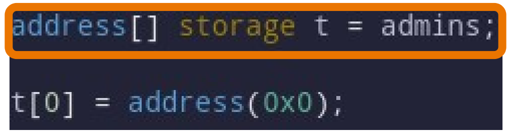
```
address[] storage t = admins;

t[0] = address(0x0);
```

What about aliasing?

State Variable Annotations



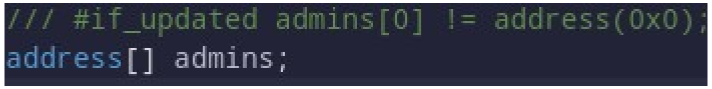
```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```



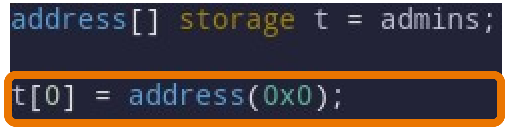
```
address[] storage t = admins;

t[0] = address(0x0);
```

What about aliasing?

State Variable Annotations



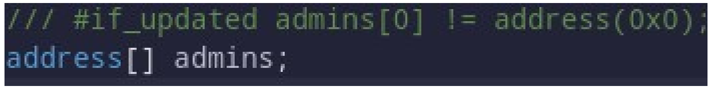
```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```



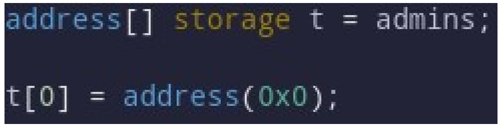
```
address[] storage t = admins;

t[0] = address(0x0);
```

What about aliasing?

State Variable Annotations



```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```



```
address[] storage t = admins;

t[0] = address(0x0);
```

What about aliasing?

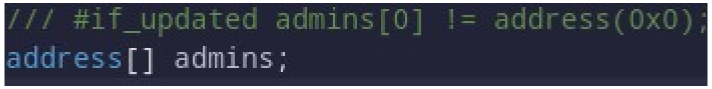
Reject annotation on potentially aliased vars

Demo

Plan

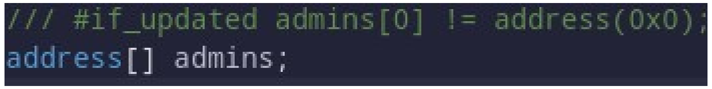
State Variable Annotations
Universal Quantification
Statement Annotations
User Functions

Universal Quantifiers



```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```

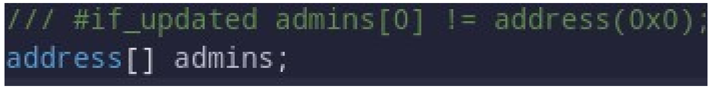
Universal Quantifiers



```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```

What if we want to talk about all admins?

Universal Quantifiers

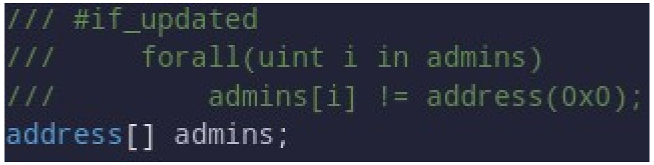


```
/// #if_updated admins[0] != address(0x0);
address[] admins;
```

What if we want to talk about all admins?

Scribble allows universal quantification!

Universal Quantifiers

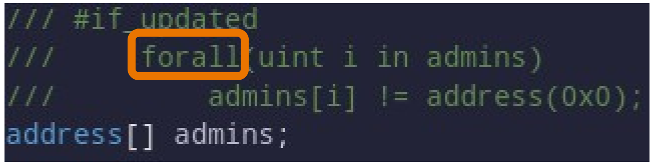


```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

What if we want to talk about all admins?

Scribble allows universal quantification!

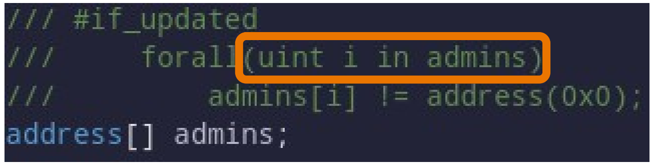
Universal Quantifiers



```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

Introduced with forall keyword

Universal Quantifiers

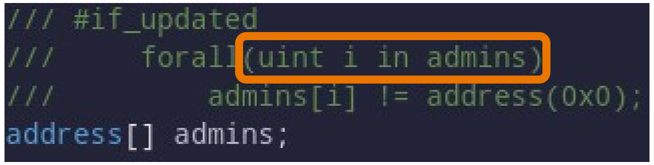


```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

Introduced with forall keyword

Can iterate over indices of an array

Universal Quantifiers



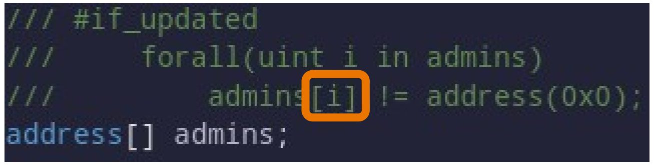
```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

Introduced with forall keyword

Can iterate over indices of an array

Can iterate over explicit range a...b

Universal Quantifiers



```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

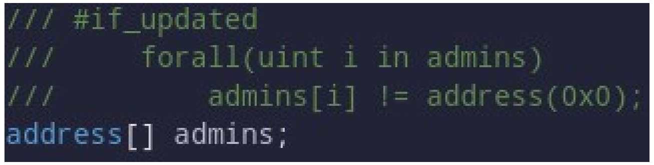
Introduced with forall keyword

Can iterate over indices of an array

Can iterate over explicit range a...b

Loop variable only valid in forall body

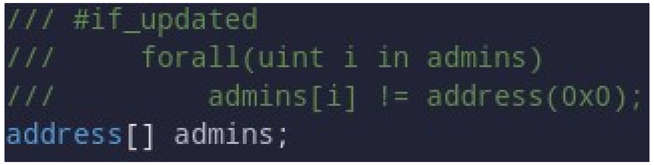
Universal Quantifiers



```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

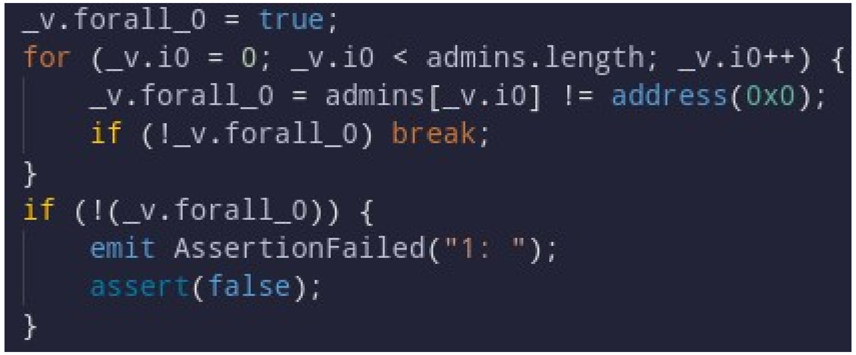
How are universal quantifiers instrumented?

Universal Quantifiers



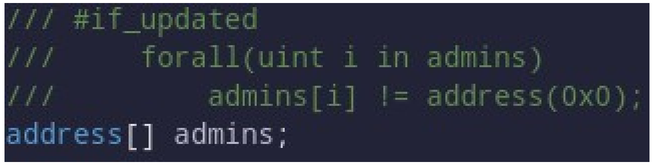
```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

How are universal quantifiers instrumented?

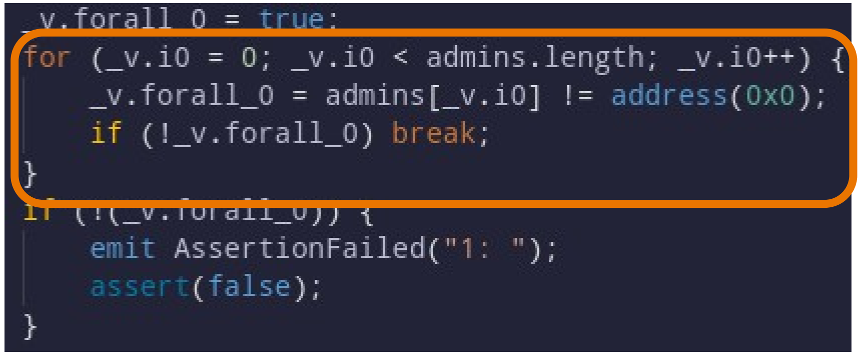


```
_v.forall_0 = true;
for (_v.i0 = 0; _v.i0 < admins.length; _v.i0++) {
    _v.forall_0 = admins[_v.i0] != address(0x0);
    if (!_v.forall_0) break;
}
if (!(_v.forall_0)) {
    emit AssertionFailed("1: ");
    assert(false);
}
```

Universal Quantifiers



```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

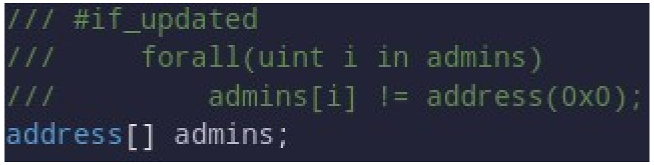


```
_v.forall_0 = true;
for (_v.i0 = 0; _v.i0 < admins.length; _v.i0++) {
    _v.forall_0 = admins[_v.i0] != address(0x0);
    if (!_v.forall_0) break;
}
if (!(_v.forall_0)) {
    emit AssertionFailed("1: ");
    assert(false);
}
```

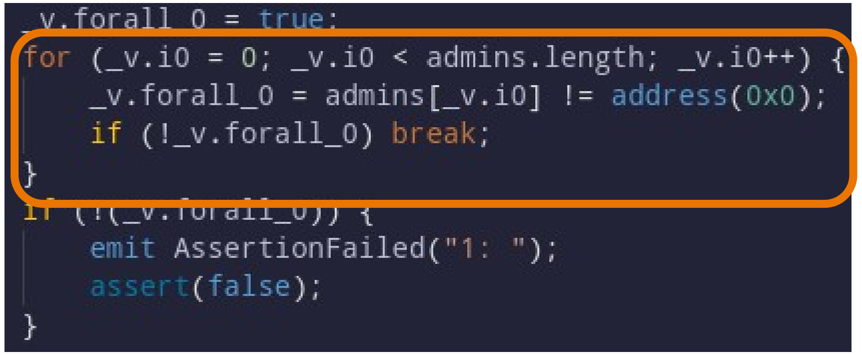
How are universal quantifiers instrumented?

Currently loop inserted at every check site

Universal Quantifiers



```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```



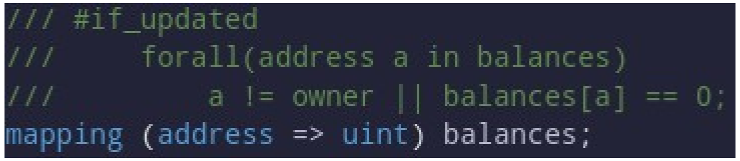
```
_v.forall_0 = true;
for (_v.i0 = 0; _v.i0 < admins.length; _v.i0++) {
    _v.forall_0 = admins[_v.i0] != address(0x0);
    if (!_v.forall_0) break;
}
if (!(_v.forall_0)) {
    emit AssertionFailed("1: ");
    assert(false);
}
```

How are universal quantifiers instrumented?

Currently loop inserted at every check site

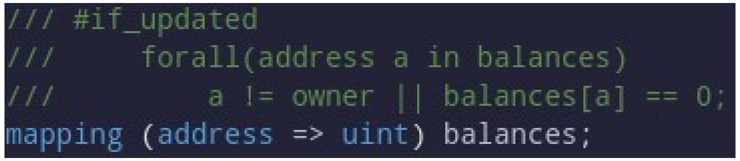
Runtime cost may slow down some tools

Universal Quantifiers



```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
mapping (address => uint) balances;
```

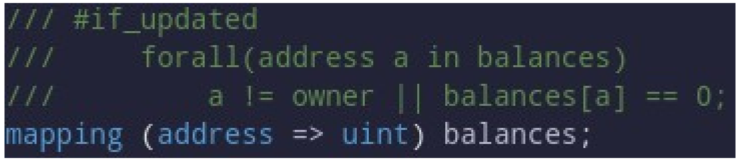
Universal Quantifiers



```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
mapping (address => uint) balances;
```

Universal quantification supported for maps

Universal Quantifiers



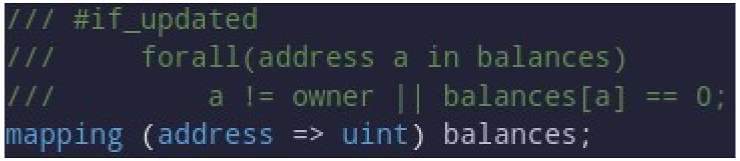
```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
mapping (address => uint) balances;
```

Universal quantification supported for maps

Iteration only over **explicitly assigned** keys

DILIGENCE Fuzzing

Universal Quantifiers



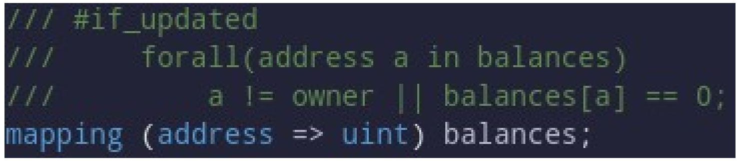
```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
mapping (address => uint) balances;
```

Universal quantification supported for maps

Iteration only over **explicitly assigned** keys

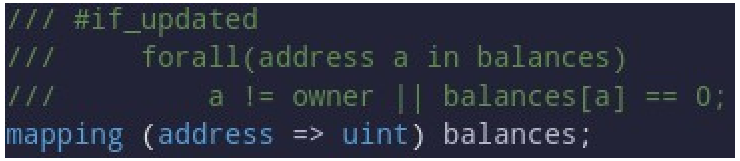
Maps re-written to keep track of keys

Universal Quantifiers



```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
mapping (address => uint) balances;
```

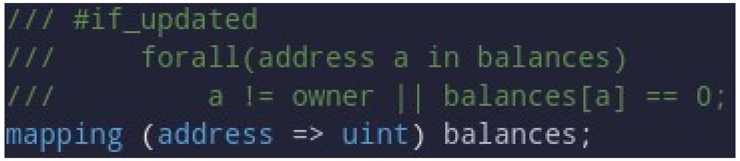
Universal Quantifiers



```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
mapping (address => uint) balances;
```

Map instrumentation more heavy

Universal Quantifiers

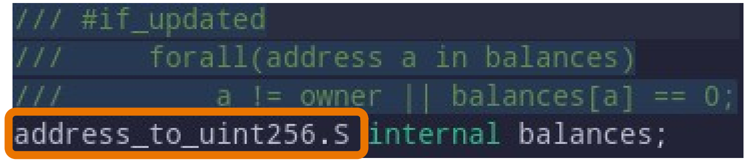


```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
mapping (address => uint) balances;
```

Map instrumentation more heavy

1. Maps re-written to custom structs tracking keys

Universal Quantifiers

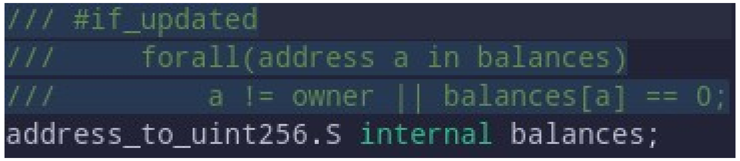


```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
address_to_uint256.S internal balances;
```

Map instrumentation more heavy

1. Maps re-written to custom structs tracking keys

Universal Quantifiers



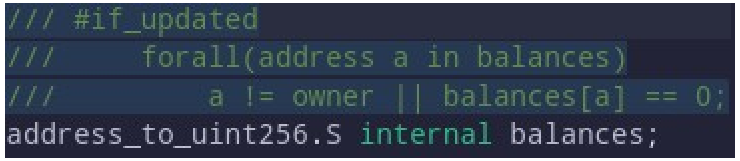
```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
address_to_uint256.S internal balances;
```

Map instrumentation more heavy

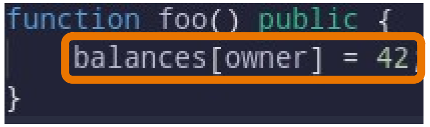
1. Maps re-written to custom structs tracking keys

2. All map updates also re-written

Universal Quantifiers



```
/// #if_updated
///      forall(address a in balances)
///           a != owner || balances[a] == 0;
address_to_uint256.S internal balances;
```



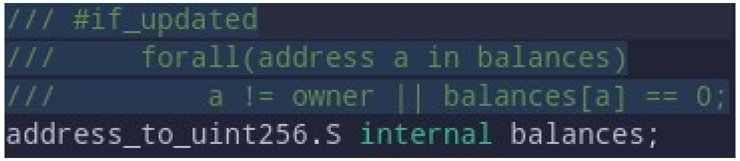
```
function foo() public {
    balances[owner] = 42;
}
```

Map instrumentation more heavy

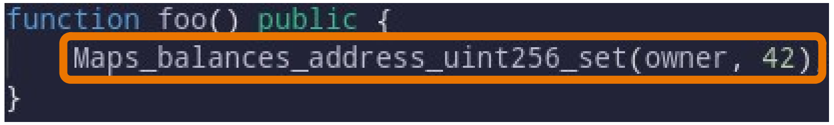
1. Maps re-written to custom structs tracking keys

2. All map updates also re-written

Universal Quantifiers



```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
address_to_uint256.S internal balances;
```



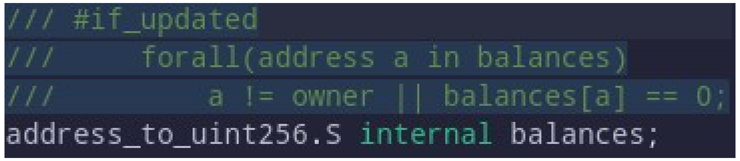
```
function foo() public {
    Maps_balances_address_uint256_set(owner, 42)
}
```

Map instrumentation more heavy

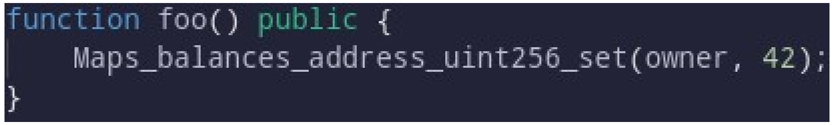
1. Maps re-written to custom structs tracking keys

2. All map updates also re-written

Universal Quantifiers



```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
address_to_uint256.S internal balances;
```



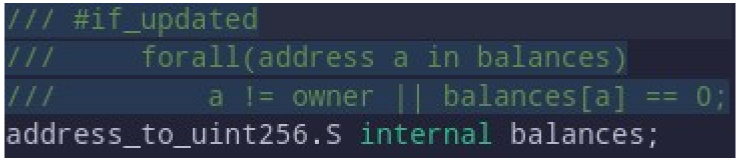
```
function foo() public {
    Maps_balances_address_uint256_set(owner, 42);
}
```

Map instrumentation more heavy

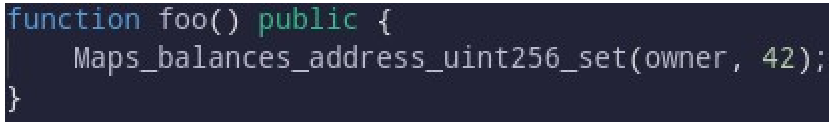
1. Maps re-written to custom structs tracking keys

2. All map updates also re-written

Universal Quantifiers



```
/// #if_updated
///     forall(address a in balances)
///         a != owner || balances[a] == 0;
address_to_uint256.S internal balances;
```



```
function foo() public {
    Maps_balances_address_uint256_set(owner, 42);
}
```

Map instrumentation more heavy

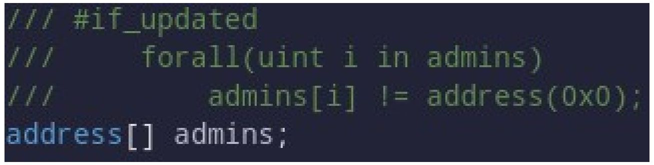
1. Maps re-written to custom structs tracking keys

2. All map updates also re-written

3. Generate wrappers for all updates

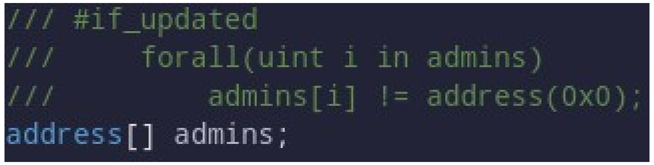
Demo

State Variables Again



```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

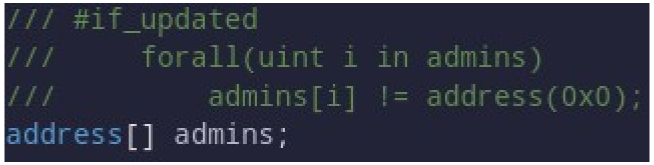
State Variables Again



```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

Forall instrumentation is heavy

State Variables Again

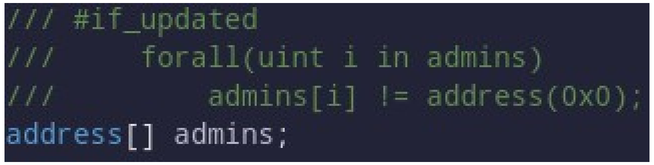


```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

Forall instrumentation is heavy

We check entire array even if 1 index updated

State Variables Again



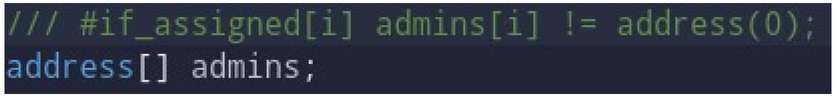
```
/// #if_updated
///     forall(uint i in admins)
///         admins[i] != address(0x0);
address[] admins;
```

Forall instrumentation is heavy

We check entire array even if 1 index updated

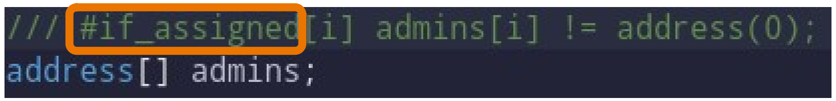
We could do better in some cases with #if_assigned

State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

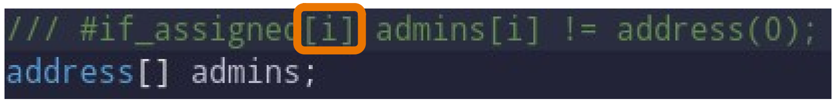
State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Only applies to **explicit** assignments

State Variables Again

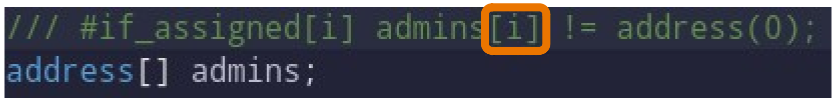


```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Only applies to **explicit** assignments

The `[i]` allows us to name the index at which the update is happening

State Variables Again



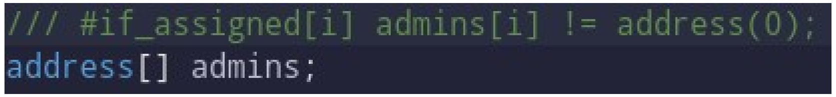
```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Only applies to **explicit** assignments

The `[i]` allows us to name the index at which the update is happening

Can use `i` in the body of the annotation

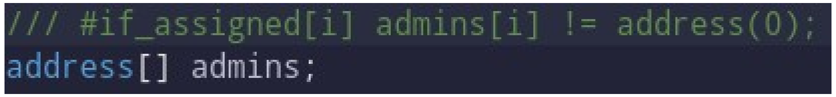
State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

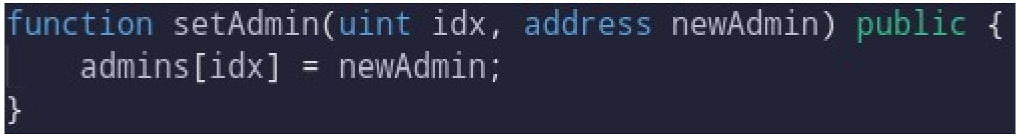
Explicit assignments re-written

State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

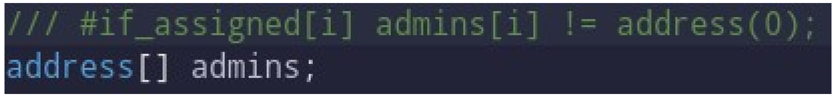
Explicit assignments re-written



```
function setAdmin(uint idx, address newAdmin) public {
    admins[idx] = newAdmin;
}
```

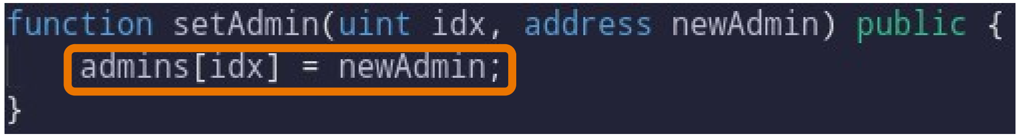
DILIGENCE Fuzzing

State Variables Again



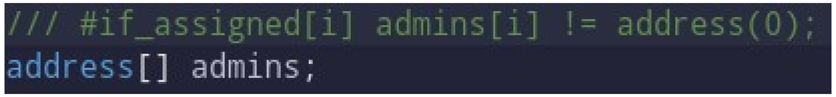
```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Explicit assignments re-written



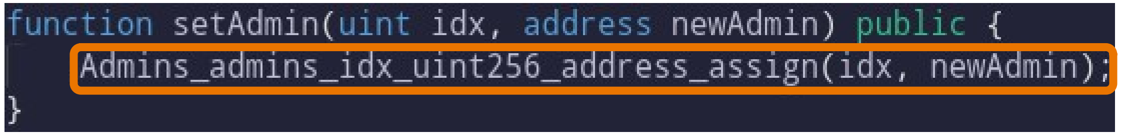
```
function setAdmin(uint idx, address newAdmin) public {
    admins[idx] = newAdmin;
}
```

State Variables Again



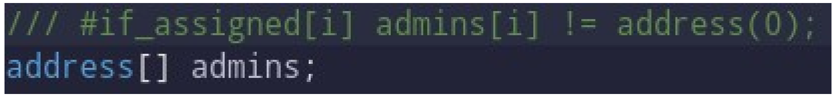
```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Explicit assignments re-written



```
function setAdmin(uint idx, address newAdmin) public {
    Admins_admins_idx_uint256_address_assign(idx, newAdmin);
}
```

State Variables Again

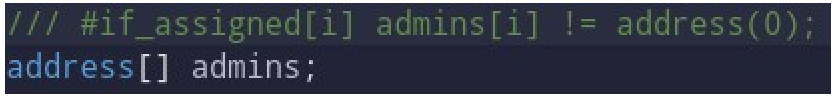

```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Explicit assignments re-written

Non-explicit updates don’t get re-written!

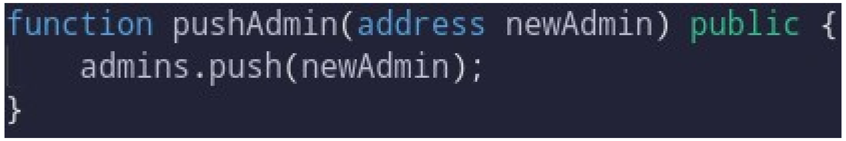
DILIGENCE Fuzzing

State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

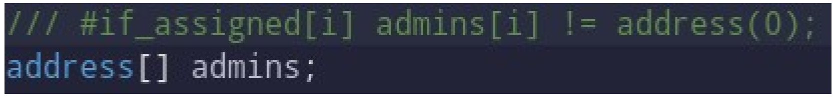
Explicit assignments re-written



```
function pushAdmin(address newAdmin) public {
    admins.push(newAdmin);
}
```

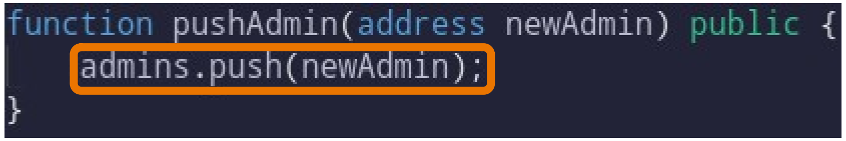
Non-explicit updates don’t get re-written!

State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

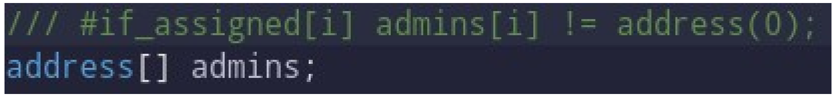
Explicit assignments re-written



```
function pushAdmin(address newAdmin) public {
    admins.push(newAdmin);
}
```

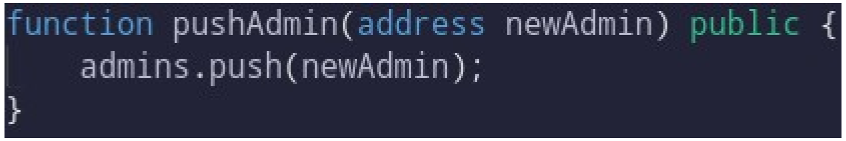
Non-explicit updates don’t get re-written!

State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Explicit assignments re-written

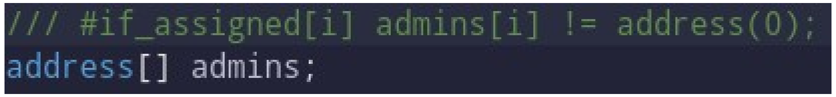


```
function pushAdmin(address newAdmin) public {
    admins.push(newAdmin);
}
```

Non-explicit updates don’t get re-written!

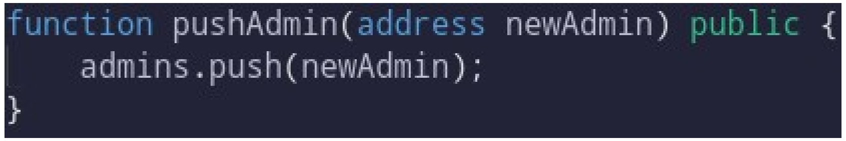
Prefer if_updated over if_assigned

State Variables Again



```
/// #if_assigned[i] admins[i] != address(0);
address[] admins;
```

Explicit assignments re-written



```
function pushAdmin(address newAdmin) public {
    admins.push(newAdmin);
}
```

Non-explicit updates don’t get re-written!

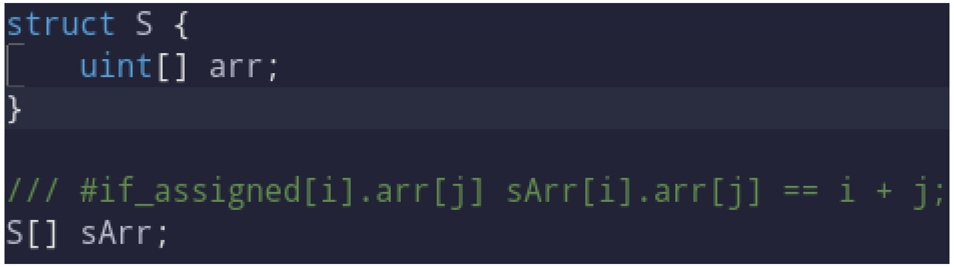
Prefer if_updated over if_assigned

Don’t assume all updates are covered with if_assigned.
Check the code manually!

State Variables Again

if_assigned works for nested arrays and structs

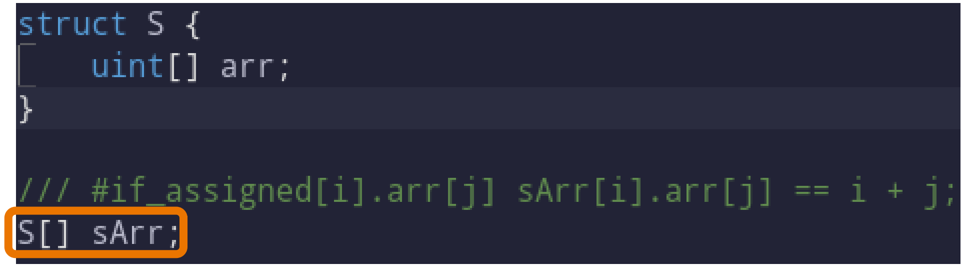
State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

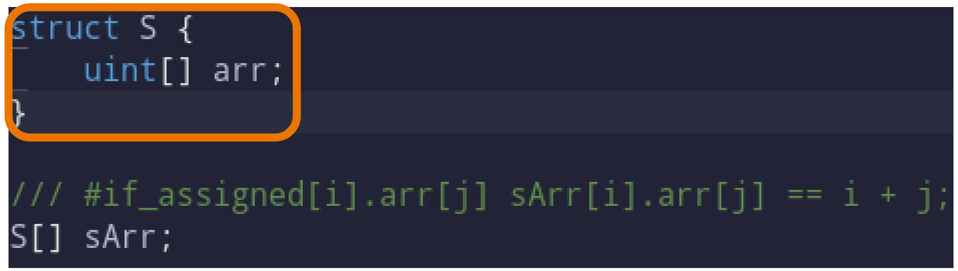
State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

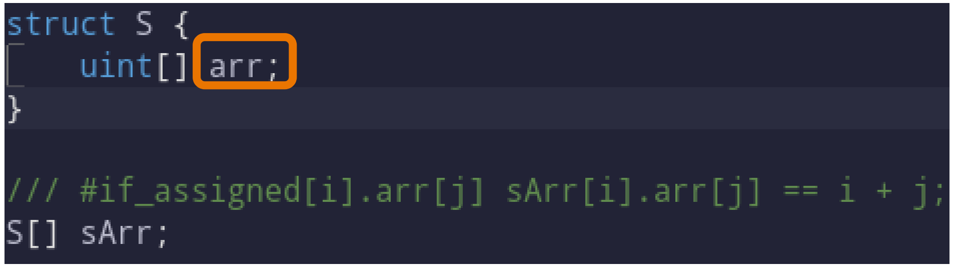
State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

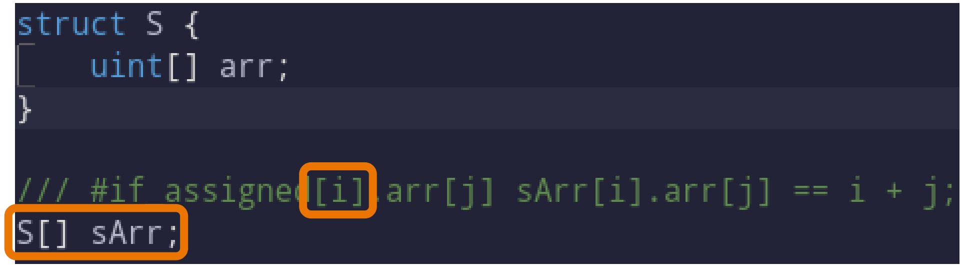
State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

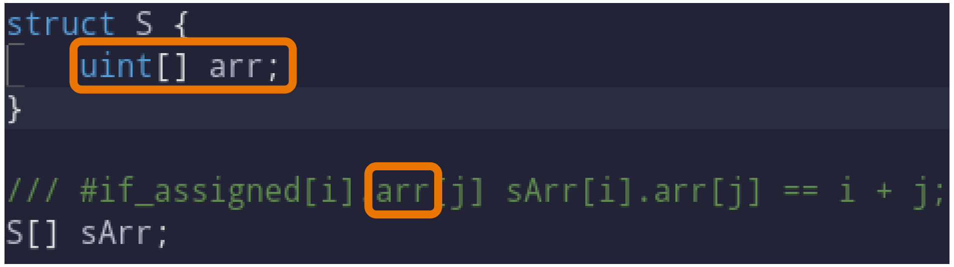
State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

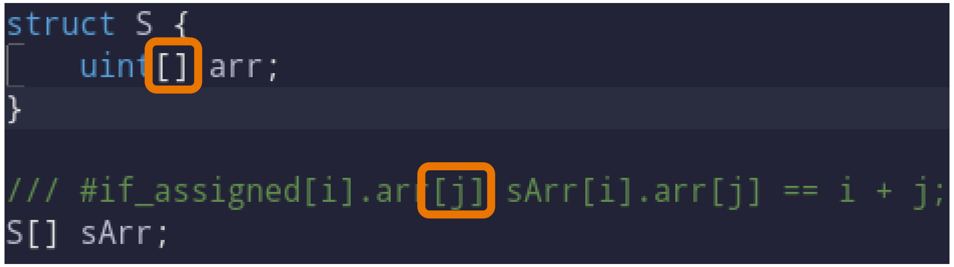
State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

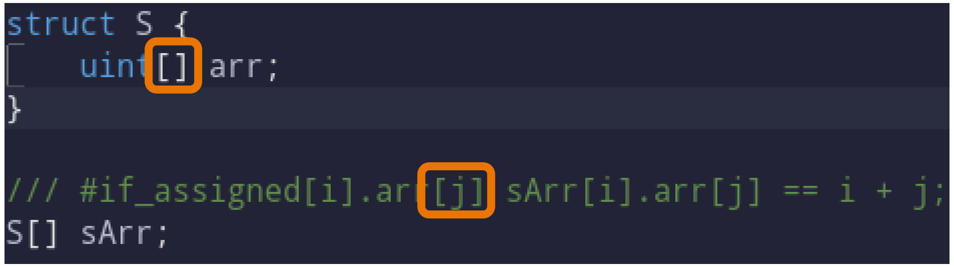
State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

State Variables Again



```
struct S {
    uint[] arr;
}

/// #if_assigned[i].arr[j] sArr[i].arr[j] == i + j;
S[] sArr;
```

if_assigned works for nested arrays and structs

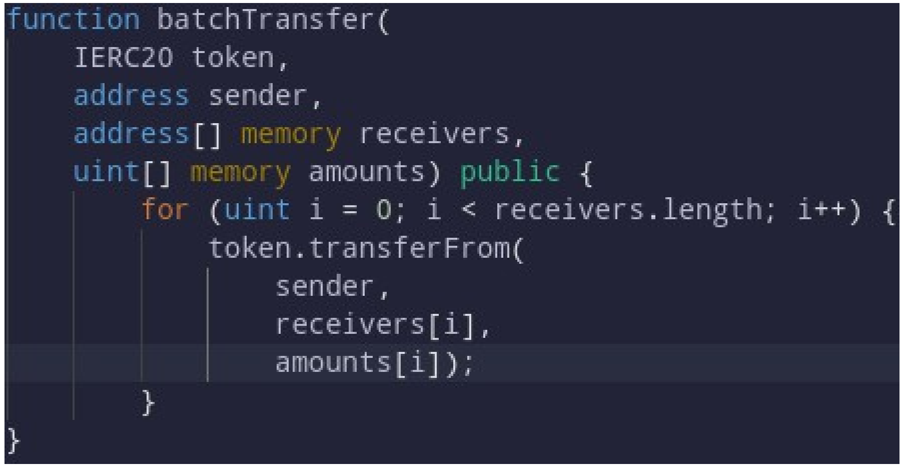
Higher-level assignments (e.g. `sArr =..` or `sArr[i] = ...`) won’t be instrumented!

Plan

State Variable Annotations
Universal Quantification
Statement Annotations
User Functions

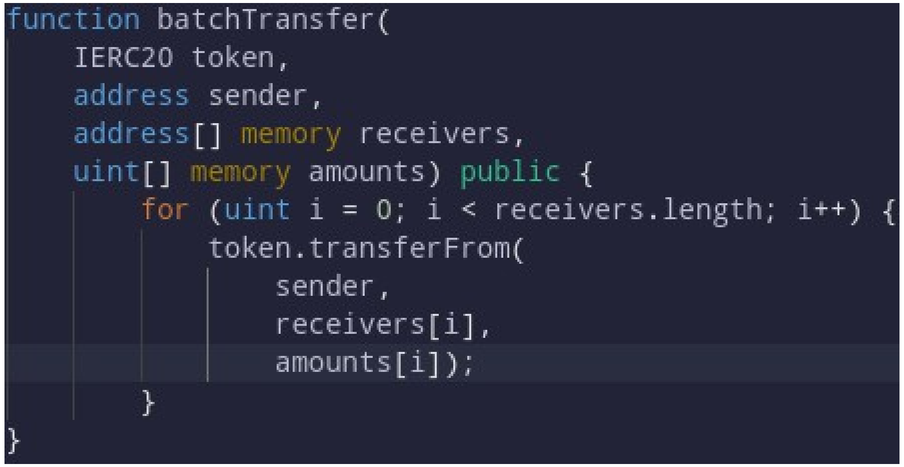
Statement Annotations

Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

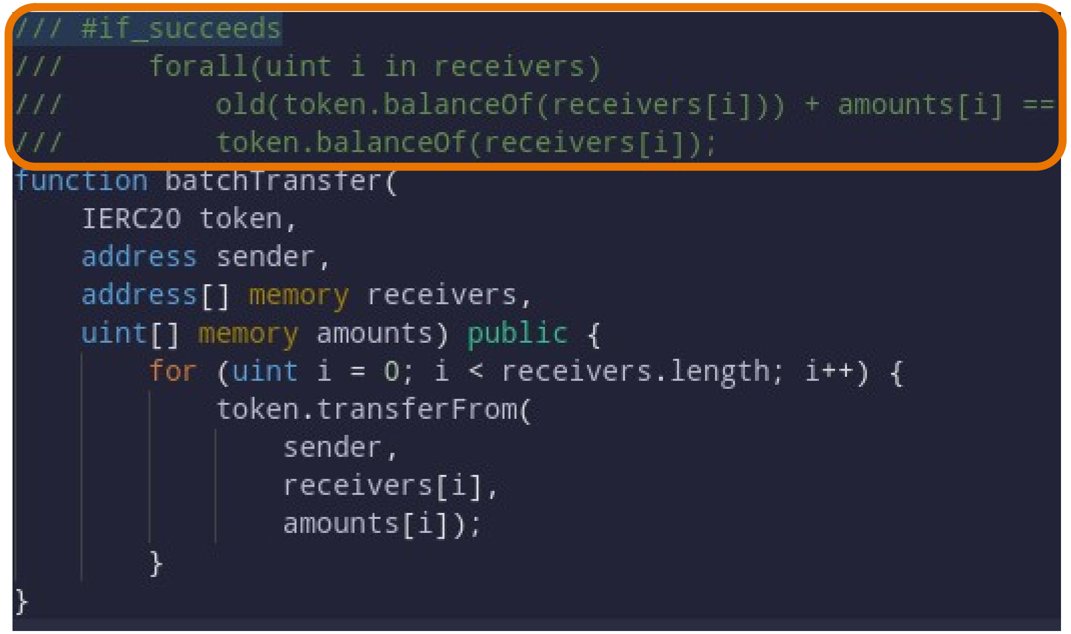
Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

How do we specify that all transfers succeed?

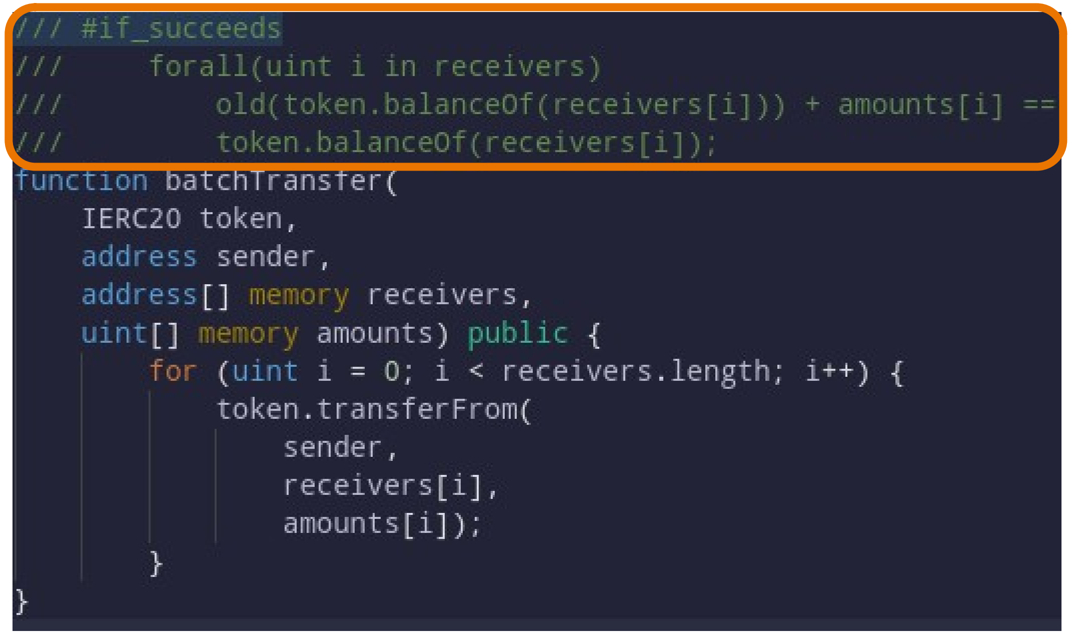
Statement Annotations



```
/// #if_succeeds
///     forall(uint i in receivers)
///         old(token.balanceOf(receivers[i])) + amounts[i] ==
///         token.balanceOf(receivers[i]);
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

How do we specify that all transfers succeed?

Statement Annotations

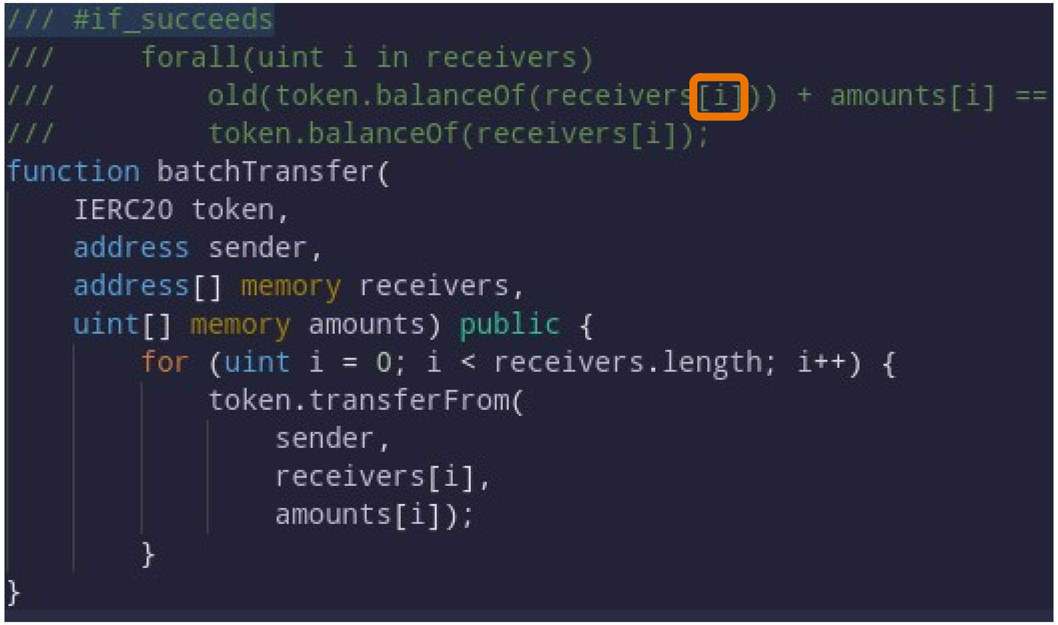


```
/// #if_succeeds
///     forall(uint i in receivers)
///         old(token.balanceOf(receivers[i])) + amounts[i] ==
///         token.balanceOf(receivers[i]);
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

How do we specify that all transfers succeed?

Not instrumentable with scribble currently

Statement Annotations



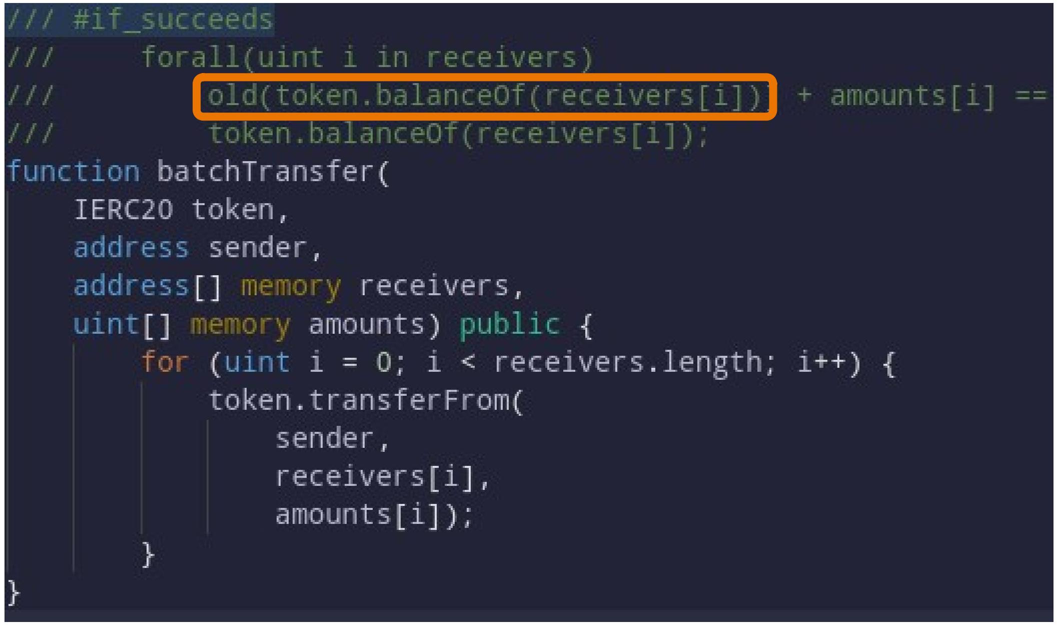
```
/// #if_succeeds
///     forall(uint i in receivers)
///         old(token.balanceOf(receivers[i])) + amounts[i] ==
///         token.balanceOf(receivers[i]);
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

How do we specify that all transfers succeed?

Not instrumentable with scribble currently

DILIGENCE Fuzzing

Statement Annotations



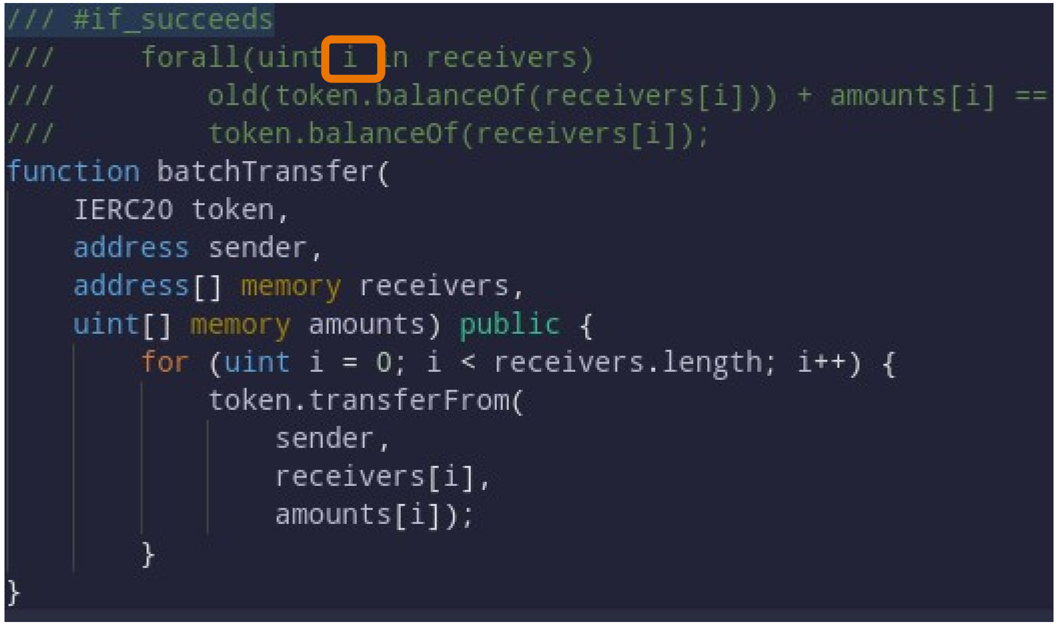
```
/// #if_succeeds
///     forall(uint i in receivers)
///         old(token.balanceOf(receivers[i])) + amounts[i] ==
///         token.balanceOf(receivers[i]);
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

How do we specify that all transfers succeed?

Not instrumentable with scribble currently

DILIGENCE Fuzzing

Statement Annotations

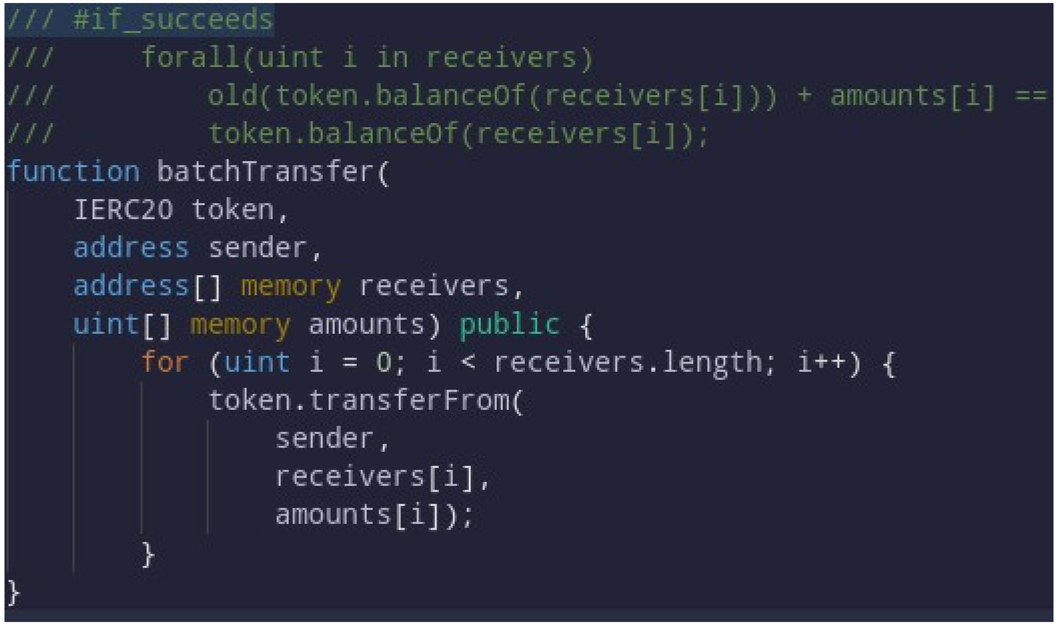


```
/// #if_succeeds
///     forall(uint i in receivers)
///         old(token.balanceOf(receivers[i])) + amounts[i] ==
///         token.balanceOf(receivers[i]);
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

How do we specify that all transfers succeed?

Not instrumentable with scribble currently

Statement Annotations



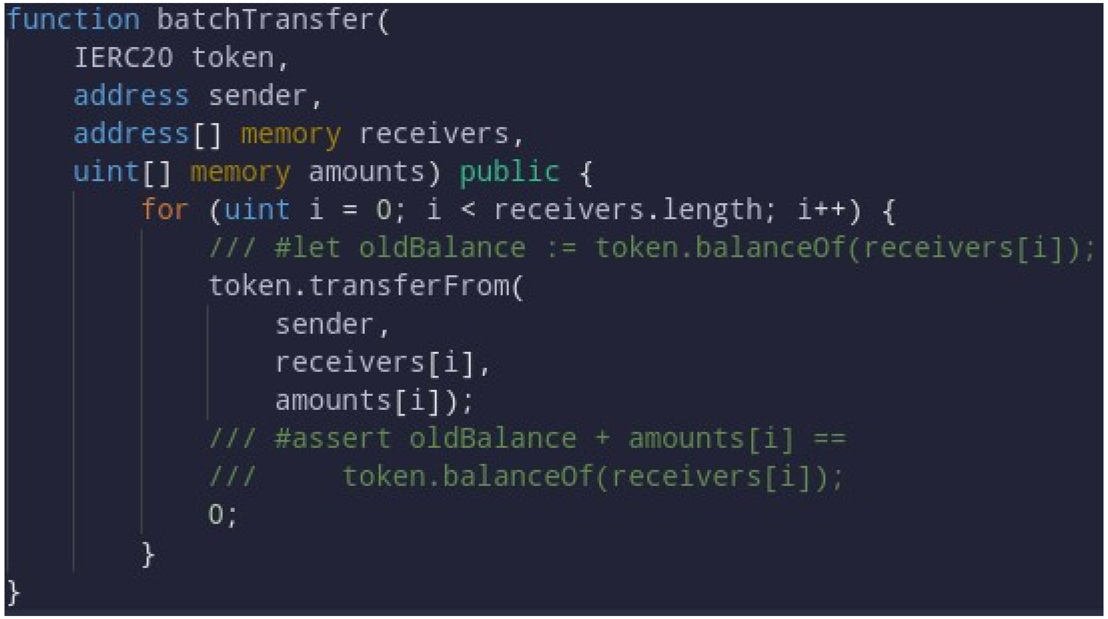
```
/// #if_succeeds
///     forall(uint i in receivers)
///         old(token.balanceOf(receivers[i])) + amounts[i] ==
///         token.balanceOf(receivers[i]);
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
        }
}
```

How do we specify that all transfers succeed?

Not instrumentable with scribble currently

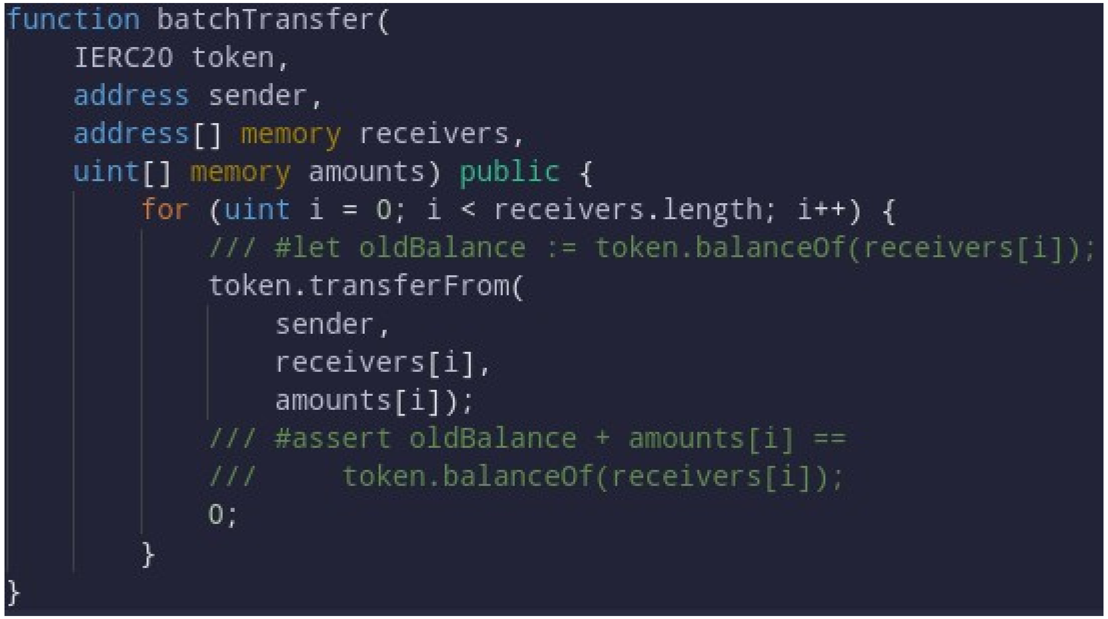
Not true when receivers has duplicates

Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

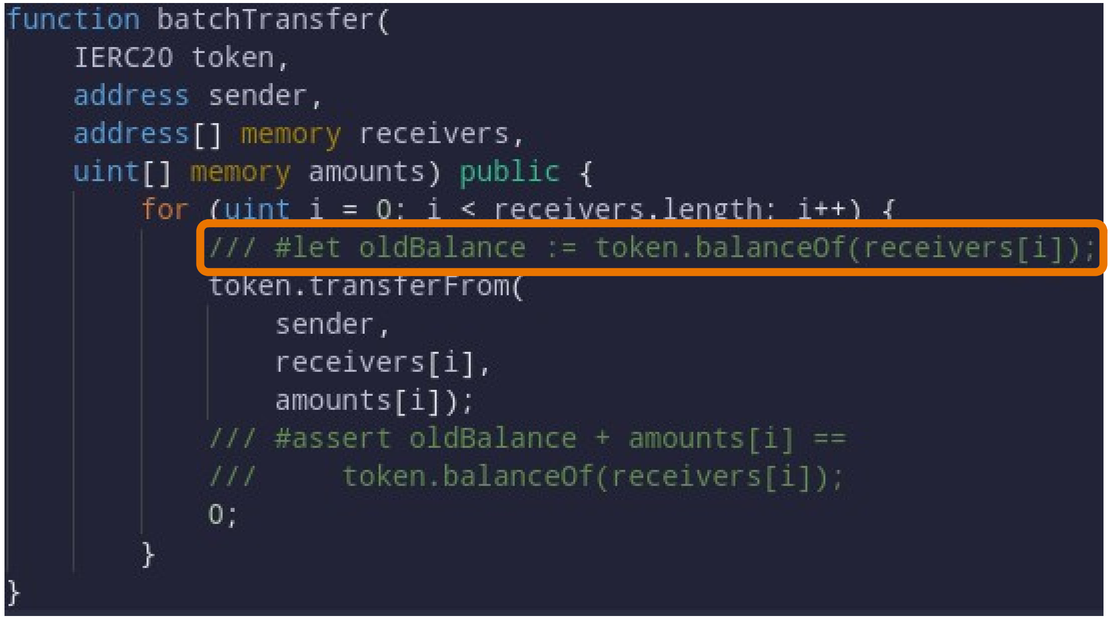
Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

Use inline #let and #assert

Statement Annotations



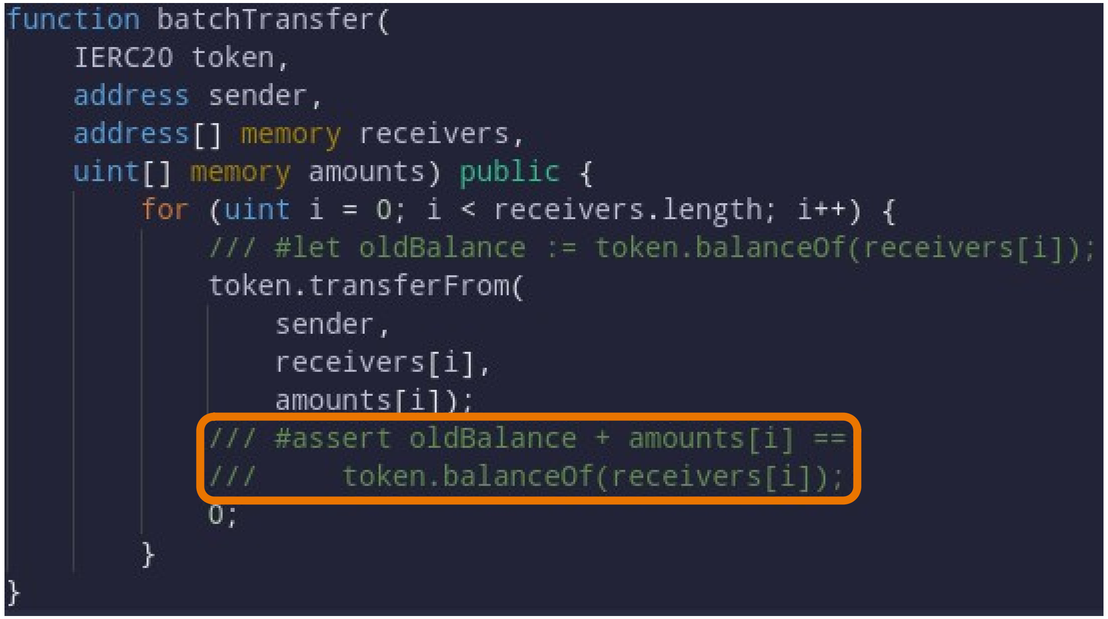
```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

Use inline #let and #assert

let allows arbitrary local binding

DILIGENCE Fuzzing

Statement Annotations



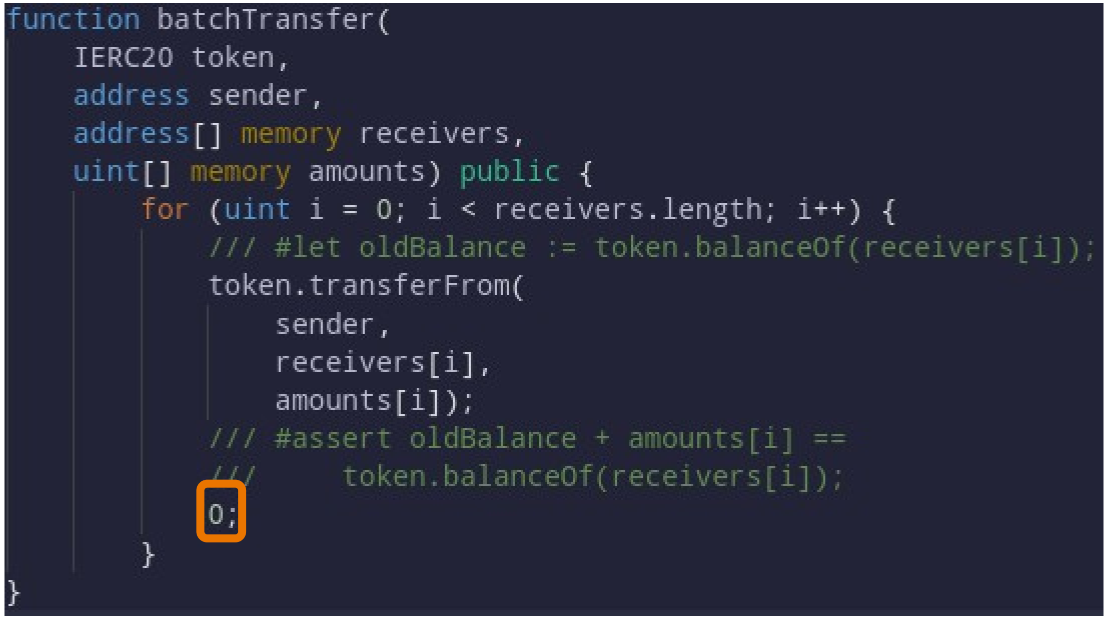
```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

Use inline #let and #assert

let allows arbitrary local binding

#assert equivalent to solidity assert

Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

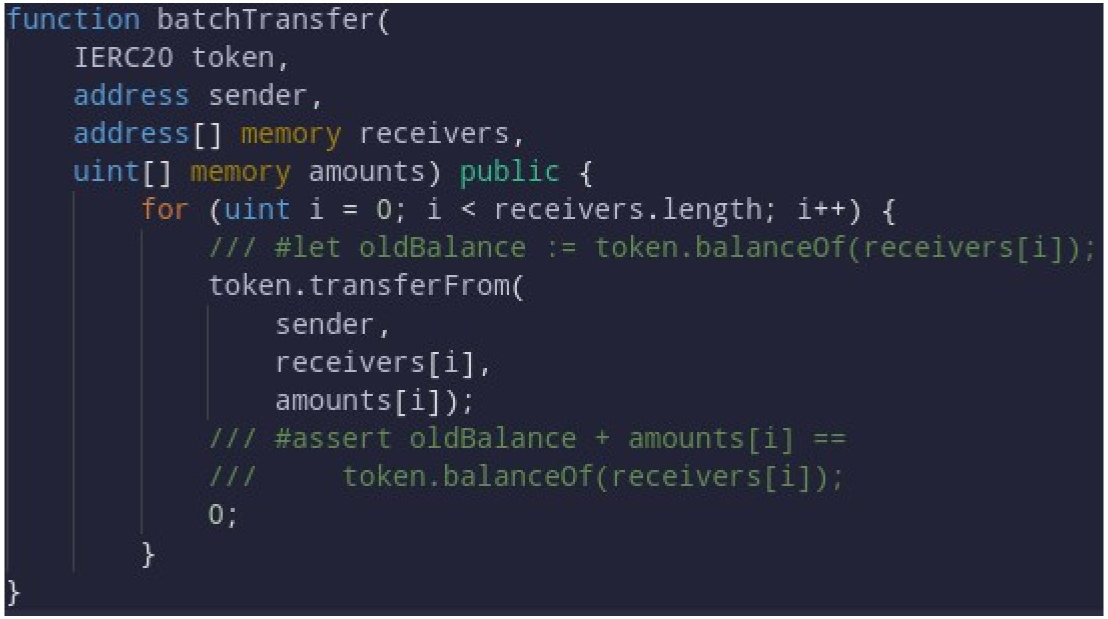
Use inline #let and #assert

let allows arbitrary local binding

#assert equivalent to solidity assert

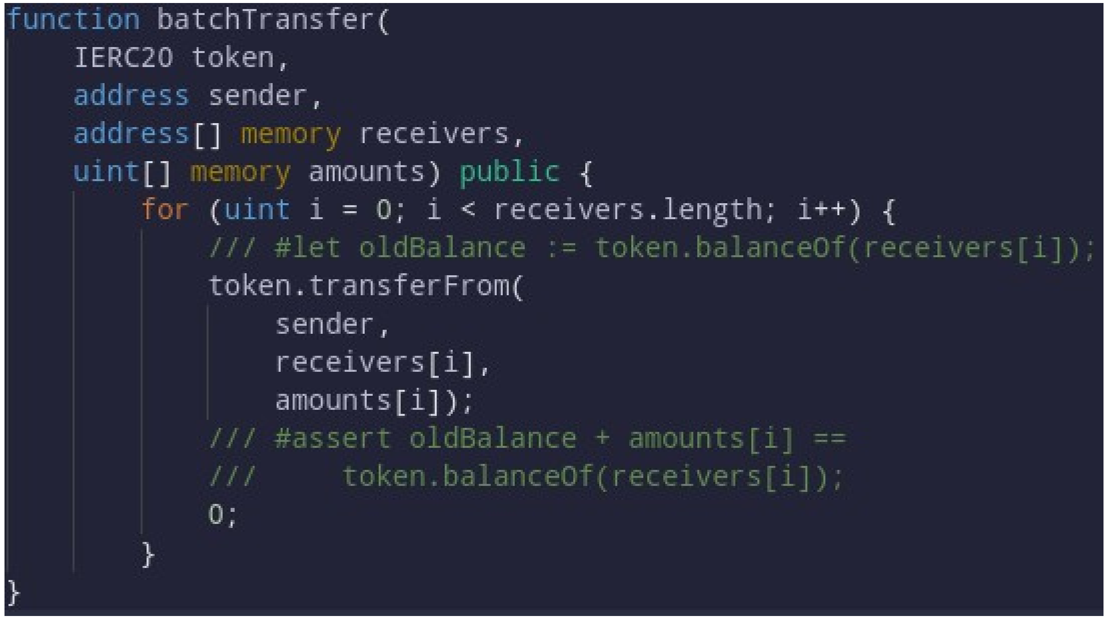
Dummy statement needed for annotation at end of block

Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

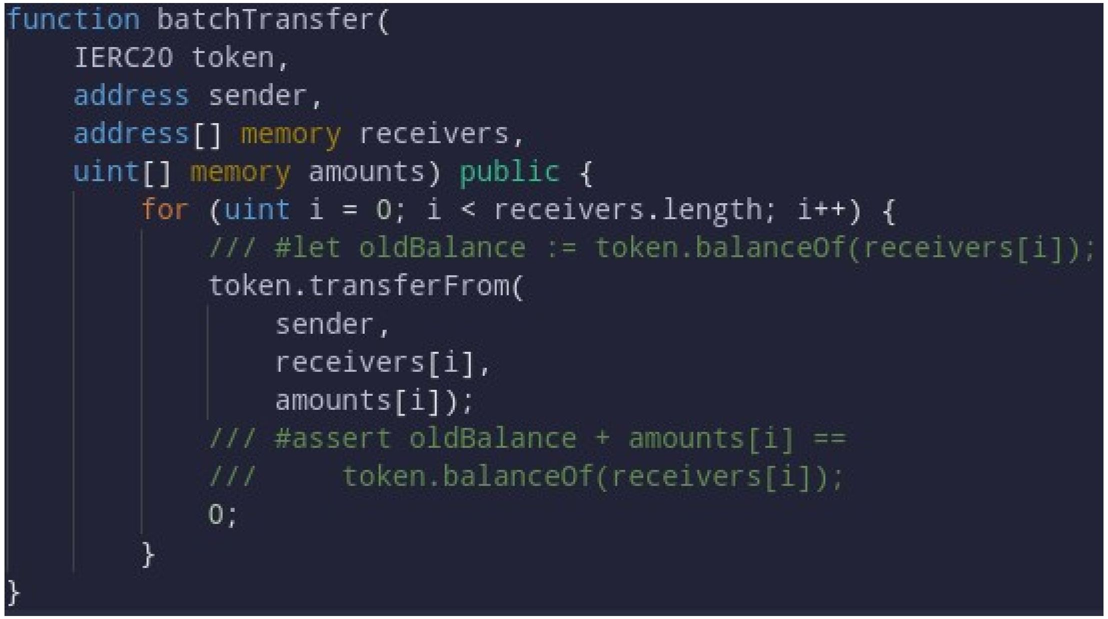
Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

Useful when other annotations not expressive enough

Statement Annotations



```
function batchTransfer(
    IERC20 token,
    address sender,
    address[] memory receivers,
    uint[] memory amounts) public {
        for (uint i = 0; i < receivers.length; i++) {
            /// #let oldBalance := token.balanceOf(receivers[i]);
            token.transferFrom(
                sender,
                receivers[i],
                amounts[i]);
            /// #assert oldBalance + amounts[i] ==
            ///     token.balanceOf(receivers[i]);
            0;
        }
}
```

Useful when other annotations not expressive enough

Useful to reduce number of universal quantifiers

Plan

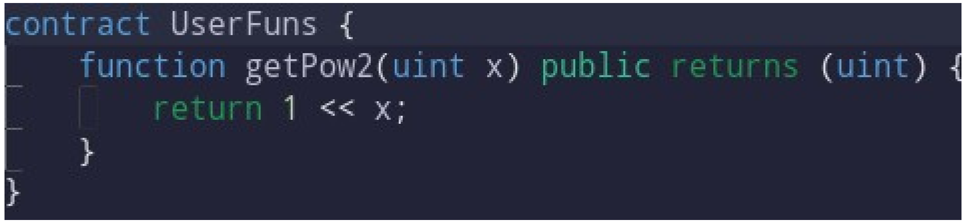
State Variable Annotations
Universal Quantification
Statement Annotations
User Functions

User Functions

User Functions

You can define your own scribble functions!

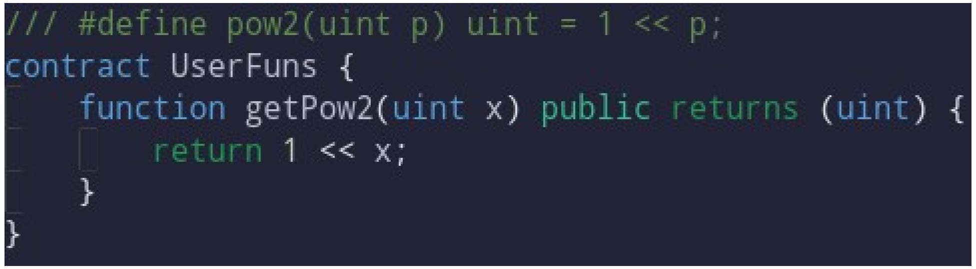
User Functions



```
contract UserFuns {
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

You can define your own scribble functions!

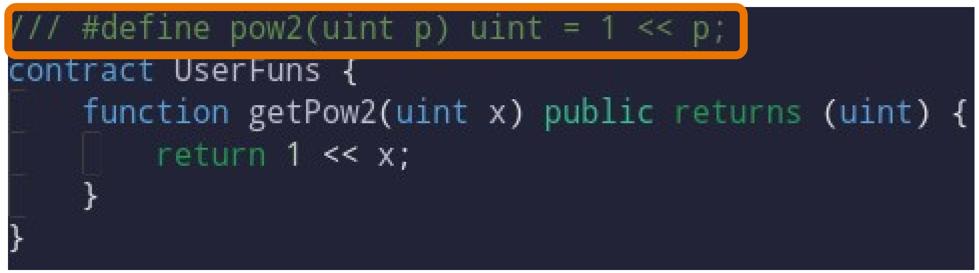
User Functions



```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

You can define your own scribble functions!

User Functions

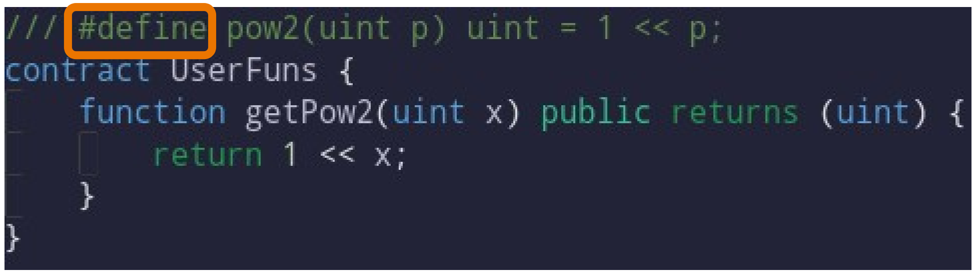


```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

You can define your own scribble functions!

Defined on **contract** docstring

User Functions



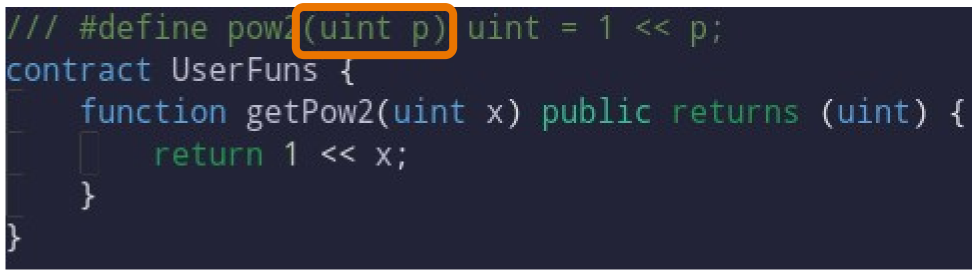
```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

You can define your own scribble functions!

Defined on **contract** docstring

Introduced with #define keyword

User Functions



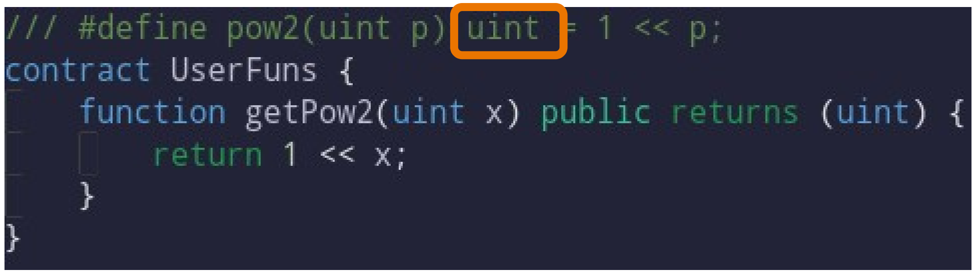
```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

You can define your own scribble functions!

Defined on **contract** docstring

Introduced with #define keyword

User Functions



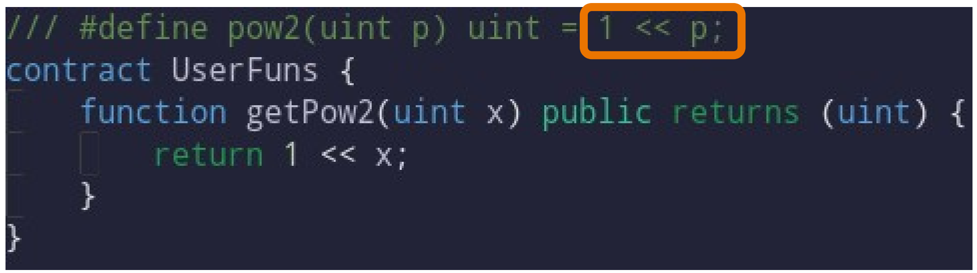
```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

You can define your own scribble functions!

Defined on **contract** docstring

Introduced with #define keyword

User Functions



```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

You can define your own scribble functions!

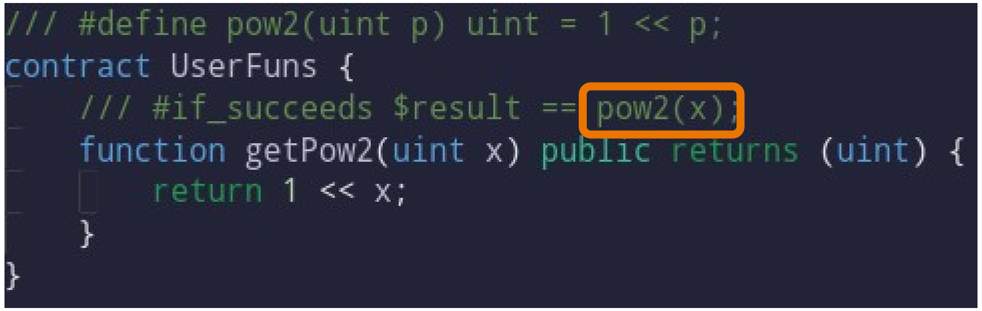
Defined on **contract** docstring

Introduced with #define keyword

Body can be any scribble expression

DILIGENCE Fuzzing

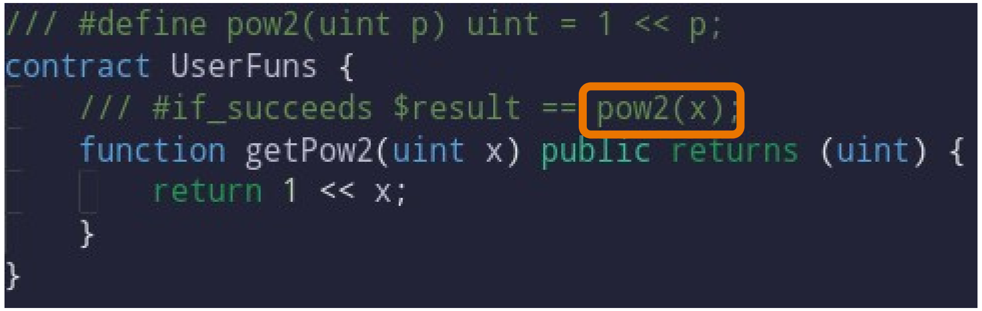
User Functions



```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    /// #if_succeeds $result == pow2(x);
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

Can use it in any annotation in the body of the contract

User Functions



```
/// #define pow2(uint p) uint = 1 << p;
contract UserFuns {
    /// #if_succeeds $result == pow2(x);
    function getPow2(uint x) public returns (uint) {
        return 1 << x;
    }
}
```

Can use it in any annotation in the body of the contract

Can't be used in sub-contracts

Summary

Summary

- Universal Qunatifiers

Summary

- Universal Qunatifiers
- State Variable Annotations

Summary

- Universal Qunatifiers
- State Variable Annotations
- Statement Annotations

Summary

- Universal Qunatifiers
- State Variable Annotations
- Statement Annotations
- User Functions

Homework

Questions

Course Material:
https://github.com/ConsenSys/secureum-diligence-bootcamp/

Other:
Scribble: https://github.com/consensys/scribble
Scribble Docs: https://docs.scribble.codes/
Discord Channel: #carex-diligence-scribble-nov22
Instructors: dimo#1001, wuestholz#3558