Solidity Language Summit

Verifying LLM-powered Code Transformations with Equivalence Checking

John Toman

certora

November 2025

Agenda

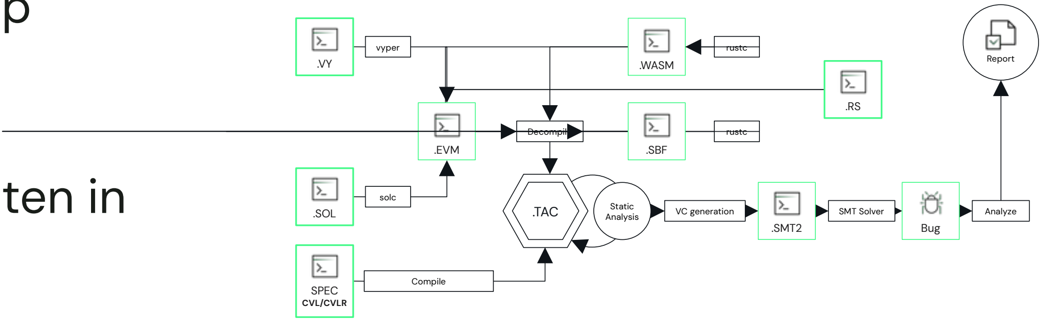
- Background & Motivation
- Concordance: In Brief
- *Defining* Equivalence
- *Checking* Equivalence
- Concordance: The Details

certora

Background & Motivation

A brief digression…

- Certora Prover is our “flagship” technology
- Verifies smart contracts written in Solidity (& others)
- Relies on static analysis of memory, storage, etc.

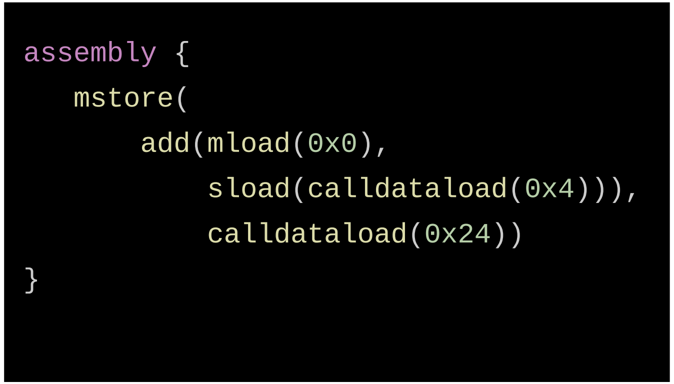


Analysis Kryptonite

- Solidity's frequently (ab)used feature:

  Inline assembly!

- Frequently causes issues with critical static analyses

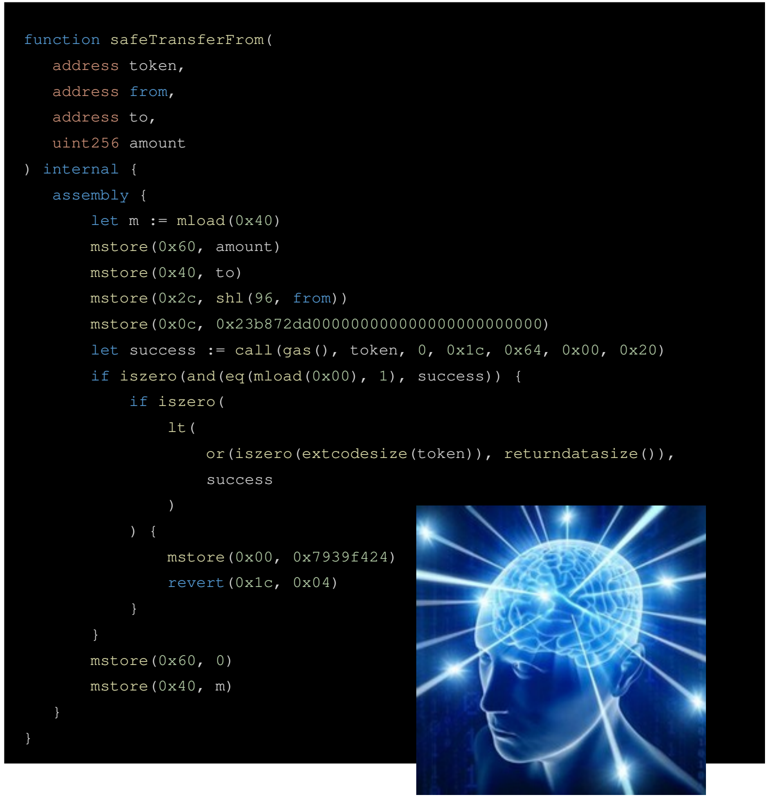


```
assembly {
   mstore(
       add(mload(0x0),
           sload(calldataload(0x4))),
           calldataload(0x24))
}
```

Motivating Example: Solady

- Representative example of Solady code

- Heavily optimized, gas efficient



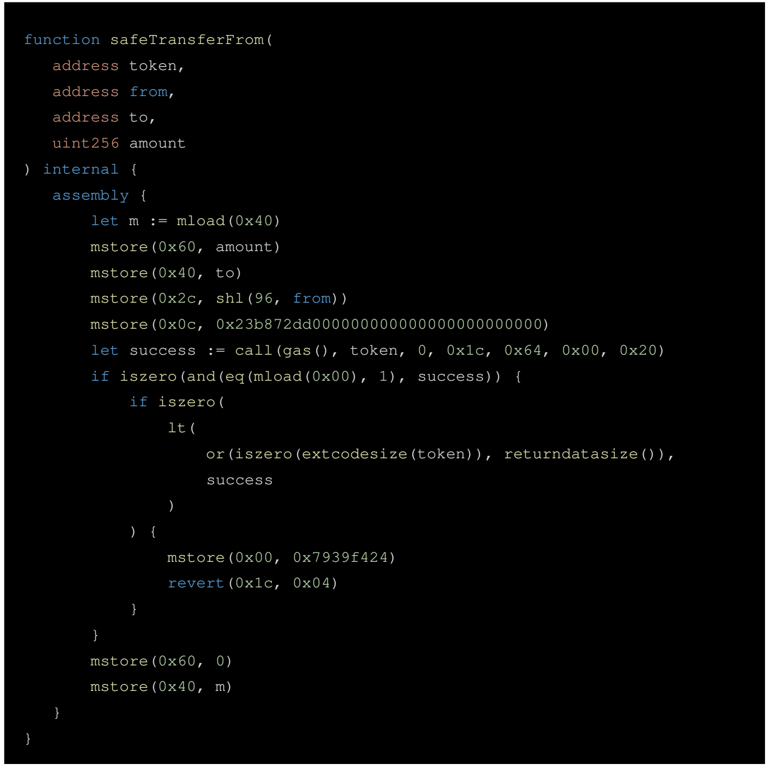
```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd000000000000000000000000)
       let success := call(gas(), token, 0, 0x1c, 0x64, 0x00, 0x20)
       if iszero(and(eq(mload(0x00), 1), success)) {
           if iszero(
               lt(
                   or(iszero(extcodesize(token)), returndatasize()),
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

certora

Motivating Example: Solady (cont)

- Representative of a lot of modern DeFi code

- Absolutely *kills* some of our static analyses...



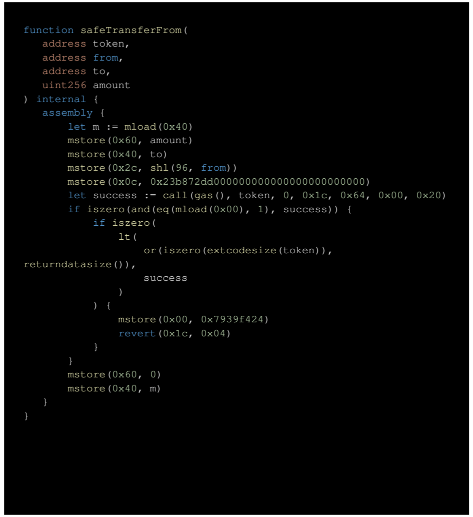
```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd000000000000000000000000)
       let success := call(gas(), token, 0, 0x1c, 0x64, 0x00, 0x20)
       if iszero(and(eq(mload(0x00), 1), success)) {
           if iszero(
               lt(
                   or(iszero(extcodesize(token)), returndatasize()),
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

certora

Motivation

Can we automatically generate “equivalent” implementations?

(for auditing, code review, analysis …)



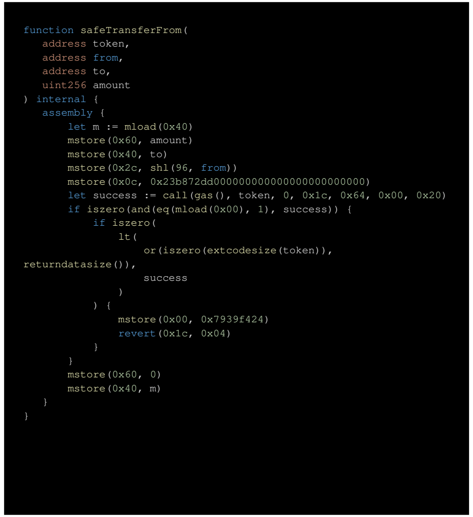
```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd000000000000000000000000)
       let success := call(gas(), token, 0, 0x1c, 0x64, 0x00, 0x20)
       if iszero(and(eq(mload(0x00), 1), success)) {
           if iszero(
               lt(
                   or(iszero(extcodesize(token)),
returndatasize()),
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

?

certora

Motivation

Can we automatically generate “equivalent” implementations?



```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd000000000000000000000000)
       let success := call(gas(), token, 0, 0x1c, 0x64, 0x00, 0x20)
       if iszero(and(eq(mload(0x00), 1), success)) {
           if iszero(
               lt(
                   or(iszero(extcodesize(token)),
returndatasize()),
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

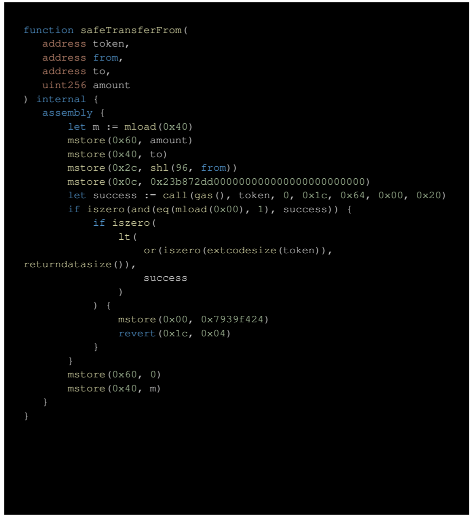
?

certora

Concordance: In Brief

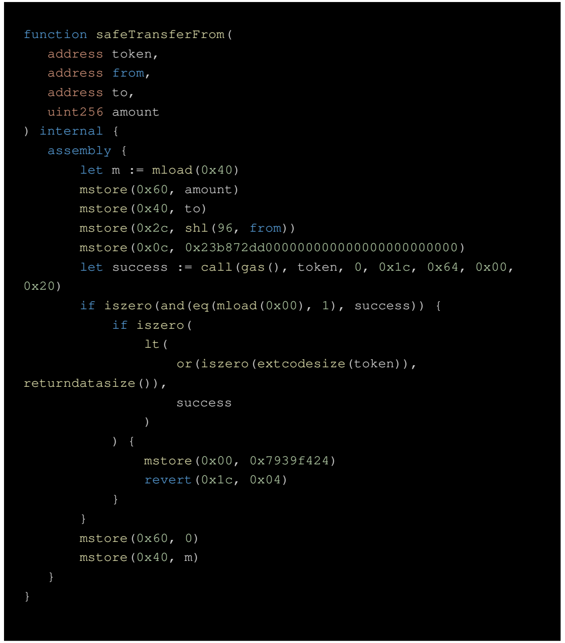
Motivation

Can we automatically generate “equivalent” implementations?



```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd000000000000000000000000)
       let success := call(gas(), token, 0, 0x1c, 0x64, 0x00, 0x20)
       if iszero(and(eq(mload(0x00), 1), success)) {
           if iszero(
               lt(
                   or(iszero(extcodesize(token)),
returndatasize()),
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

“AI powered”

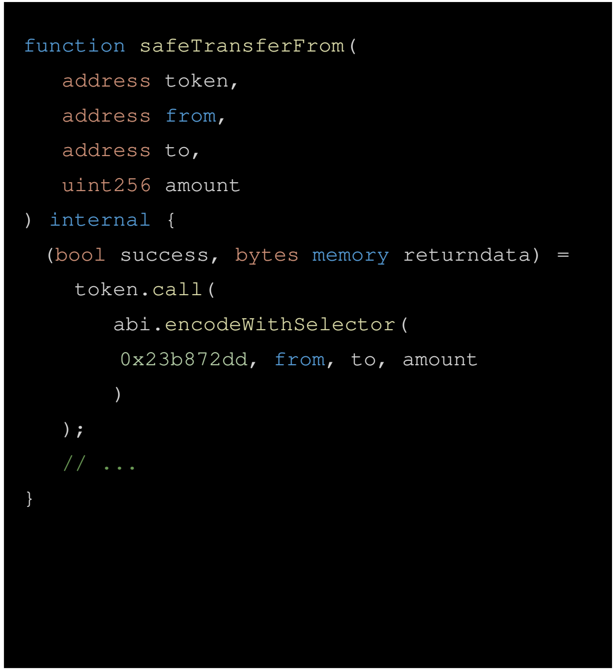


```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd000000000000000000000000)
       let success := call(gas(), token, 0, 0x1c, 0x64, 0x00,
0x20)
       if iszero(and(eq(mload(0x00), 1), success)) {
           if iszero(
               lt(
                   or(iszero(extcodesize(token)),
returndatasize()),
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

Any available LLM

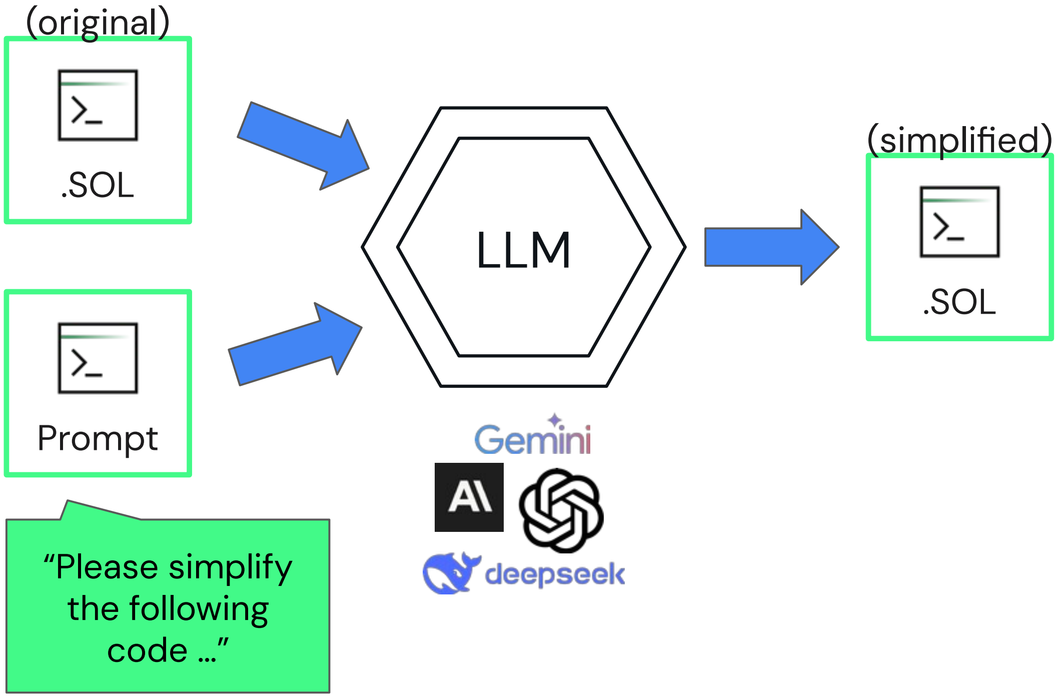
“Please rewrite the following code to use standard solidity instead of inline assembly. Make sure it’s equivalent …”

Is it this simple?



```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
  (bool success, bytes memory returndata) =
    token.call(
       abi.encodeWithSelector(
        0x23b872dd, from, to, amount
       )
   );
   // ...
}
```

Concordance: “AI powered”



(original)

.SOL

Prompt

“Please simplify the following code …”

LLM

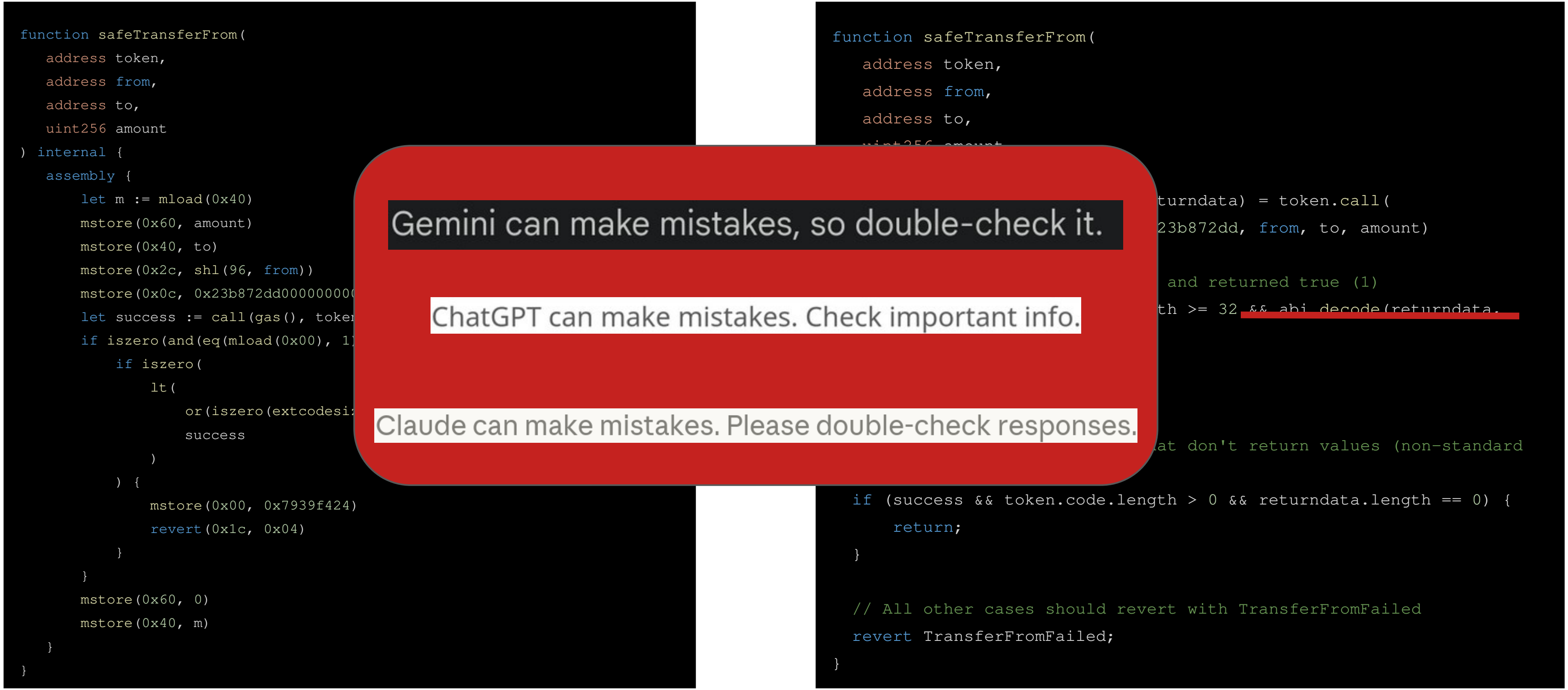
(simplified)

.SOL

Is it this simple?

“AI powered” … mistakes

Proposed rewrite by Claude



```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd00000000(
       let success := call(gas(), toker
       if iszero(and(eq(mload(0x00), 1)
           if iszero(
               lt(
                   or(iszero(extcodesiz
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

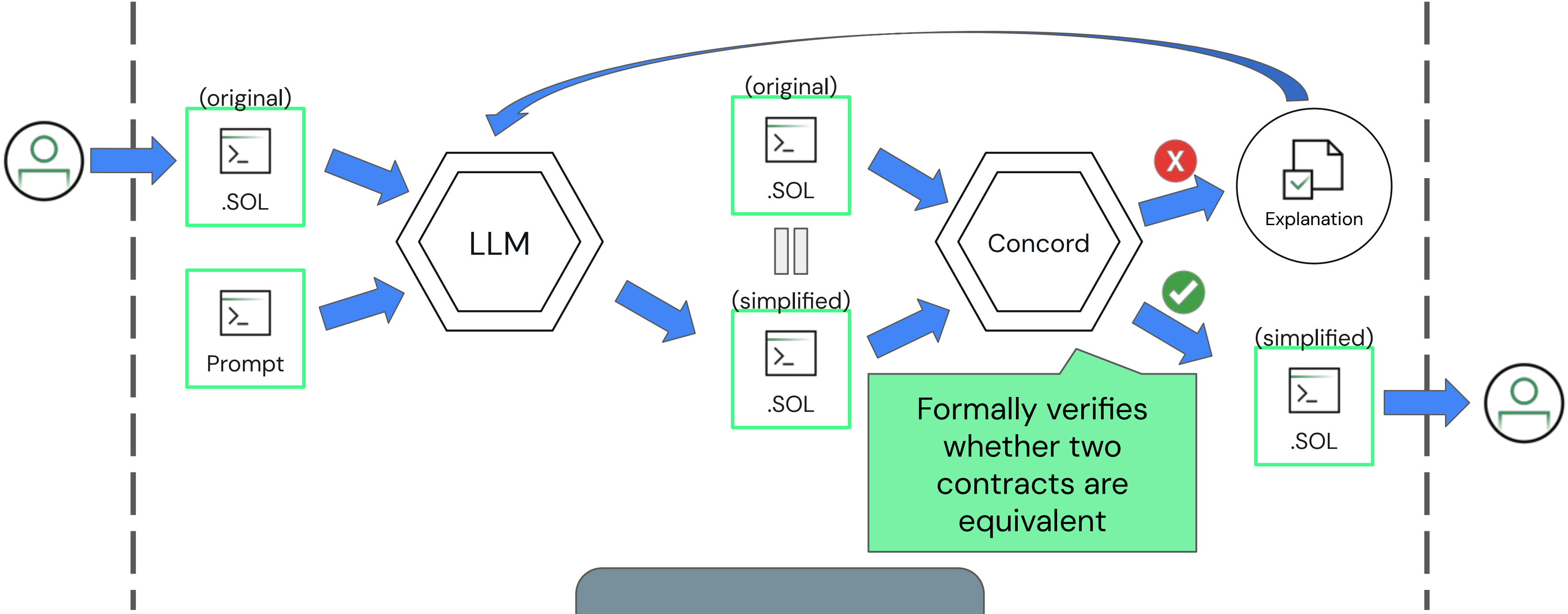
Gemini can make mistakes, so double-check it.

ChatGPT can make mistakes. Check important info.

Claude can make mistakes. Please double-check responses.

```
function safeTransferFrom(
   address token,
   address from,
   address to,
                        turndata) = token.call(
                        23b872dd, from, to, amount)

                         and returned true (1)
                        th >= 32 ~~&& abi.decode(returndata,~~

                        at don't return values (non-standard

  if (success && token.code.length > 0 && returndata.length == 0) {
      return;
  }

  // All other cases should revert with TransferFromFailed
  revert TransferFromFailed;
}
```

Concordance: AI powered … with guardrails

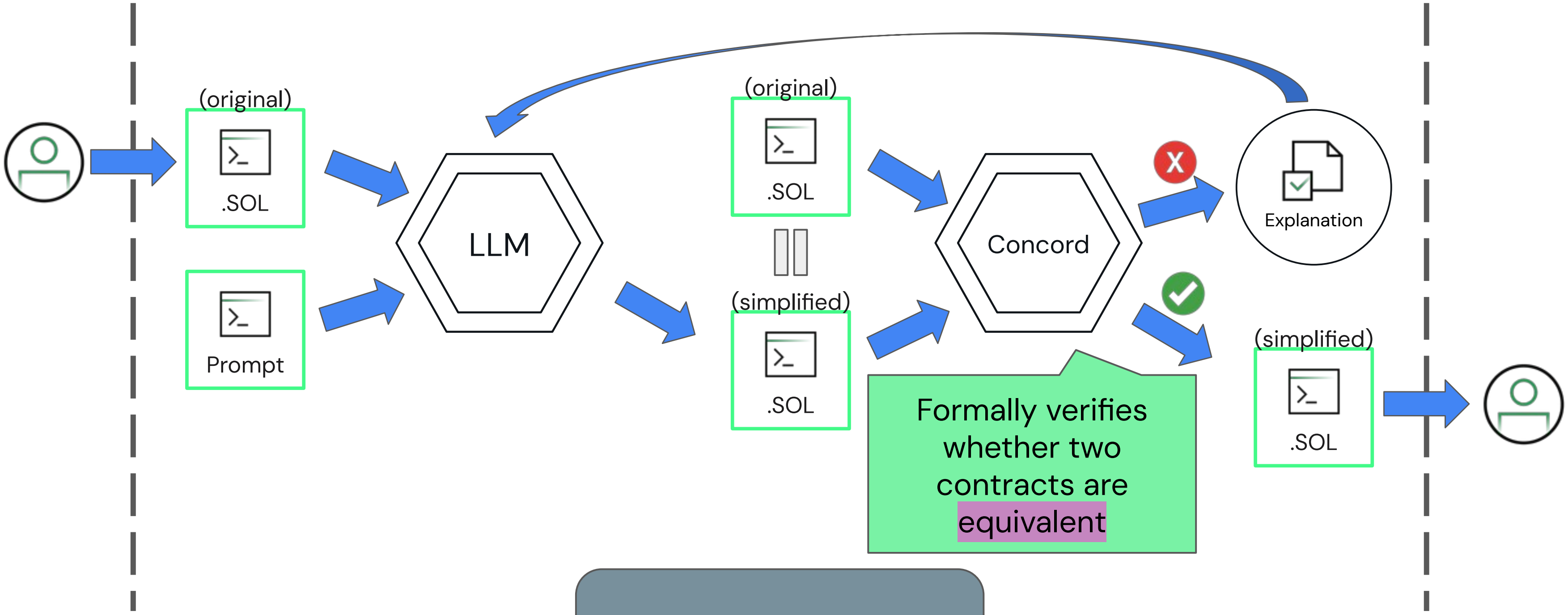


Formally verifies whether two contracts are equivalent

Concordance

certora

Concordance: AI powered … with guardrails



Formally verifies whether two contracts are equivalent

Concordance

certora

Defining Equivalence

Intuitive Definition

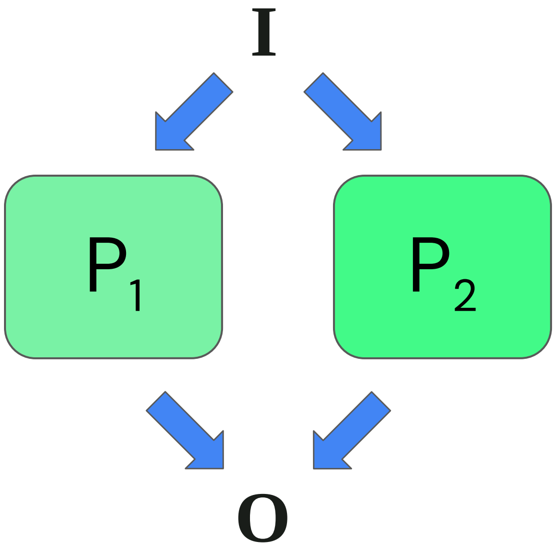
Two programs are "equivalent" if they "do the same thing" on "the same inputs".

Intuitive Definition

Two programs are “equivalent” if they “do the same thing” on “the same inputs”.

Intuitive Definition

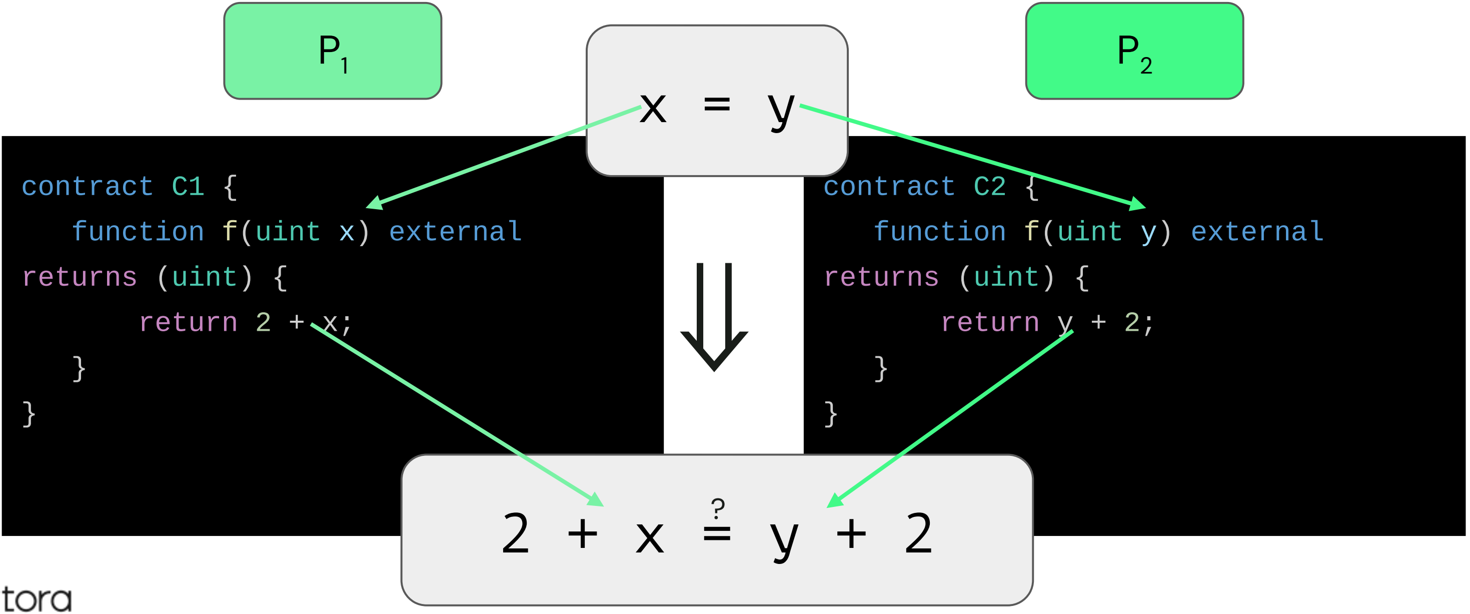
Two programs are “equivalent” if they “produce the same outputs” on “the same inputs”.



certora

Formalizing the Definition

Two programs are “equivalent” if they “produce the same outputs” on “the same inputs”.



P_1

$x = y$

P_2

```
contract C1 {
   function f(uint x) external
returns (uint) {
        return 2 + x;
   }
}
```

$\Downarrow$

```
contract C2 {
   function f(uint y) external
returns (uint) {
        return y + 2;
   }
}
```

$$2 + x \overset{?}{=} y + 2$$

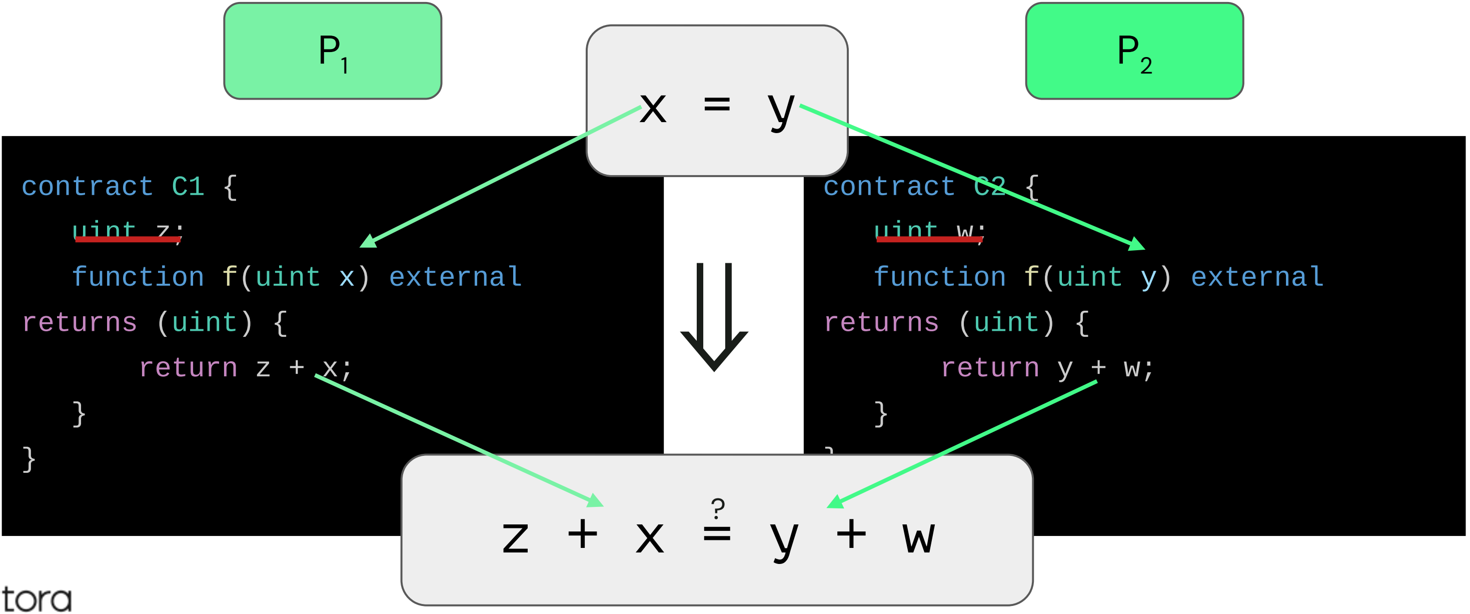
Formalizing the Definition (cont)

“the same inputs“ $\overset{\text{def}}{=}$ “the same `calldata` buffers”

“produce the same outputs“ $\overset{\text{def}}{=}$ “produce the same `returndata` buffers”

Formalizing the Definition (cont)

Two programs are “equivalent” if they “produce the same `returndata` buffers” on “the same `calldata` buffers”.



P_1

$x = y$

P_2

```
contract C1 {
   uint z;
   function f(uint x) external
returns (uint) {
        return z + x;
   }
}
```

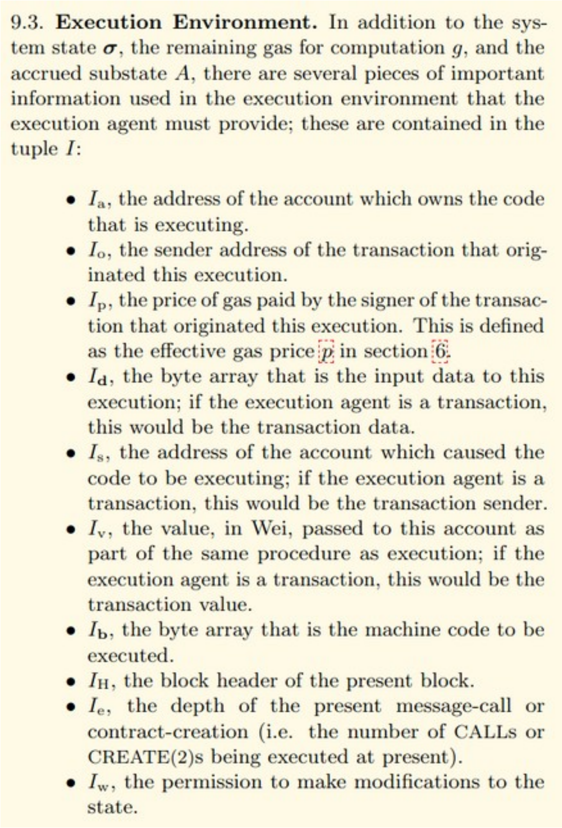
$\Downarrow$

```
contract C2 {
   uint w;
   function f(uint y) external
returns (uint) {
        return y + w;
   }
}
```

$z + x \overset{?}{=} y + w$

Expanding inputs

Smart contracts “implicitly” receive their current storage, and the state of the rest of the Ethereum blockchain (codehashes, nonces, balances…) as input!



9.3. **Execution Environment.** In addition to the system state $\boldsymbol{\sigma}$, the remaining gas for computation g, and the accrued substate A, there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple I:

- I_{a}, the address of the account which owns the code that is executing.
- I_{o}, the sender address of the transaction that originated this execution.
- I_{p}, the price of gas paid by the signer of the transaction that originated this execution. This is defined as the effective gas price p in section 6.
- $I_{\mathbf{d}}$, the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- I_{s}, the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender.
- I_{v}, the value, in Wei, passed to this account as part of the same procedure as execution; if the execution agent is a transaction, this would be the transaction value.
- $I_{\mathbf{b}}$, the byte array that is the machine code to be executed.
- I_{H}, the block header of the present block.
- I_{e}, the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATE(2)s being executed at present).
- I_{w}, the permission to make modifications to the state.

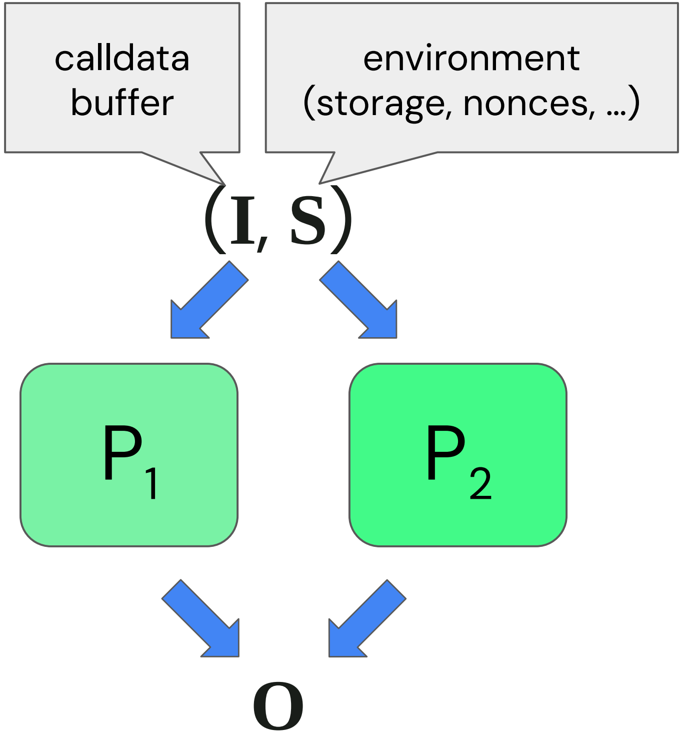
Formalizing the Definition (cont)

“the same inputs“ $\stackrel{\text{def}}{=}$ “the same `calldata` buffers AND environment”

“produce the same outputs“ $\stackrel{\text{def}}{=}$ “produce the same `returndata` buffers”

Formal Definition (?)

Two programs are "equivalent" if they "produce the same `returndata` buffers" on "the same `calldata` buffers AND environment".



certora

More outputs?

Two programs are “equivalent” if they “produce the same `returndata` buffers” on “the same `calldata` buffers AND environment”.

P$_1$

```
contract C1 {
   uint public z;
   function f(uint x) external {
       z = z + x;
   }
}
```

≠

P$_2$

```
contract C2 {
   uint public z;
   function f(uint y) external {
       z = y + z + 1;
   }
}
```

Why?

More Intuition

An external actor shouldn’t be able to “tell the difference” between two equivalent implementations.

Side Effects

```
// c1.z    = 6 =    // c2.z

c1.f(5);            c2.f(5);

c1.z() = 11≠12 = c2.z()
```

Defining Side Effects

- Mutating Storage
- Emitting a log/event
- Calling another contract
  - balance transfers
  - contract creation
- [Self-Destruct]

- Pushing/popping the stack
- Mutating memory

Why not?

Defining Side Effects

An external actor shouldn't be able to "tell the difference" between two equivalent implementations.

Stack and memory are not visible to an external actor!

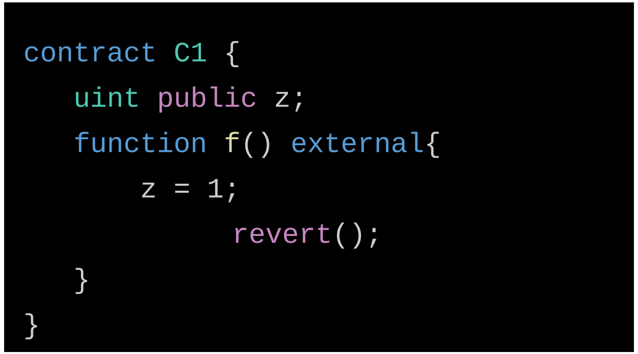
"If the stack gets popped in the forest and no one is around to hear it…"

Gas, Gas, Gas

- We *also* exclude gas!
- Definitely an observable side effect!
- Would overly restrict space of equivalent programs

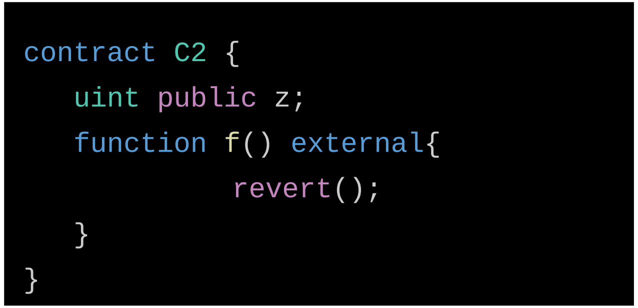
Subtleties

- If a function reverts, it has no side effects (by definition)



```
contract C1 {
   uint public z;
   function f() external{
       z = 1;
            revert();
   }
}
```

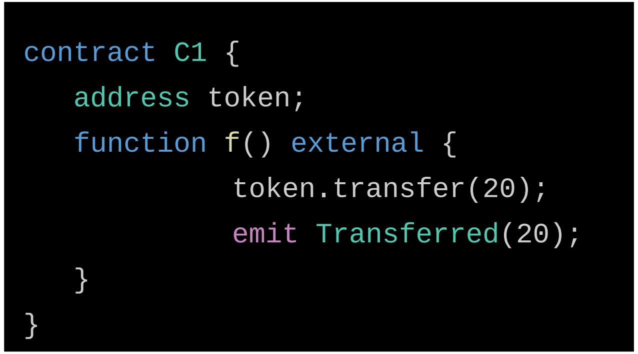
=



```
contract C2 {
   uint public z;
   function f() external{
            revert();
   }
}
```

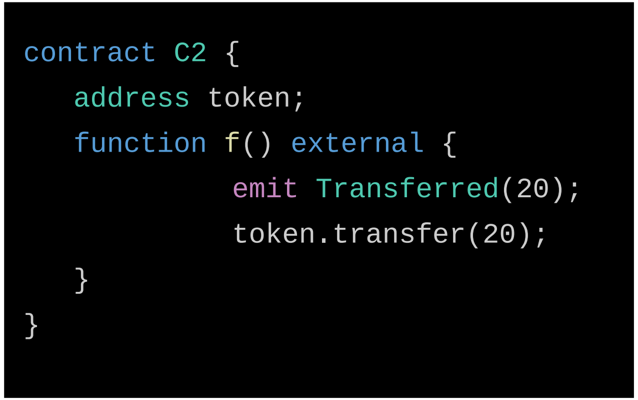
Subtleties

- If a function reverts, it has no side effects (by definition)
- Logs are not observable on-chain, so the interleaving of logs/external calls is not important



```
contract C1 {
   address token;
   function f() external {
            token.transfer(20);
            emit Transferred(20);
   }
}
```

=



```
contract C2 {
   address token;
   function f() external {
            emit Transferred(20);
            token.transfer(20);
   }
}
```

Subtleties

- If a function reverts, it has no side effects (by definition)
- Logs are not observable on-chain, so the interleaving of logs/external calls is not important
- We only care about the *final* state of storage, not intermediate states*

```
contract C1 {
   uint public z;
   function f() external{
       z = 1;
       z = 42;
   }
}
```

=

```
contract C2 {
   uint public z;
   function f() external{
       z = 42;
   }
}
```

Other Definitions

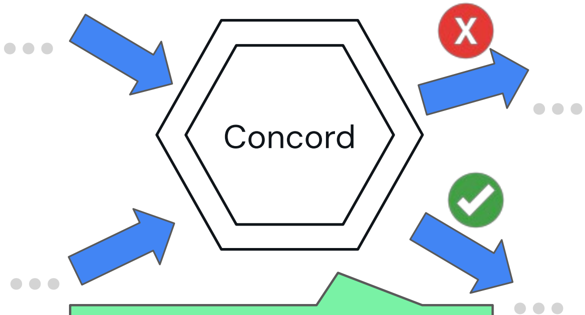
- Other reasonable definitions exist!
- Require strictly *less* gas
- Require identical interleavings of calls/logs
- Equivalent modulo valid calldata
- ...

Summary: Equivalence

Two contracts/programs are equivalent if, on the same calldata and in the same environment, they both:

1. Revert with identical `returndata` buffers, OR
2. Return with identical `returndata` buffers and:
    a. Emit the same logs (in the same order)
    b. Make the same external calls (in the same order)
    c. End with equal storages

Next: Formal Verification



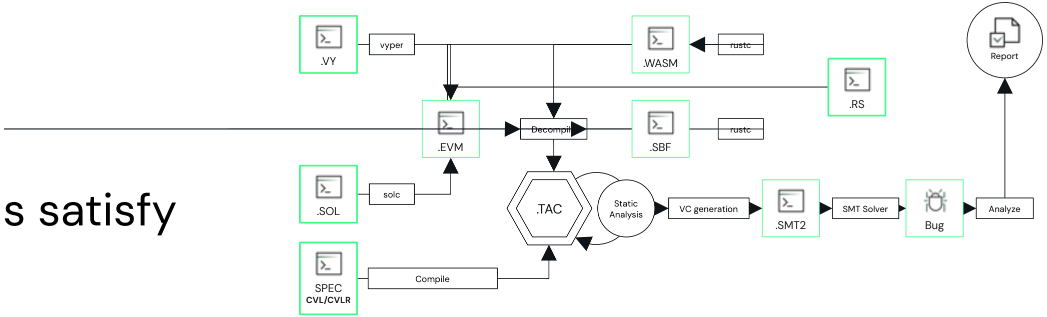
Formally verifies whether two contracts are equivalent

Checking Equivalence

(AKA the densest part of the talk)

Concord: Application of Certora Prover

- “Symbolic reasoning” tool
- Roughly: determine if programs satisfy mathematical formulae
- *Symbolic*: determination reached without running the program, searches infinite state space



Certora Prover

Checking … with Math

Mathematical property:

$$r_1 = r_2 \wedge O_1 = O_2 \wedge \Big((r_1 = 1) \bigvee$$

$$\ldots \leq i \leq n \; c_{1.i} = c_{2.i}) \wedge$$

$$\ldots {}_{2.i}))\Big)$$

same rever statu

same returndat

reverted, OR…

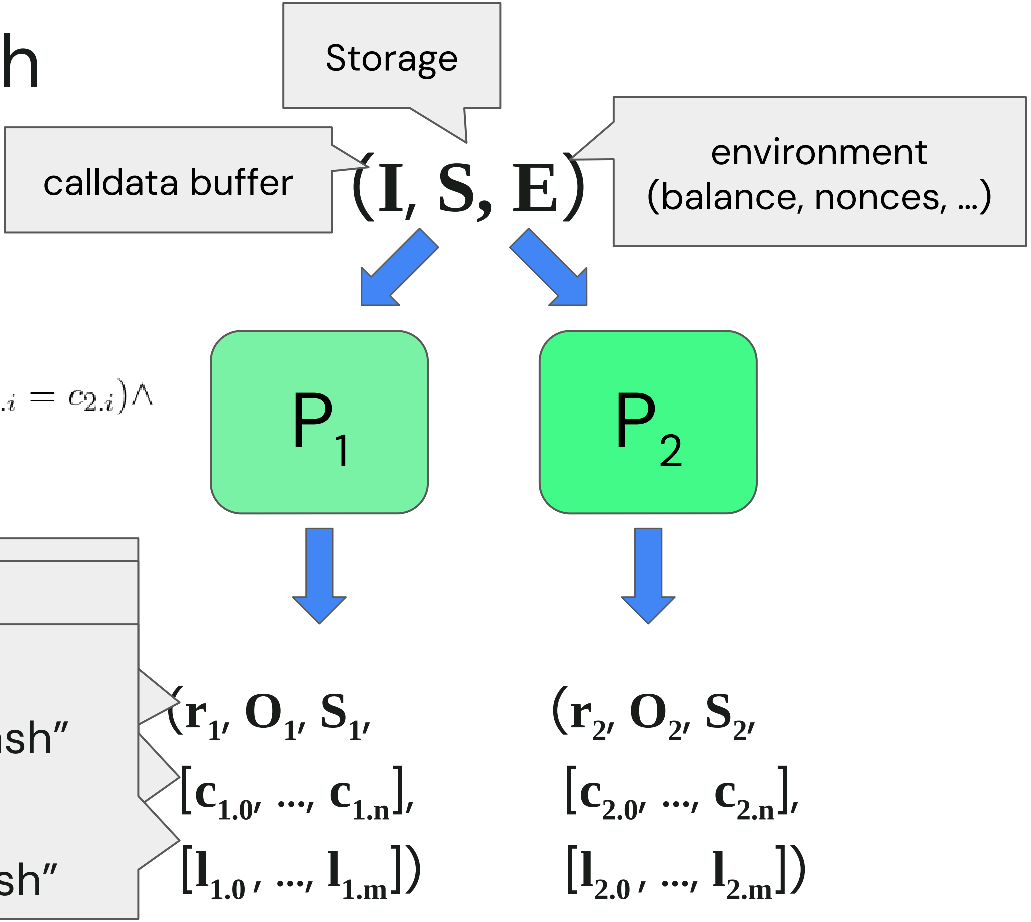
n external calls

m logs emitted

$\mathbf{l}_{1.0}$ = first log “hash”

…

$\mathbf{l}_{1.m}$= last log “hash”



Storage

calldata buffer

environment (balance, nonces, …)

$$(\mathbf{I}, \mathbf{S}, \mathbf{E})$$

P_1

P_2

$$(\mathbf{r}_1, \mathbf{O}_1, \mathbf{S}_1, [\mathbf{c}_{1.0}, \ldots, \mathbf{c}_{1.n}], [\mathbf{l}_{1.0}, \ldots, \mathbf{l}_{1.m}])$$

$$(\mathbf{r}_2, \mathbf{O}_2, \mathbf{S}_2, [\mathbf{c}_{2.0}, \ldots, \mathbf{c}_{2.n}], [\mathbf{l}_{2.0}, \ldots, \mathbf{l}_{2.m}])$$

certora

Checking … with Math

Mathematical property:

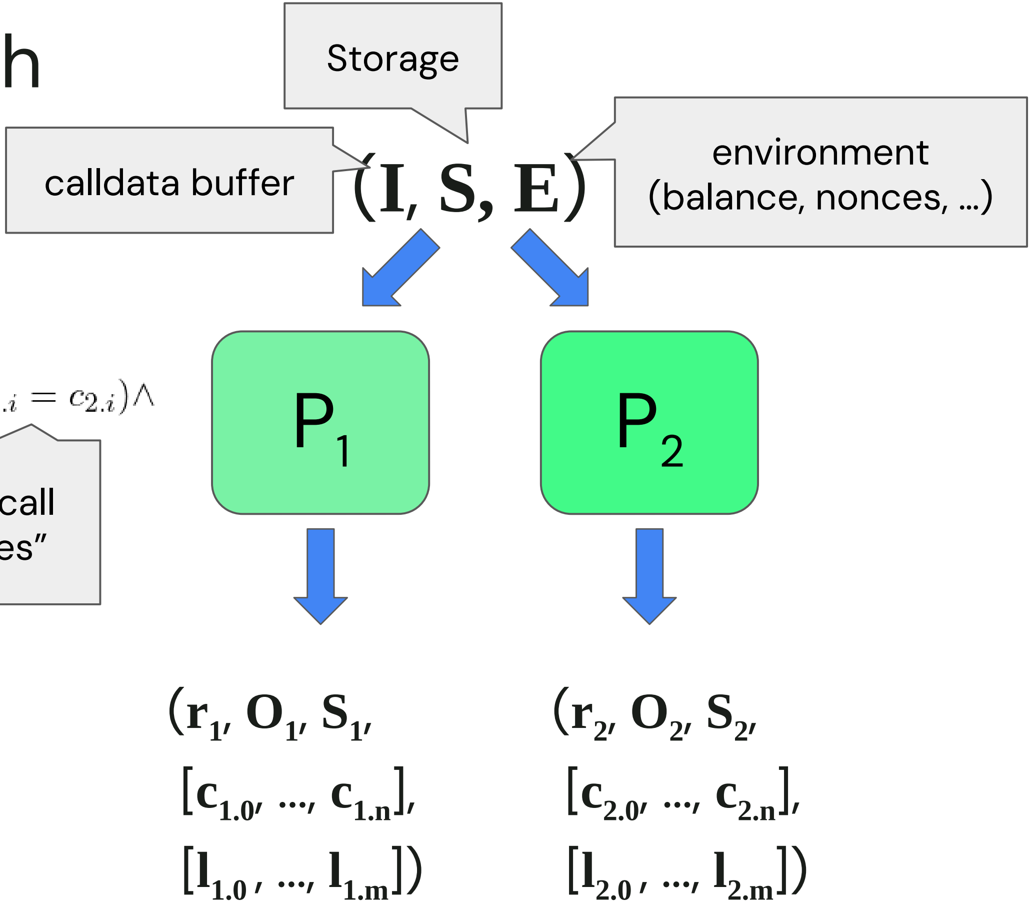
$$r_1 = r_2 \wedge O_1 = O_2 \wedge \Big((r_1 = 1) \bigvee$$

$$(S_1 = S_2 \wedge (\wedge_{0 \leq i \leq n}\ c_{1.i} = c_{2.i}) \wedge$$

$$l_{1.i}$$

Same storage

Same call “hashes”



$(\mathbf{I}, \mathbf{S}, \mathbf{E})$

$(\mathbf{r}_1, \mathbf{O}_1, \mathbf{S}_1, [\mathbf{c}_{1.0}, \ldots, \mathbf{c}_{1.n}], [\mathbf{l}_{1.0}, \ldots, \mathbf{l}_{1.m}])$

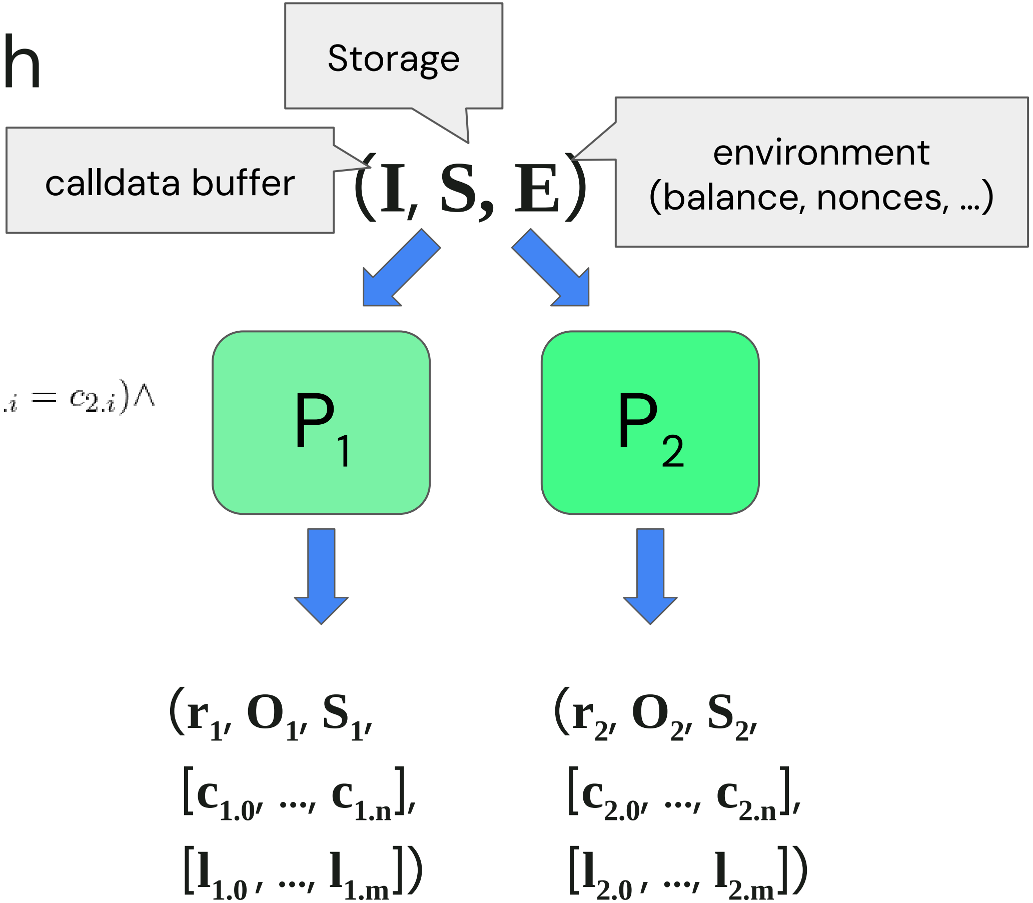
$(\mathbf{r}_2, \mathbf{O}_2, \mathbf{S}_2, [\mathbf{c}_{2.0}, \ldots, \mathbf{c}_{2.n}], [\mathbf{l}_{2.0}, \ldots, \mathbf{l}_{2.m}])$

Checking … with Math

Mathematical property:

$$r_1 = r_2 \wedge O_1 = O_2 \wedge \Big((r_1 = 1) \bigvee \\ (S_1 = S_2 \wedge (\wedge_{0 \leq i \leq n}\, c_{1.i} = c_{2.i}) \wedge \\ (\wedge_{0 \leq i \leq m}\, l_{1.i} = l_{2.i}))\Big)$$

Same log "hashes"



Storage

calldata buffer

environment (balance, nonces, …)

(I, S, E)

$(r_1, O_1, S_1,$ $[c_{1.0}, \ldots, c_{1.n}],$ $[l_{1.0}, \ldots, l_{1.m}])$

$(r_2, O_2, S_2,$ $[c_{2.0}, \ldots, c_{2.n}],$ $[l_{2.0}, \ldots, l_{2.m}])$

“Symbolic” Instrumentation

```
v1 := calldataload(4)
// ...

v2 := call(tgt = r, amt = b, gas = g, inOffs = i, inSize = sz, ...)
// ...

log3(t1, t2, t3, offs = i2, sz = sz2)
if(*) {

   return(i = ret, sz = sz3)
} else {

   revert(i = rev, sz = sz4)
}
```

“Symbolic” Instrumentation

Include hash of sent buffer

```
v1 := calldatal
// ...
c_i := hash(r, b, hash(mem[i:sz]), hash(S))
v2 := call(tgt = r, amt = b, gas                    inSize = sz, ...)
// .
l_j :            3, hash(mem[i
log             s = i2, sz =
if(
                 (mem[ret:sz3])
                sz
} e
    r := 1, 0 := hash(
    revert(i = rev, sz
}
```

Effects of call soundly modeled

function

“hash”* of current storage (re-entrancy)

Most of the engineering

certora

“Symbolic” Instrumentation

```
l := [], c = [], call_idx = 0, log_idx = 0
v1 := calldataload(4)
// ...
c[call_idx++] := hash(r, b, hash(mem[i:sz]), hash(S))
v2 := call(tgt = r, amt = b, gas = g, inOffs = i, inSize = sz, ...)
// ...
l[log_idx++] := hash(t1, t2, t3, hash(mem[i2:sz2]))
log3(t1, t2, t3, offs = i2, sz = sz2)
if(*) {
   r := 0, O := hash(mem[ret:sz3])
   return(i = ret, sz = sz3)
} else {
   r := 1, O := hash(mem[rev:sz4])
   revert(i = rev, sz = sz4)
}
```

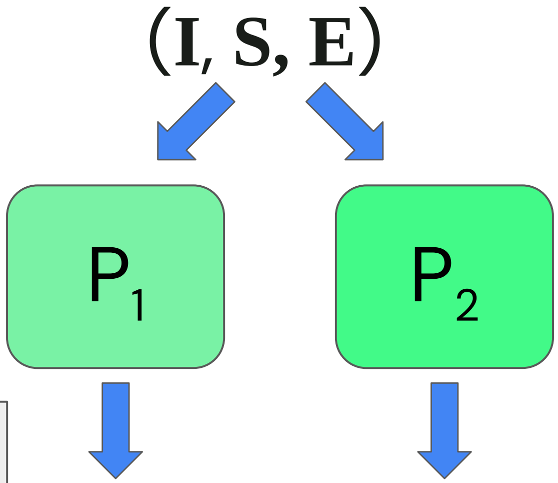
Checking with Instrumentation

Mathematical property:

$$r_1 = r_2 \wedge O_1 = O_2 \wedge \left(r_1 = 1 \bigvee \left(S_1 = S_2 \wedge c_1 = c_2 \wedge l_1 = l_2\right)\right)$$

How do you compare “all of storage”?

How do you symbolically compare lists?



O_1, S_1,
c_1=[$c_{1.0}$, …, $c_{1.n}$],
l_1=[$l_{1.0}$, …, $l_{1.m}$])

(r_2, O_2, S_2,
c_2=[$c_{2.0}$, …, $c_{2.n}$],
l_2=[$l_{2.0}$, …, $l_{2.m}$])

certora

A Brief Digression on Solvers

- Fun drinking game: take a drink whenever someone from Certora says “solver”
- But, how are we using them, exactly?

Solvers: What do they do?

- Roughly: take a mathematical formula, determines if it can be “satisfied” (hence “SAT” solver)

$$x + 2 = y$$

$$x = 2$$
$$y = 4$$

- **OR** prove it can never be satisfied

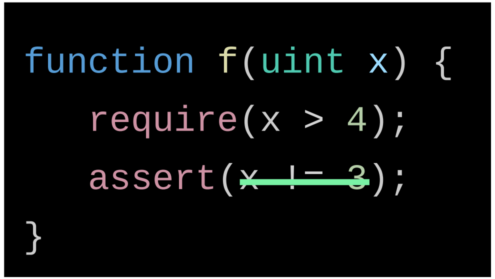
$$x + 1 = x + 2$$

404

Not Found

Solvers: For Verification?

- Ask the SAT solver to “satisfy” the *negation* of your assertion
- If it can, it “found” a way to violate your assertion
- **OR** it has proven the **negation** of your assertion **can’t** happen



```
function f(uint x) {
    require(x > 4);
    assert(~~x != 3~~);
}
```

“Is there some possible value of x…

$\exists x .$

$x > 4 \land x == 3$

… that is greater than 4 and …

… and is **equal** to 3

Nope!

certora

How to compare lists (symbolically)

$\texttt{assert } l_1 = l_2$

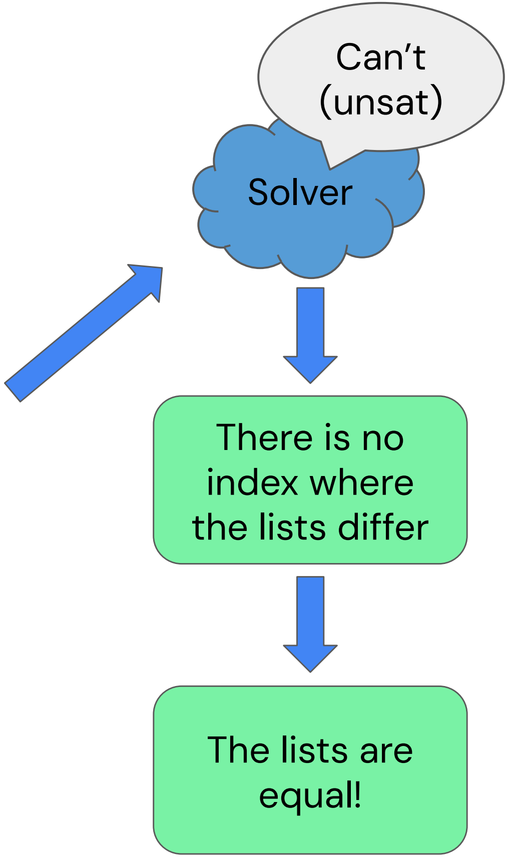
What we want to say “intuitively”

Pick any possible i…

$\texttt{i = *}$

$l_1\texttt{[i] != } l_2\texttt{[i]}$

… where the values at i are different



Solver

Can’t (unsat)

There is no index where the lists differ

The lists are equal!

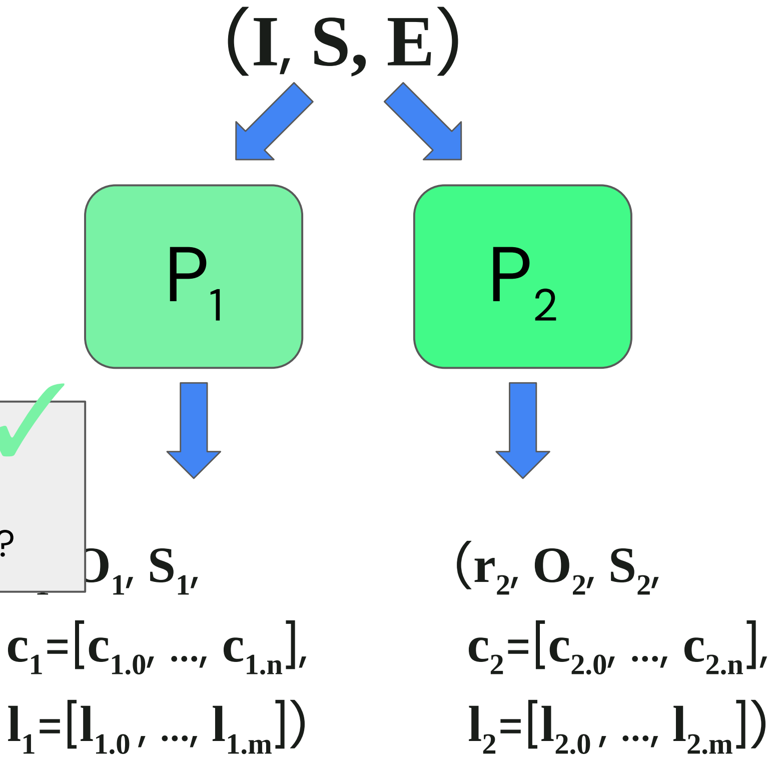
Checking with Instrumentation

Mathematical property:

$$r_1 = r_2 \wedge O_1 = O_2 \wedge \left(r_1 = 1 \bigvee \left(S_1 = S_2 \wedge c_1 = c_2 \wedge l_1 = l_2\right)\right)$$

How do you compare “all of storage”? ✓

How do you symbolically compare lists? ✓



(I, S, E)
P_1
P_2
$O_1, S_1,$
$c_1=[c_{1.0}, \dots, c_{1.n}],$
$l_1=[l_{1.0}, \dots, l_{1.m}])$
$(r_2, O_2, S_2,$
$c_2=[c_{2.0}, \dots, c_{2.n}],$
$l_2=[l_{2.0}, \dots, l_{2.m}])$

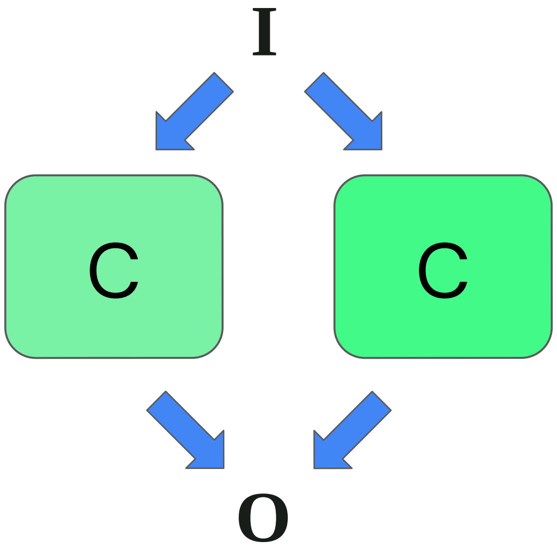
“Symbolic” Instrumentation

```
l := [], c = [], call_idx = 0, log_idx = 0
v1 := calldataload(4)
// ...
c[call_idx++] := hash(r, b, hash(mem[i:sz]), hash(S))
v2 := call(tgt = r, amt = b, gas = g, inOffs =     inSize = sz, ...)
// ...
l[log_idx++] := hash(t1, t2, t3, hash(mem[
log3(t1, t2, t3, offs = i2, sz = sz2)
if(*) {
   r := 0, O := hash(mem[ret:sz3])
   return(i = ret, sz = sz3)
} else {
   r := 1, O := hash(mem[rev:sz4])
   revert(i = rev, sz = sz4)
}
```

“hash”* of current storage (re-entrancy)

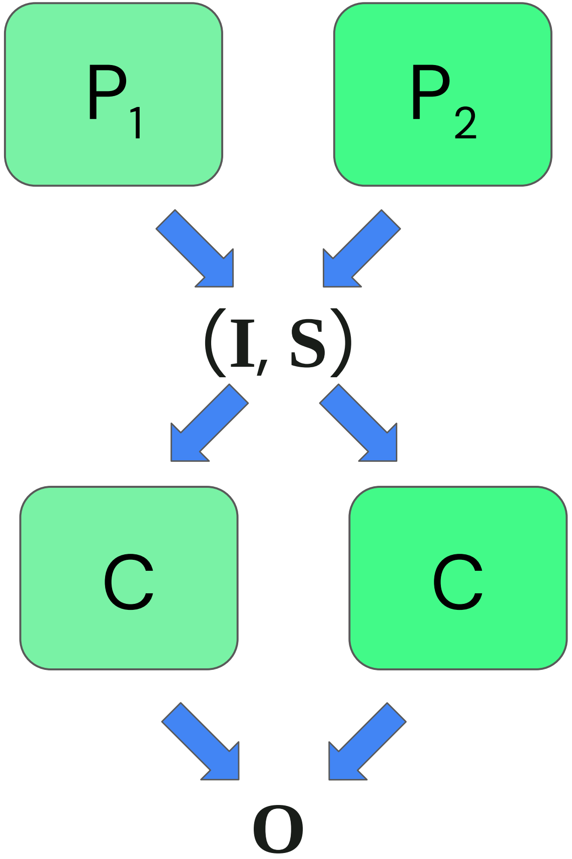
Re-entrancy and External Calls

- We **assume** that external calls are deterministic w.r.t. their inputs
- Calldata, value, etc.
- And “the environment”…



Re-entrancy and External Calls

- We **assume** that external calls are deterministic w.r.t. their inputs
- Calldata, value, etc.
- And "the environment"…
- …which includes the storage of the **caller**, i.e., P_1 and P_2



certora

“Symbolic” Instrumentation

```
c_1 = [], call_idx_1 = 0
// ...
c_1[call_idx_1++] := hash(r1, b1, hash(mem[i:sz]), hash(S_1))
v1 := call(tgt                    as = g1, inOffs = i, inSize = sz, ...)
// ...

c_2 = [], call_i
// ...
c_2[call_idx_2++] := hash(r2, b2, hash(mem[i:sz]), hash(S_2))
v2 := call(t                =                          e = sz, ...)
// ...
assert c_1 = c
```

“Injective”, uninterpreted function

If the output of the functions are the same, the inputs MUST be the same

$$f(x) = f(y) \Rightarrow x = y$$

certora

“Symbolic” Instrumentation

f(x) != f(y) ⇒
x != y

```
c_1 = [], call_idx_1 = 0
// ...
c_1[call_idx_1++] := hash(r1, b1, hash(mem[i:sz]), hash(S_1))
                  r1, amt = b1, gas = g1, inOffs = i, inSize = sz, ...)



                  = 0
// ...
c_2[call_idx_2++] := hash(r2, b2, hash(mem[i:sz]), hash(S_2))
v2 := call(tgt = r2, amt = b2, gas = g2, inOffs = i, inSize = sz, ...)
// ...
assert c_1 = c_2
```

“try to find difference in hashes”

“try to find difference in lists”

“Symbolic” Instrumentation

$$f(x) \mathrel{!=} f(y) \Rightarrow x \mathrel{!=} y$$

```
c_1 = [], call_idx_1 = 0
// ...
c_1[call_idx_1++] := hash(r1, b1, hash(mem[i:sz]), hash(S_1))
v1                   amt = b1, gas = g1, inOffs = i, inSize = sz, ...)
//


c_2 =
// ...
c_2[call_idx_2++] := hash(r2, b2, hash(mem[i:sz]), hash(S_2))
v2 := call(tgt = r2, amt = b2, gas = g2, inOffs = i, inSize = sz, ...)
// ...
assert c_1 = c_2
```

“try to find difference in hash arguments”

“try to find difference in lists”

“Symbolic” Instrumentation

$$f(x) \mathrel{!=} f(y) \Rightarrow x \mathrel{!=} y$$

```
c1 = [], call_idx1 = 0
// ...
c1[call_idx1++] := hash(r1, b1, hash(mem[i:sz]), hash(S1))
v1                   amt = b1, gas = g1, inOffs = i, inSize = sz, ...)
//


c2 =
// ...
c2[call_idx2++] := hash(r2, b2, hash(mem[i:sz]), hash(S2))
v2 := call(tgt = r2, amt = b2, gas = g2, inOffs = i, inSize = sz, ...)
// ...
assert c1 = c2
```

“try to find difference in hash arguments”

Find difference in storage…
… which is a list

“try to find difference in lists”

certora

“Symbolic” Instrumentation

```
c1 = [], call_idx1 = 0
// ...
repr1 = S1[f(call_idx1)]
c1[call_idx1++] := hash(r1                                  , hash(S1))
v1 := call(tgt = r1, amt =                                s = i, inSize = sz, ...)
// ...

c2 = [], call_idx2 = 0
// ...
repr2 = S2[f(call_idx2)]
c2[call_idx2++] := hash(r2, b2, hash(mem[i:sz]), hash(S2))
v2 := call(tgt = r2, amt = b2, gas = g2, inOffs = i, inSize = sz, ...)
// ...
assert c1 = c2
```

Pick an arbitrary index, but that index is the same at both calls

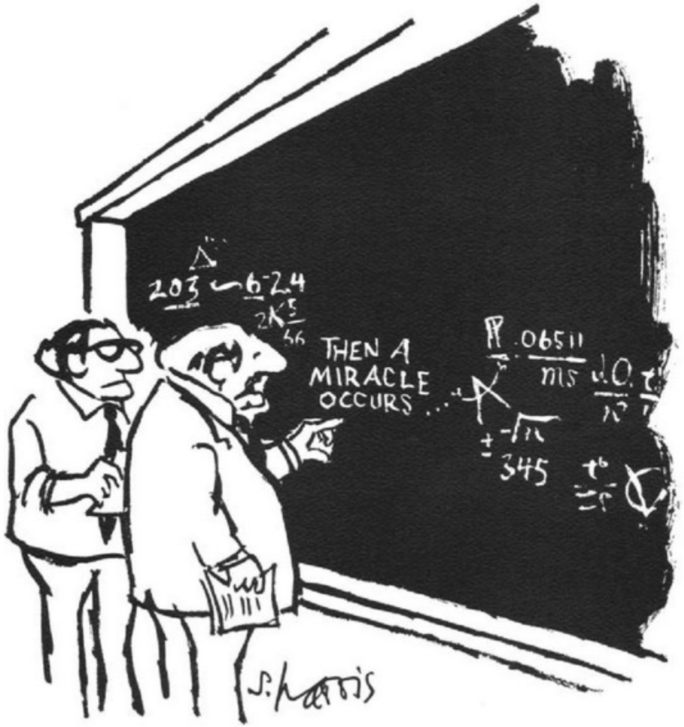
certora

“Symbolic” Instrumentation

```
c1 = [], call_idx1 = 0
// ...
repr1 = S1[f(call_idx1)]
c1[call_idx1++] := hash(r1, b1, hash(mem[i:sz]), repr1)
v1 := call(tgt = r1, amt = b1, gas = g1, inOffs = i, inSize = sz, ...)
// ...

c2 = [], call_idx2 = 0
// ...
repr2 = S2[f(call_idx2)]
c2[call_idx2++] := hash(r2, b2, hash(mem[i:sz]), repr2)
v2 := call(tgt = r2, amt = b2, gas = g2, inOffs = i, inSize = sz, ...)
// ...
assert c1 = c2
```

Glossed Over...

- Details of buffer hashing
- Model of external calls
- ...



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

(Concord tech report)

Limitations

- Contracts **MUST** use the exact same storage layout
- Must have the same ABI
- Assumes no “reflection” by other actors (extcodesize, extcodehash, gasleft, etc.)

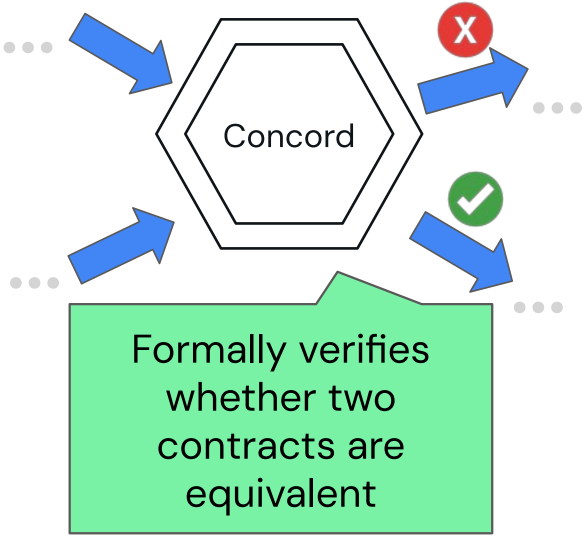
Success Stories

- Concord is crucial to our verified rewriting
- Useful in other contexts: verifying compiler optimizations
- To wit: it uncovered a bug in Vyper’s experimental compilation pipeline

Finding Compiler Bugs With Formal Verification

Summary: Concord

- Instrument code to record side effects
- Use symbolic reasoning of Certora Prover to check results/side effects are always the same
- Next: AI Hype-train



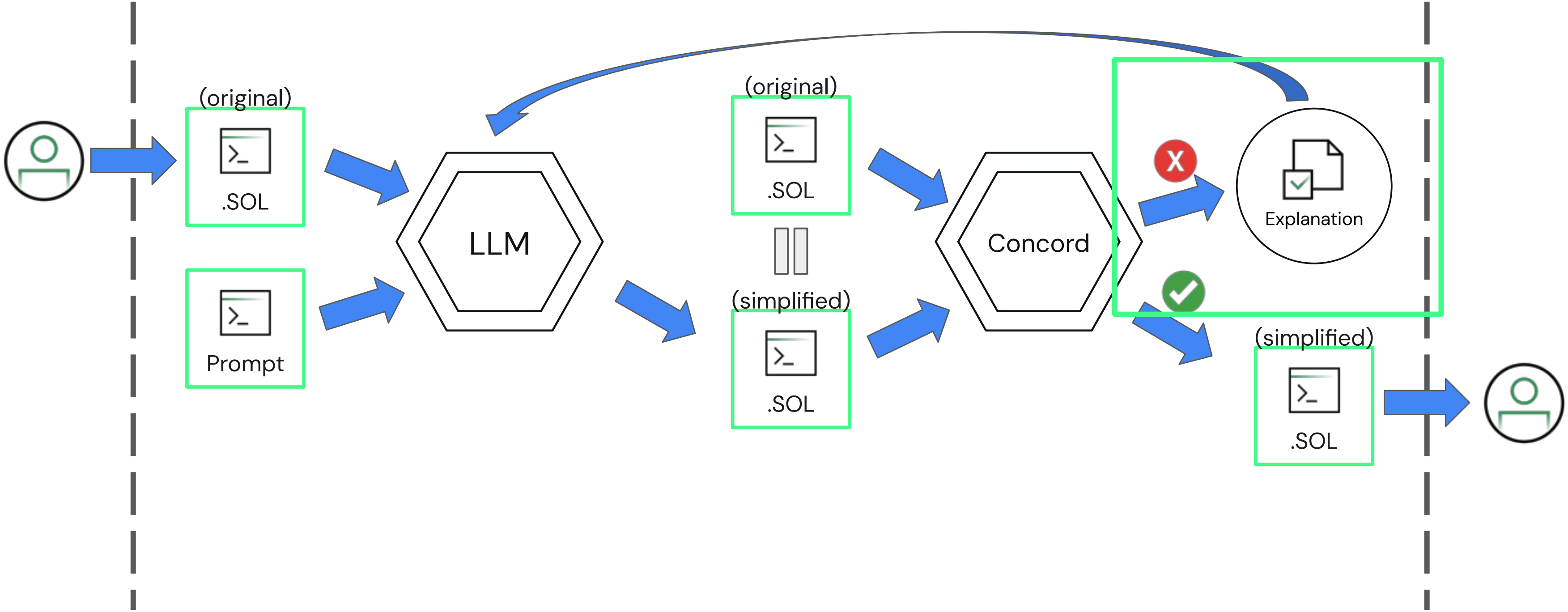
Formally verifies whether two contracts are equivalent

certora

Concordance:
The Details

certora

Concordance: Feedback Loop

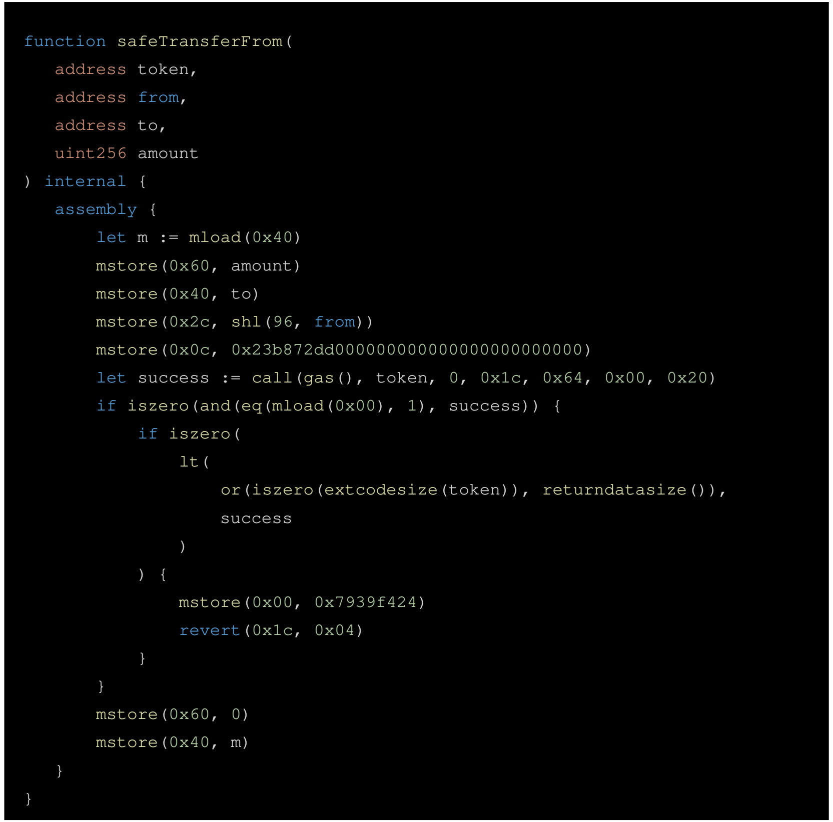


certora

Explaining Explanations

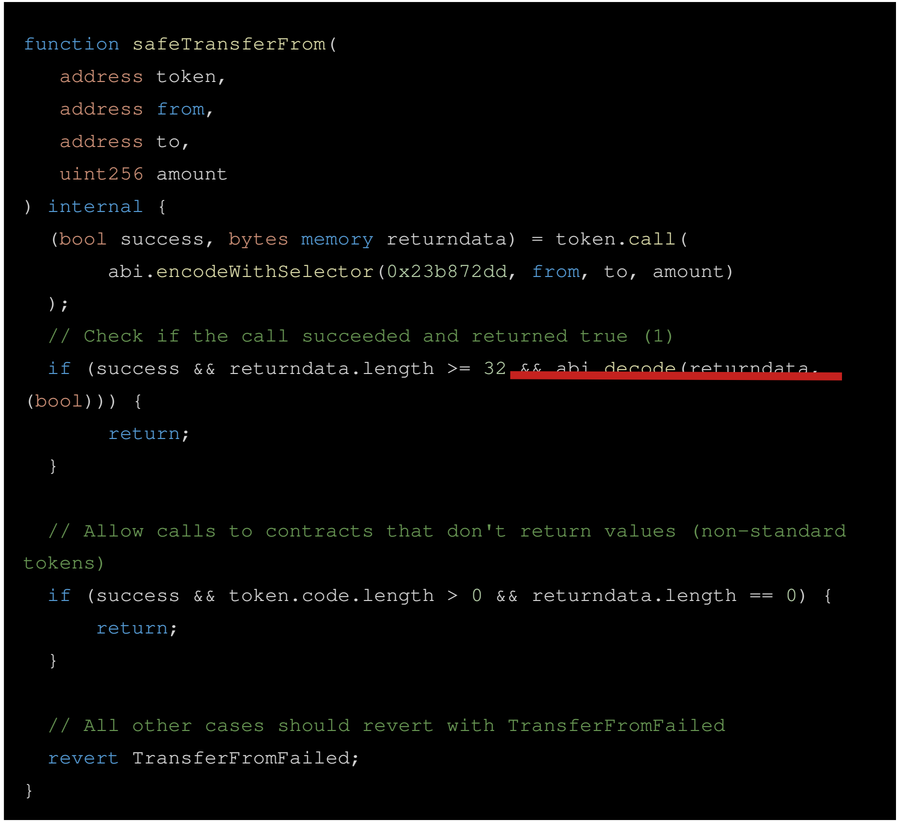
- Certora Prover uses SMT solvers
- These produce a *concrete counterexample* demonstrating a violation
- Analyze counterexample; extract description of events

Explaining Explanations



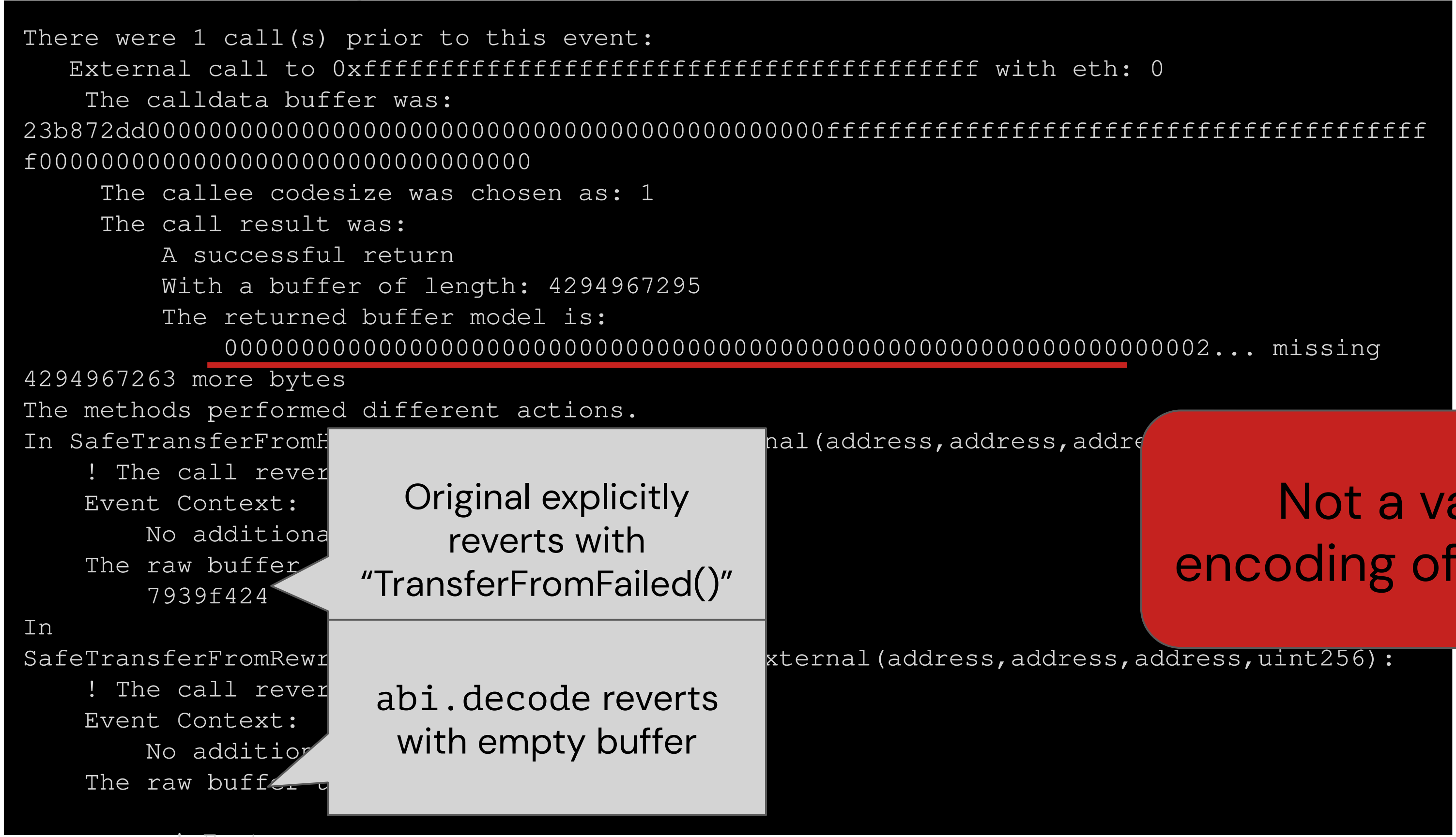
```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
   assembly {
       let m := mload(0x40)
       mstore(0x60, amount)
       mstore(0x40, to)
       mstore(0x2c, shl(96, from))
       mstore(0x0c, 0x23b872dd000000000000000000000000)
       let success := call(gas(), token, 0, 0x1c, 0x64, 0x00, 0x20)
       if iszero(and(eq(mload(0x00), 1), success)) {
           if iszero(
               lt(
                   or(iszero(extcodesize(token)), returndatasize()),
                   success
               )
           ) {
               mstore(0x00, 0x7939f424)
               revert(0x1c, 0x04)
           }
       }
       mstore(0x60, 0)
       mstore(0x40, m)
   }
}
```

≠



```
function safeTransferFrom(
   address token,
   address from,
   address to,
   uint256 amount
) internal {
  (bool success, bytes memory returndata) = token.call(
       abi.encodeWithSelector(0x23b872dd, from, to, amount)
  );
  // Check if the call succeeded and returned true (1)
  if (success && returndata.length >= 32 ~~&& abi.decode(returndata,~~
(bool))) {
       return;
  }

  // Allow calls to contracts that don't return values (non-standard
tokens)
  if (success && token.code.length > 0 && returndata.length == 0) {
      return;
  }

  // All other cases should revert with TransferFromFailed
  revert TransferFromFailed;
}
```

Explaining Explanations



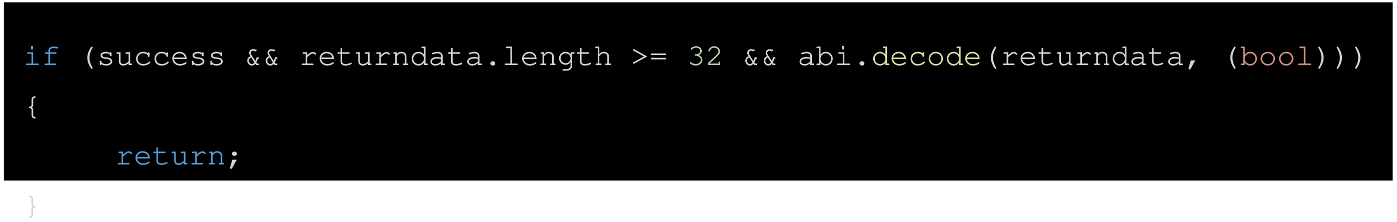
```
There were 1 call(s) prior to this event:
   External call to 0xffffffffffffffffffffffffffffffffffffffff with eth: 0
    The calldata buffer was:
23b872dd00000000000000000000000000000000000000000000fffffffffffffffffffffffffffffffffffffff
f00000000000000000000000000000000
     The callee codesize was chosen as: 1
     The call result was:
         A successful return
         With a buffer of length: 4294967295
         The returned buffer model is:
             0000000000000000000000000000000000000000000000000000000000000002... missing
4294967263 more bytes
The methods performed different actions.
In SafeTransferFromH                            nal(address,address,addre
    ! The call rever
    Event Context:
        No additiona
    The raw buffer
        7939f424
In
SafeTransferFromRewr                            xternal(address,address,address,uint256):
    ! The call rever
    Event Context:
        No additior
    The raw buff
          ! Empty
```

Original explicitly reverts with “TransferFromFailed()”

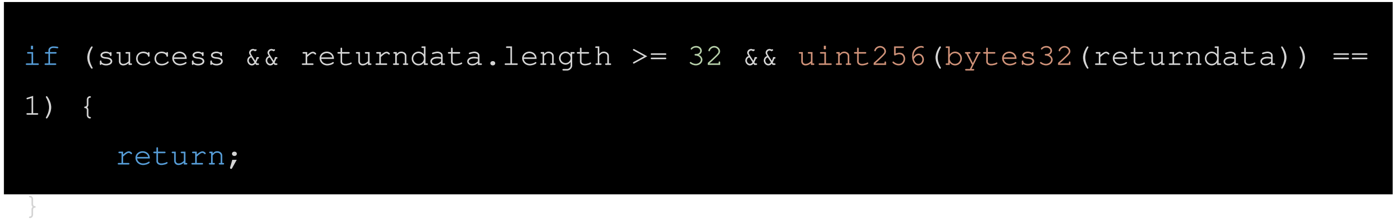
abi.decode reverts with empty buffer

Not a valid encoding of a bool!

LLM Fix

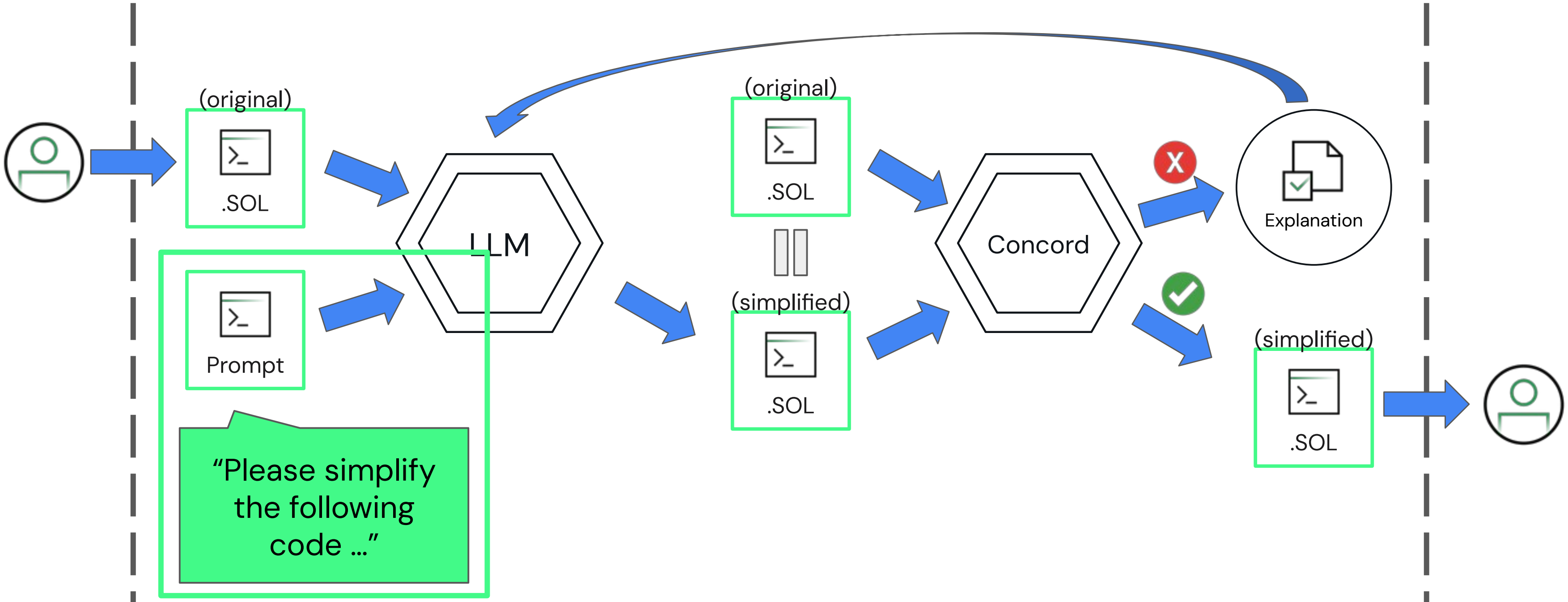


```
if (success && returndata.length >= 32 && abi.decode(returndata, (bool)))
{
      return;
}
```



```
if (success && returndata.length >= 32 && uint256(bytes32(returndata)) ==
1) {
      return;
}
```

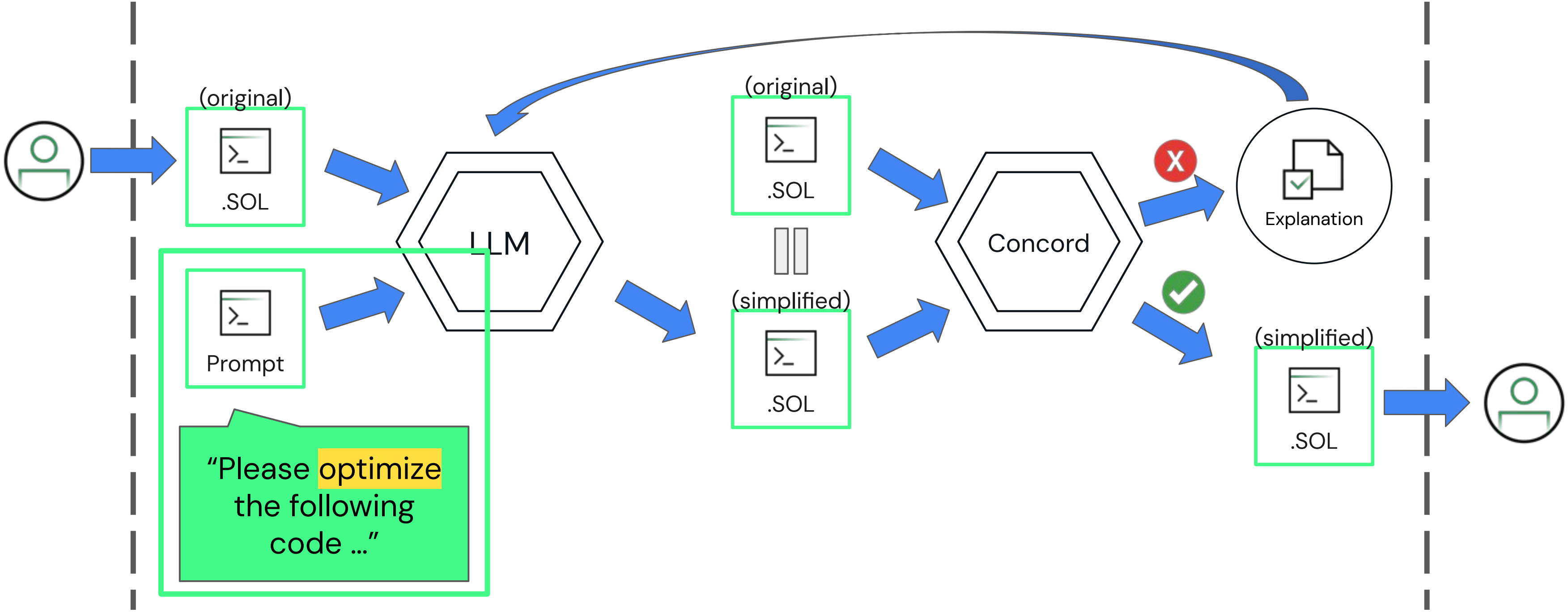
Concordance: Not Simply Simplification



"Please simplify the following code …"

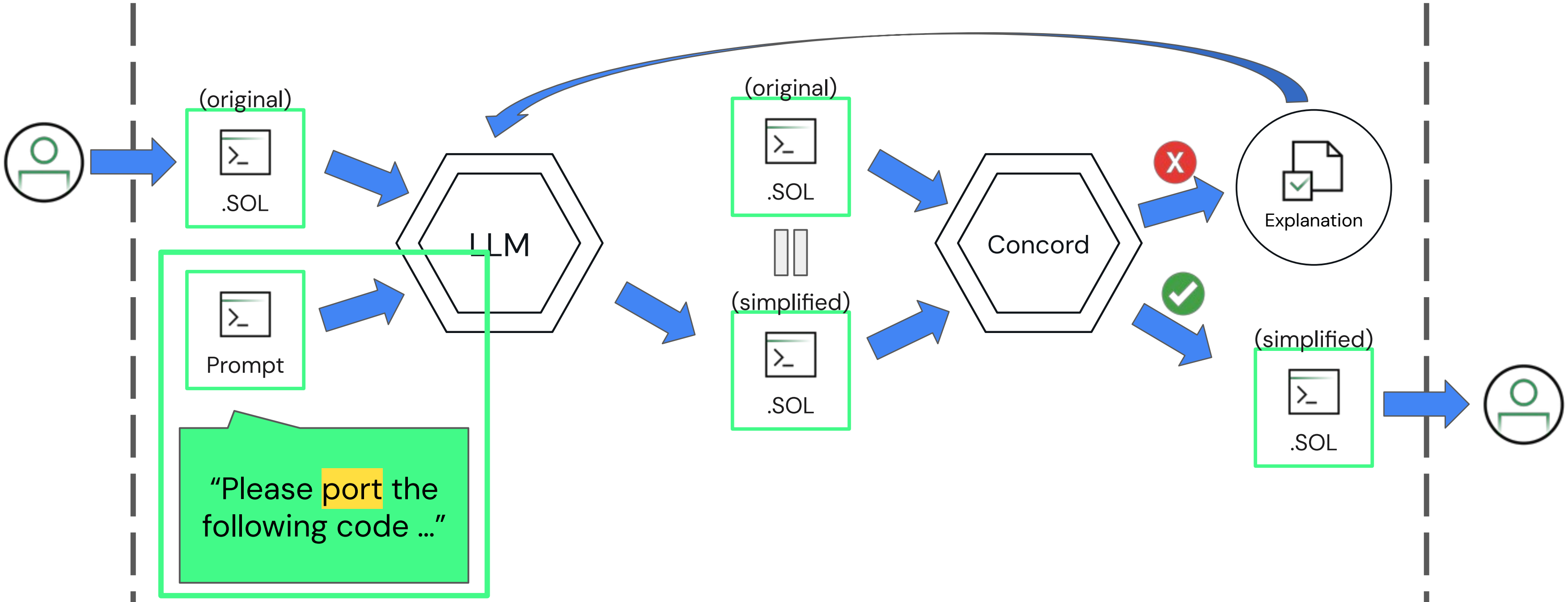
certora

Concordance: Not Simply Simplification



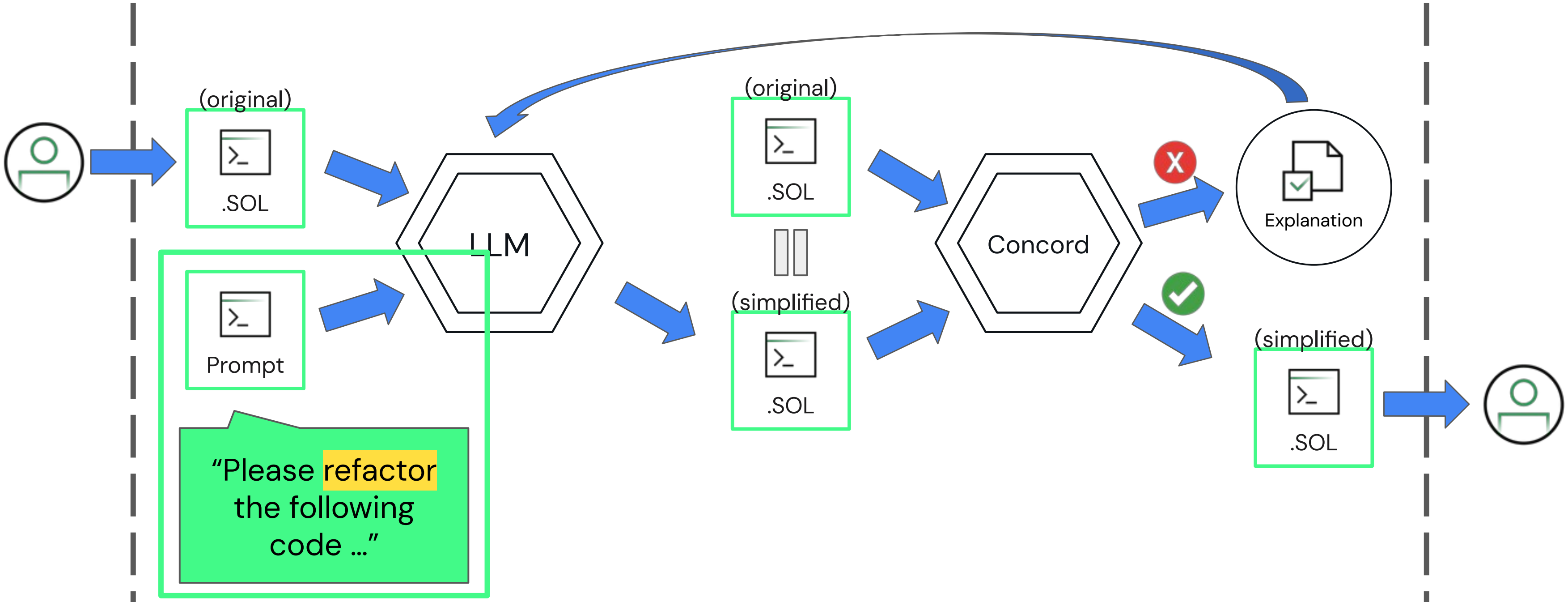
certora

Concordance: Not Simply Simplification



certora

Concordance: Not Simply Simplification



“Please refactor the following code ...”

certora

It’s open source!

(github link)

Thanks!

Questions?

certora