CERTORA

Move fast and break nothing

2022

CERTORA

The Certora Prover Pipeline

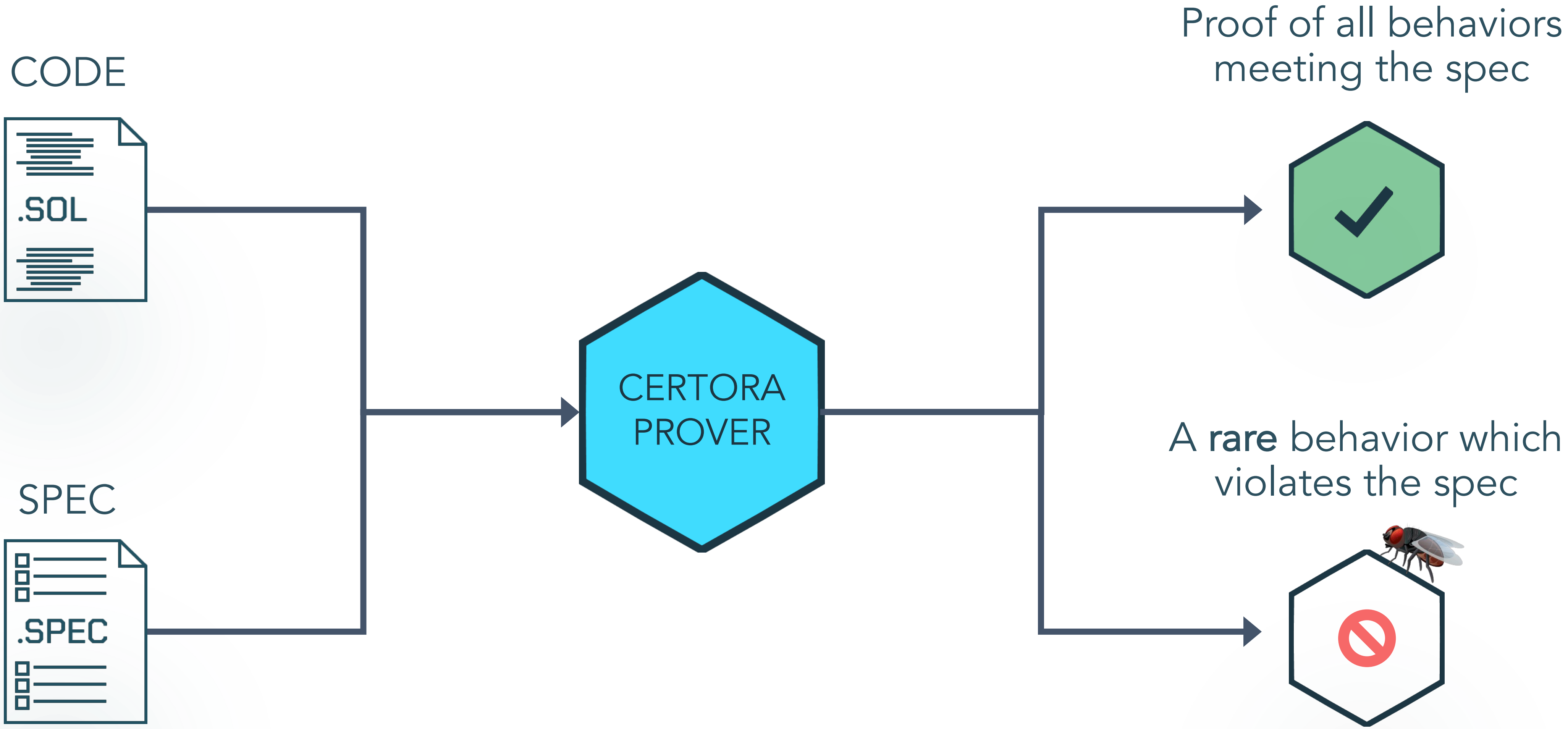
Chandrakana Nandi

Stanford Workshop

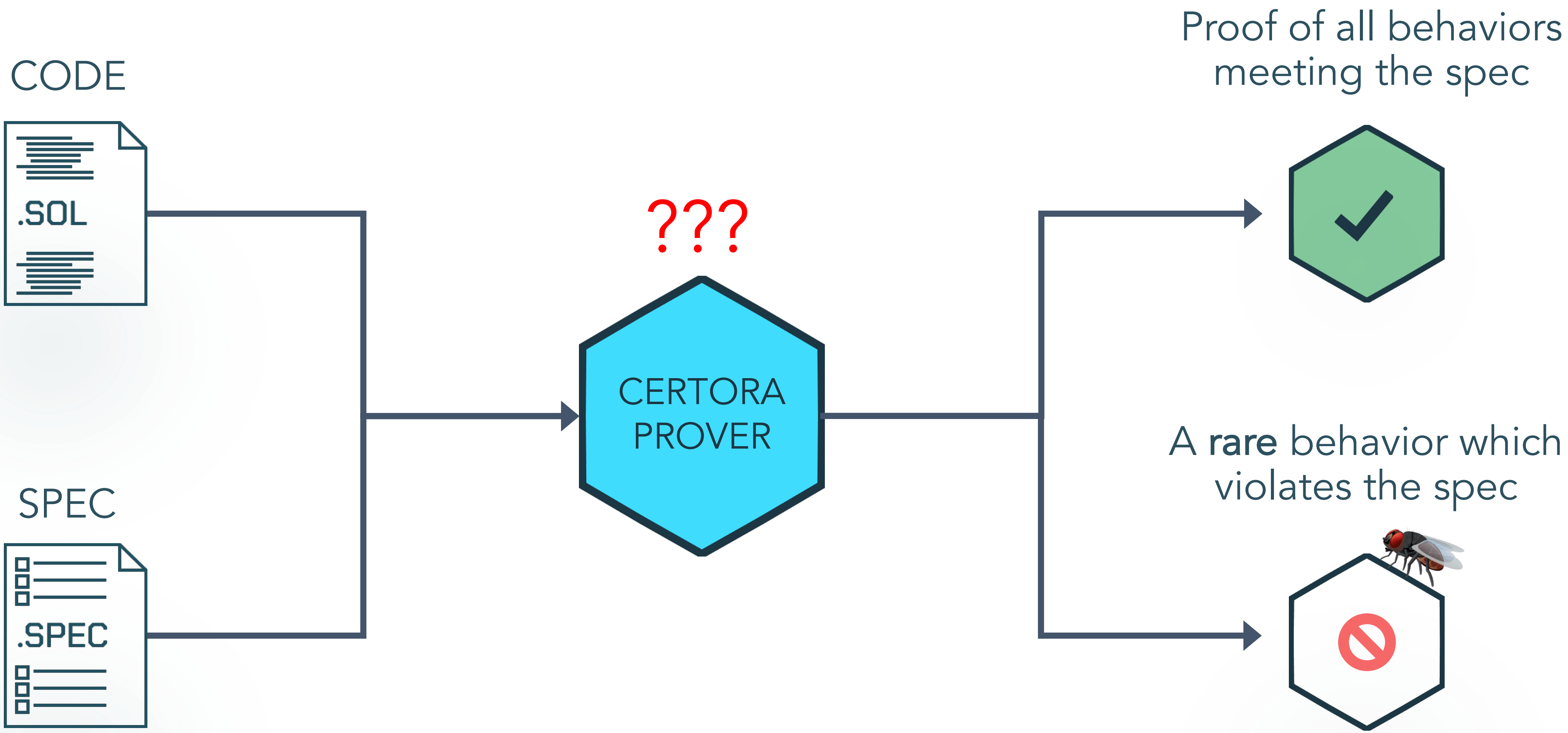
August 30, 2022

CERTORA

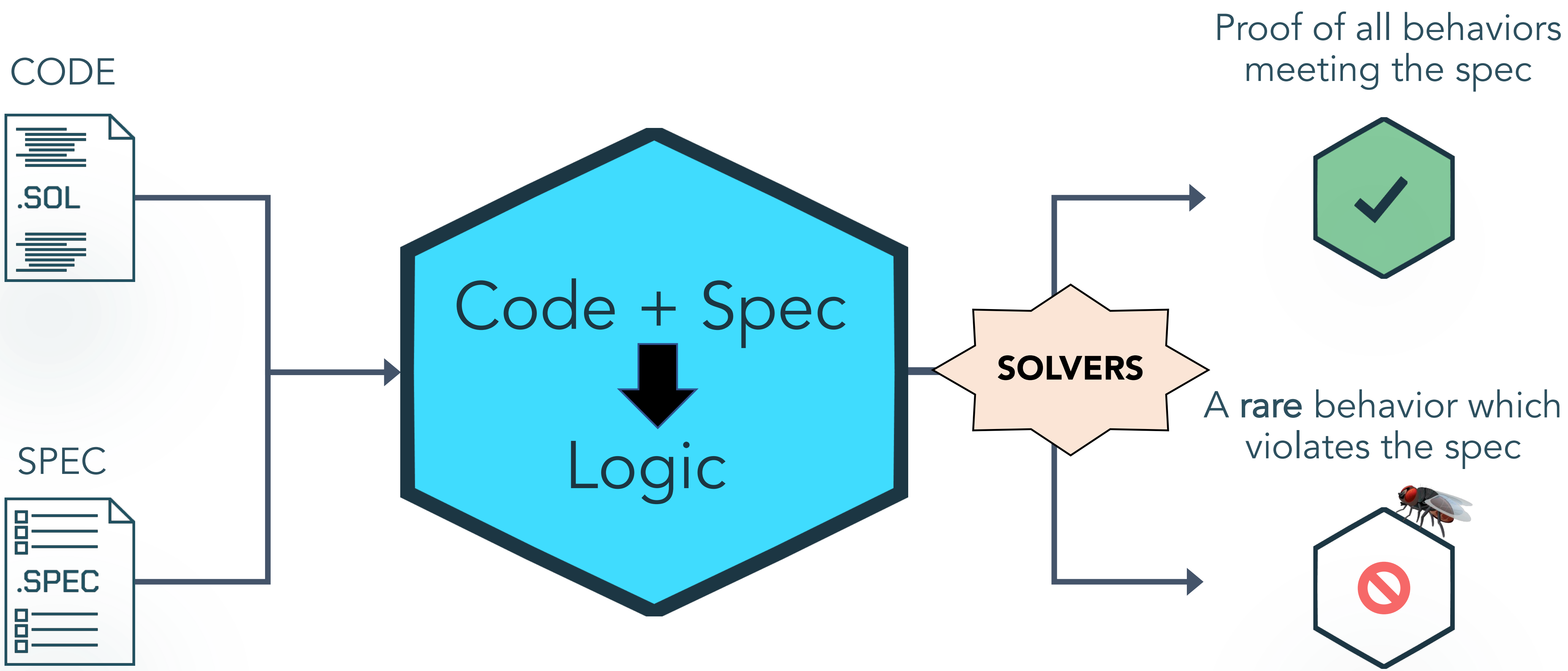
Formal Verification with Certora Prover



Formal Verification with Certora Prover



Formal Verification with Certora Prover

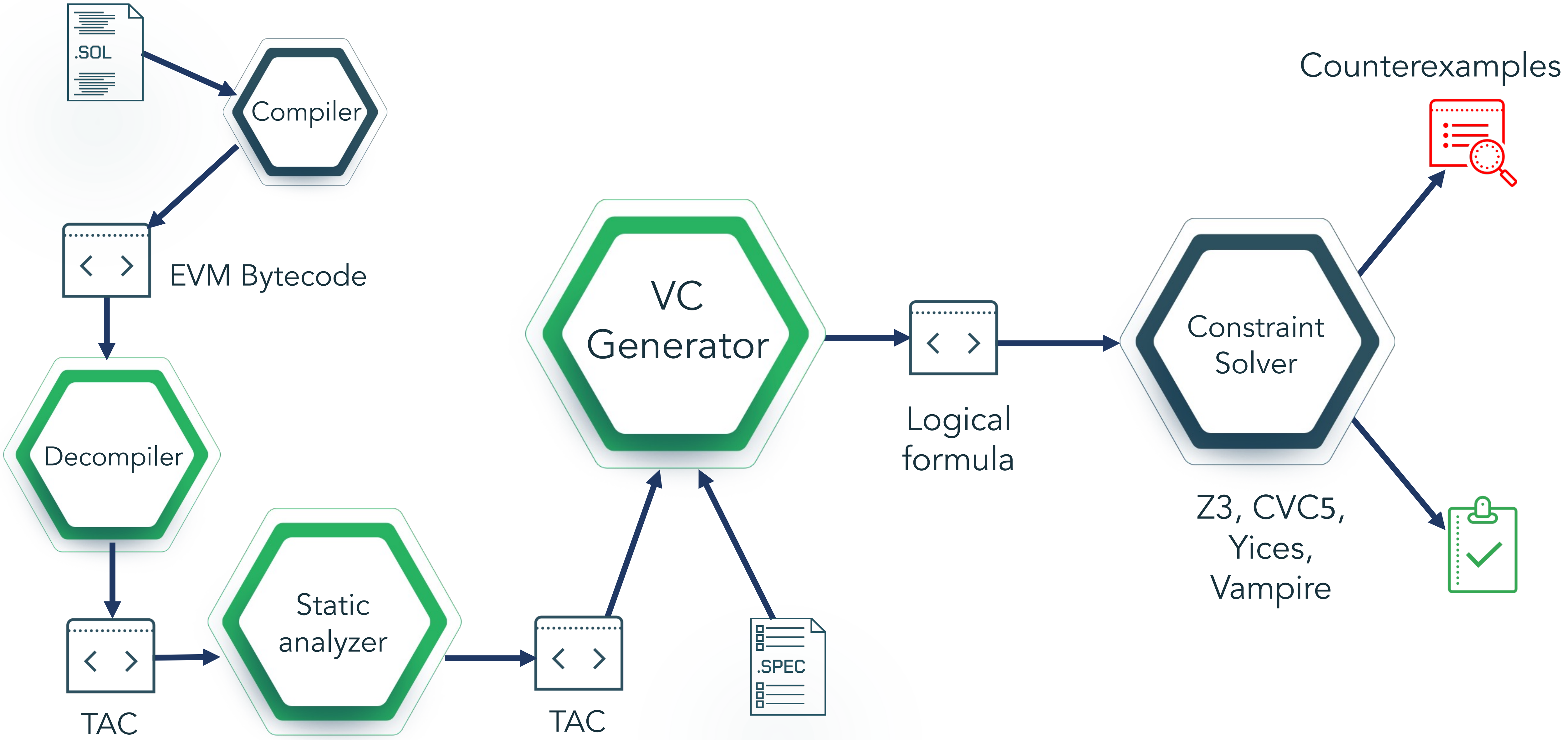


Formal Verification with Certora Prover

Systematically translate code + spec to equivalent formula

- Wrap code in specification using ergonomic DSL (CVL)
- Break code down into simple operations
- Meaning preserving simplifications and optimizations

Certora Prover Architecture



CERTORA

Simple Example

```
contract Bank {
    mapping (address => uint256) public funds;

    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }

    function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Simple Example

```
contract Bank {
    mapping (address => uint256) public funds;

    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }

    function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Simple Example

```
contract Bank {
    mapping (address => uint256) public funds;

    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }

    function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Simple Example

```
contract Bank {
    mapping (address => uint256) public funds;

    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }

    function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

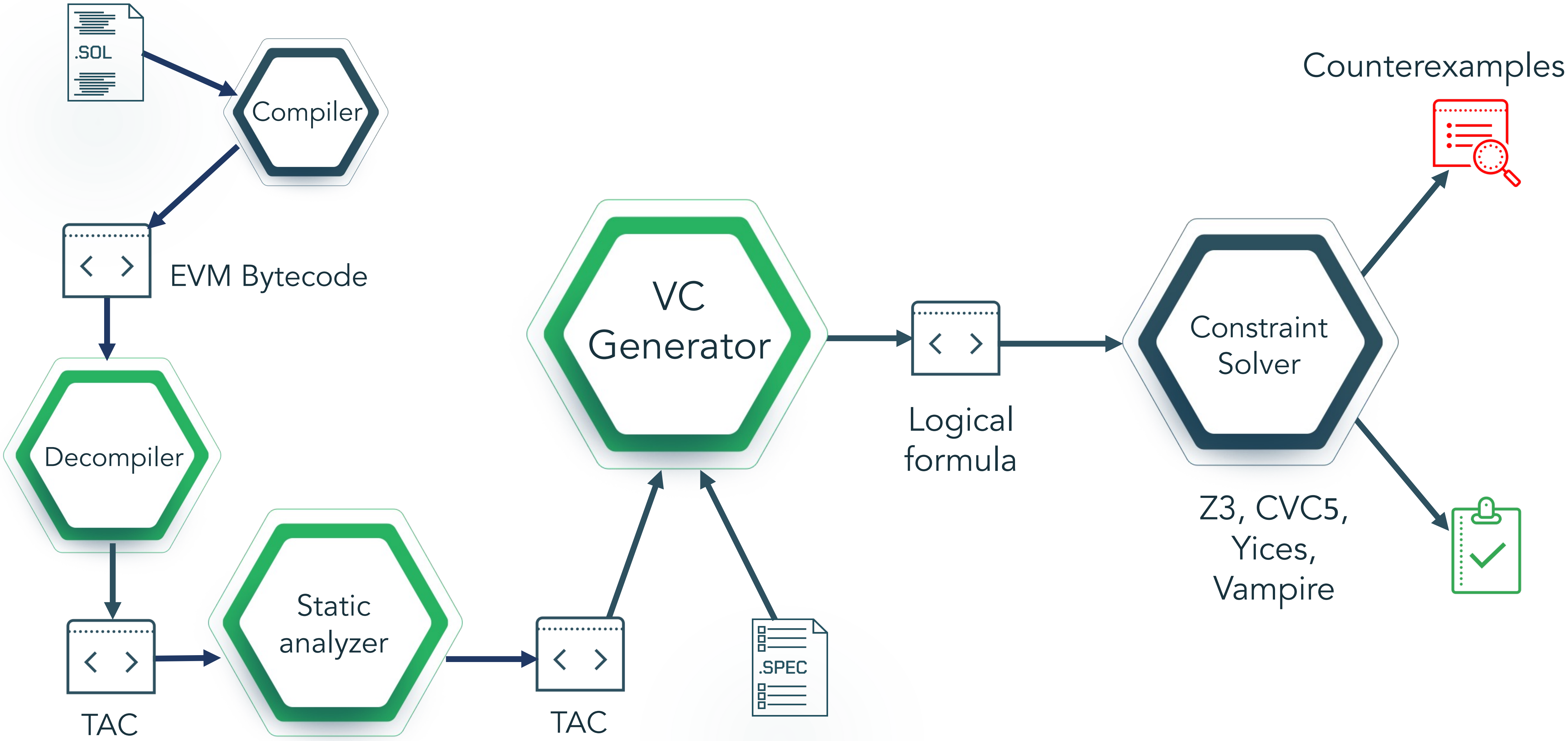
Simple Example

How do we know that **deposit** increases **funds** by **amount**?

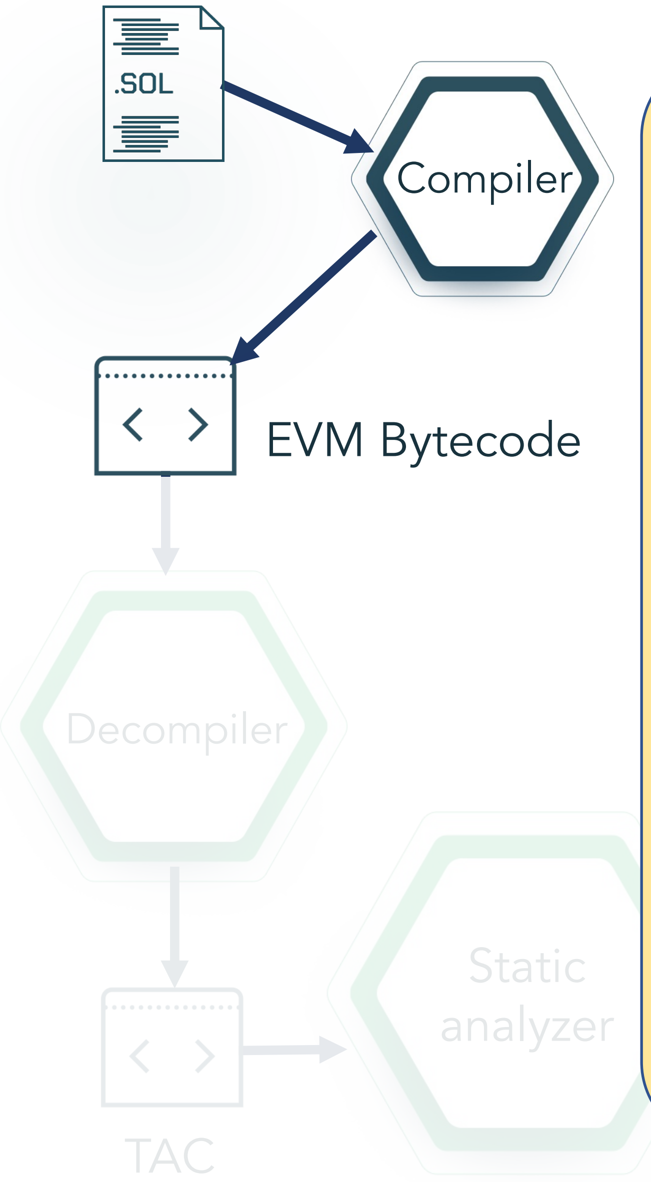
```
function deposit (uint256 amount) public payable {
      funds[msg.sender] += amount;
}

function getFunds (address account) public view returns (uint256) {
      return funds[account];
}
}
```

Certora Prover to the Rescue!



Certora Prover Works on Bytecode



Compile Solidity to get EVM Bytecode

Can support other EVM languages (Vyper)

Helps find compiler bugs!

Compiler Bugs Found by Certora Prover

CERTORA

Non-deterministic Solidity Transactions — Certora Bug Disclosure

Uri Kirstein Follow
May 31 · 5 min read

The Solidity Compiler Silently Corrupts Storage — Certora Bug Disclosure

Uri Kirstein Follow
May 30 · 4 min read

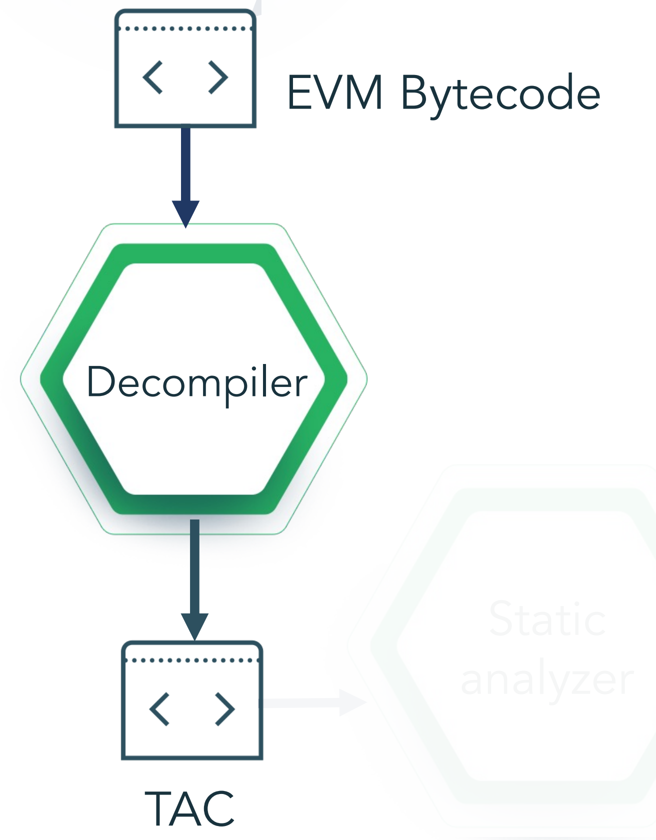
Memory Isolation Violation in Deserialization Code — Certora Bug Disclosure

Uri Kirstein Follow
Jun 2 · 4 min read

Bug Disclosure — Solidity Code Generation Bug Can Cause Memory Corruption

Uri Kirstein Follow
May 27 · 6 min read

Bytecode to Three-Address Code



Break down code into small simple steps

One operation per TAC instruction

Only a small number of instructions in TAC

Easier to analyze

CERTORA

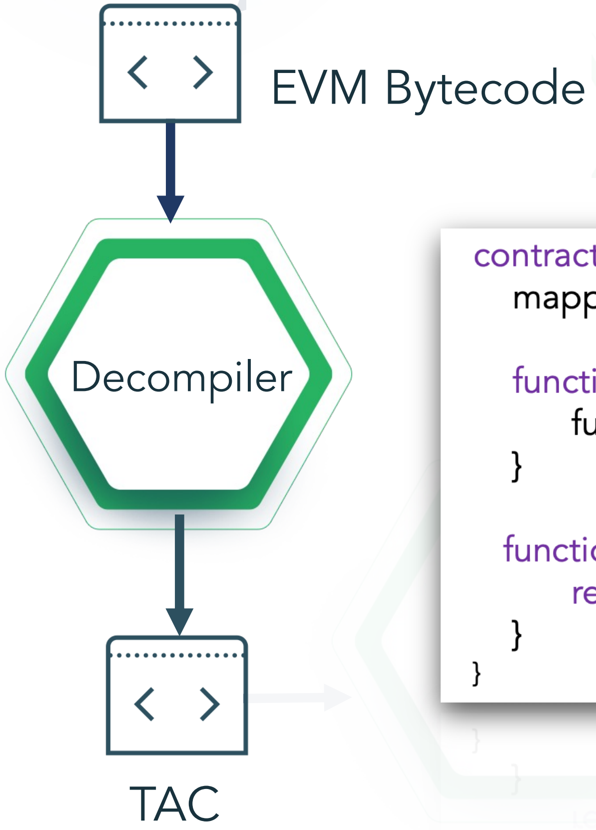
Bytecode to Three-Address Code

Break down code into small simple steps

One operation per TAC instruction

Only a small number of instructions in TAC

Easier to analyze




```
contract Bank {
    mapping (address => uint256) public funds;

    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }

    function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

```
Block 0_0_0_0_0_0_0_0: lastHasThrown = false
lastReverted = false
R0 = tacExtcodesize[tacAddress]
B1 = R0>0x0
assume B1
TRANSIENT::MetaKey(name=internal.func.finder.info, typ=class
analysis.ip.InternalFunctionFinderReport)=InternalFunctionFinderReport(unresolvedFunctions=[])::
tacM0x40 = 0x80
R2 = tacCalldatasize
B4 = R2<0x4
assume !B4
R15 = tacSighash
B19 = 0xb6b55f25==R15
assume B19
JUMPDEST 57_1024_0_0_0_0_0_0
R21 = tacCalldatasize
R22 = R21-0x4
B25 = R22<0x20
if B25:bool goto 75_1021_0_0_0_0_0_0 else goto 79_1021_0_0_0_0_0_0

Block 75_1021_0_0_0_0_0_0: lastHasThrown = false
lastReverted = true
TRANSIENT::MetaKey(name=tac.revert.path, typ=class java.lang.Boolean)=true::
revert and return M@0[0x0:0x0+0x0]

Block 79_1021_0_0_0_0_0_0: JUMPDEST 79_1021_0_0_0_0_0_0
R35 = tacCalldatabuf!4
TRANSIENT::MetaKey(name=internal.func.start, typ=class
analysis.ip.InternalFuncStartAnnotation)=InternalFuncStartAnnotation(id=2, startPc=208, exitPc=[86],
args=[InternalFuncArg(s=R35:bv256, offset=1, sort=SCALAR)],
functionId=ParseableName(exp=deposit(uint256)), stackOffsetToArgPos={1=0})::
JUMPDEST 208_1022_0_0_0_0_0_0
TRANSIENT::MetaKey(name=tac.internal.function.hint, typ=class
analysis.ip.InternalFunctionHint)=InternalFunctionHint(id=0, flag=0, sym=0xf196e50000)::
TRANSIENT::MetaKey(name=tac.internal.function.hint, typ=class
analysis.ip.InternalFunctionHint)=InternalFunctionHint(id=0, flag=1, sym=0x1)::
TRANSIENT::MetaKey(name=tac.internal.function.hint, typ=class
analysis.ip.InternalFunctionHint)=InternalFunctionHint(id=0, flag=4096, sym=R35:bv256)::
R53 = tacCaller
tacM0x0 = R53
tacM0x20 = 0x0
R65 = keccak256simple(tacM0x0,tacM0x20)
R68 = tacS!ce4604a00000000000000000000000001[R65]
R76 = R35+R68
tacS!ce4604a00000000000000000000000001[R65] = R76
TRANSIENT::MetaKey(name=internal.func.end, typ=class
analysis.ip.InternalFuncExitAnnotation)=InternalFuncExitAnnotation(id=2, rets=[])::
JUMPDEST 86_1024_0_0_0_0_0_0
TRANSIENT::MetaKey(name=tac.return.path, typ=class java.lang.Boolean)=true::
return M@0[0x0:0x0+0x0]
```

Static Analysis on TAC

Even in TAC, instructions can have subtle dependencies

Gather facts at various program points (e.g., points-to relation)

Segment memory into disjoint non-interfering sets of pointers

Lower burden on subsequent steps in the pipeline

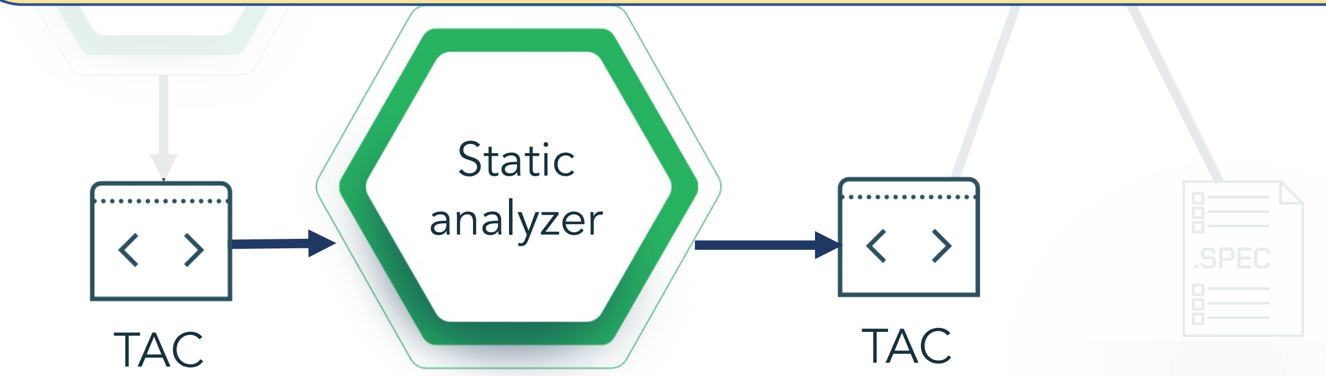
Static Analysis on TAC

Even in TAC, instructions can have subtle dependencies

Gather facts at various program points (e.g., points-to relation)

Segment memory into disjoint non-interfering sets of pointers

Lower burden on subsequent steps in the pipeline




```
MyStruct memory x = { f: 1 };
MyStruct memory y = { f: 2 };
y.f = 3;
assert(x.f == 1);
```

Static Analysis on TAC

Even in TAC, instructions can have subtle dependencies

Gather facts at various program points (e.g., points-to relation)

Segment memory into disjoint non-interfering sets of pointers

Lower burden on subsequent steps in the pipeline

does not affect x

```
MyStruct memory x = { f: 1 };
MyStruct memory y = { f: 2 };
y.f = 3;
assert(x.f == 1);
```

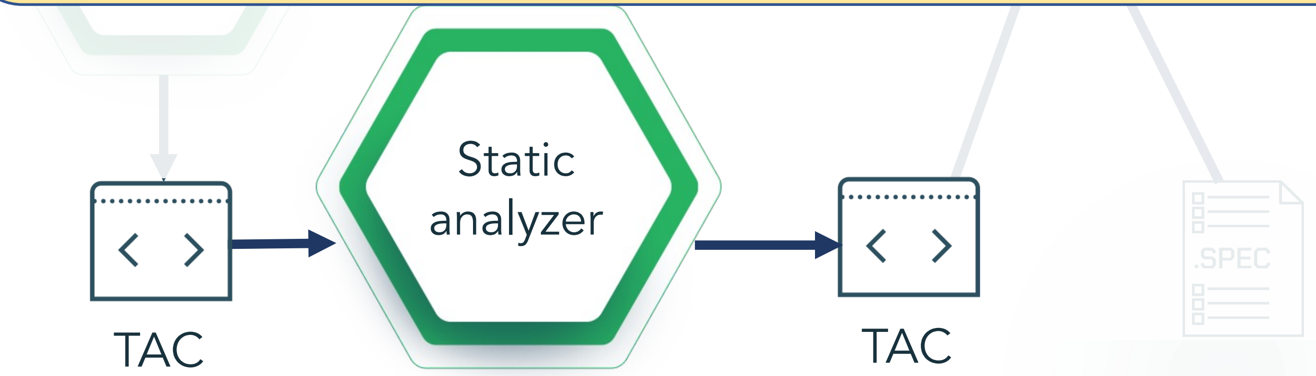
Static Analysis on TAC

Even in TAC, instructions can have subtle dependencies

Gather facts at various program points (e.g., points-to relation)

Segment memory into disjoint non-interfering sets of pointers

Lower burden on subsequent steps in the pipeline




```
MyStruct memory x = { f: 1 };
MyStruct memory y = { f: 2 };
y.f = 3;
assert(x.f == 1);
```

Static Analysis on TAC

Even in TAC, instructions can have subtle dependencies

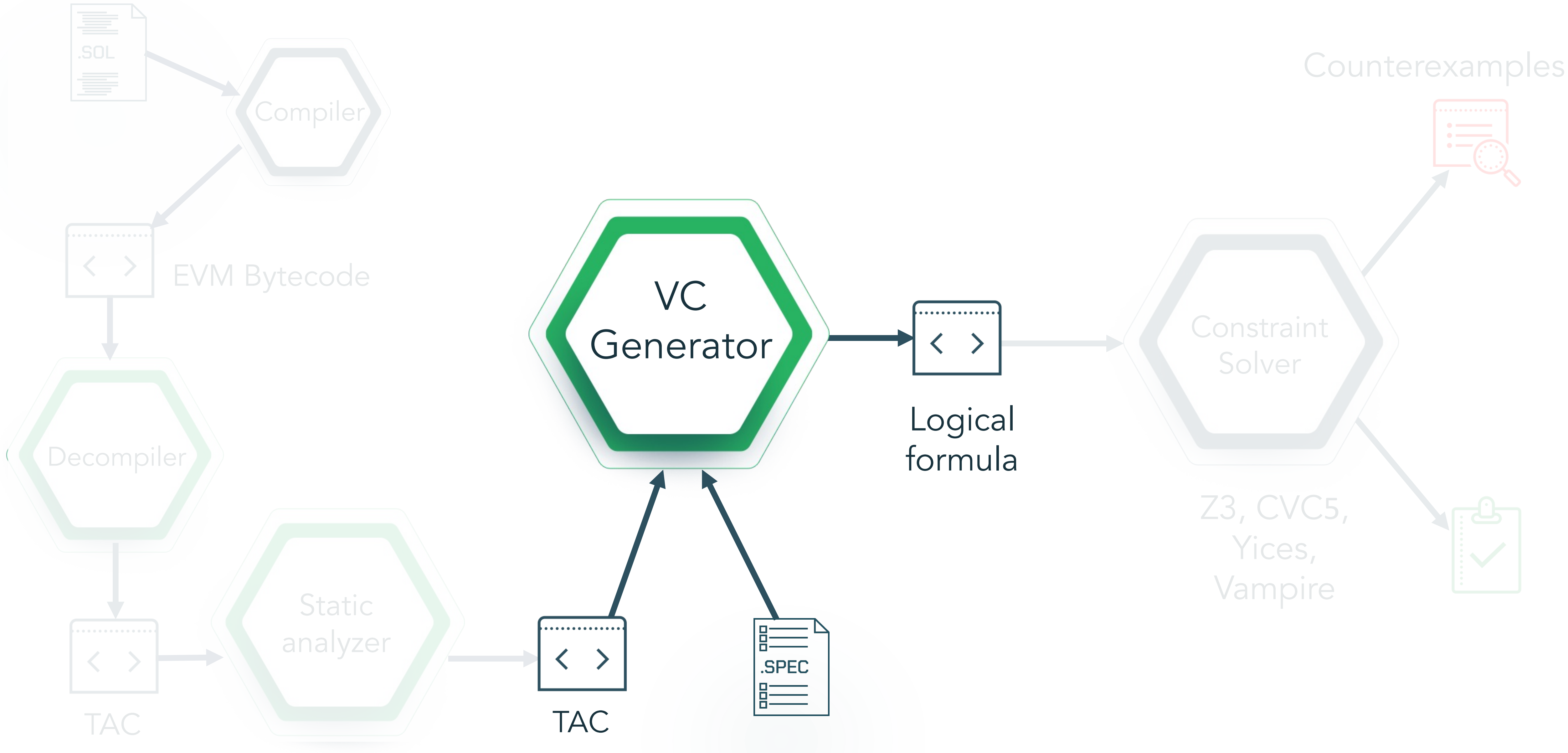
Gather facts at various program points (e.g., points-to relation)

Segment memory into disjoint non-interfering sets of pointers

Lower burden on subsequent steps in the pipeline


```
MyStruct memory x = { f: 1 };
assert(x.f == 1);
```

Generate Verification Conditions



Hoare Triples

Hoare Triple: $\{P\}\ S\ \{Q\}$

Hoare Triples

Hoare Triple: {P} S {Q}

If P holds before executing S, then Q holds after executing S

Weakest Precondition

Hoare Triple: {P} S {Q}

If P holds before executing S, then Q holds after executing S

WP (S, Q): weakest predicate such that Q holds after executing S

{WP (S, Q)} S {Q}

Weakest Precondition

CERTORA

Hoare Triple: {P} S {Q}

If P holds before executing S, then Q holds after executing S

WP (S, Q): weakest predicate such that Q holds after executing S

{WP (S, Q)} S {Q}

Then to prove the triple, just show that P ⇒ WP(S, Q)

Weakest Precondition

Hoare Triple: {P} S {Q}

If P holds before executing S, then Q holds after executing S

WP (S, Q): weakest predicate such that Q holds after executing S

{WP (S, Q)} S {Q}

Then to prove the triple, just show that P $\Rightarrow$ WP(S, Q)

Thus, if P $\Rightarrow$ WP (S, Q) then {P} S {Q}

Weakest Precondition

Hoare Triple: {P} S {Q}

If P holds before executing S, then Q holds after executing S

WP (S, Q): weakest predicate such that Q holds after executing S

{WP (S, Q)} S {Q}

Then to prove the triple, just show that <u>$P \Rightarrow WP(S, Q)$</u>

Thus, if $P \Rightarrow WP(S, Q)$ then {P} S {Q}

Weakest Precondition

Where do P and Q come from?

Hoare Triple: {P} S {Q}

If P holds before executing S, then Q holds after executing S

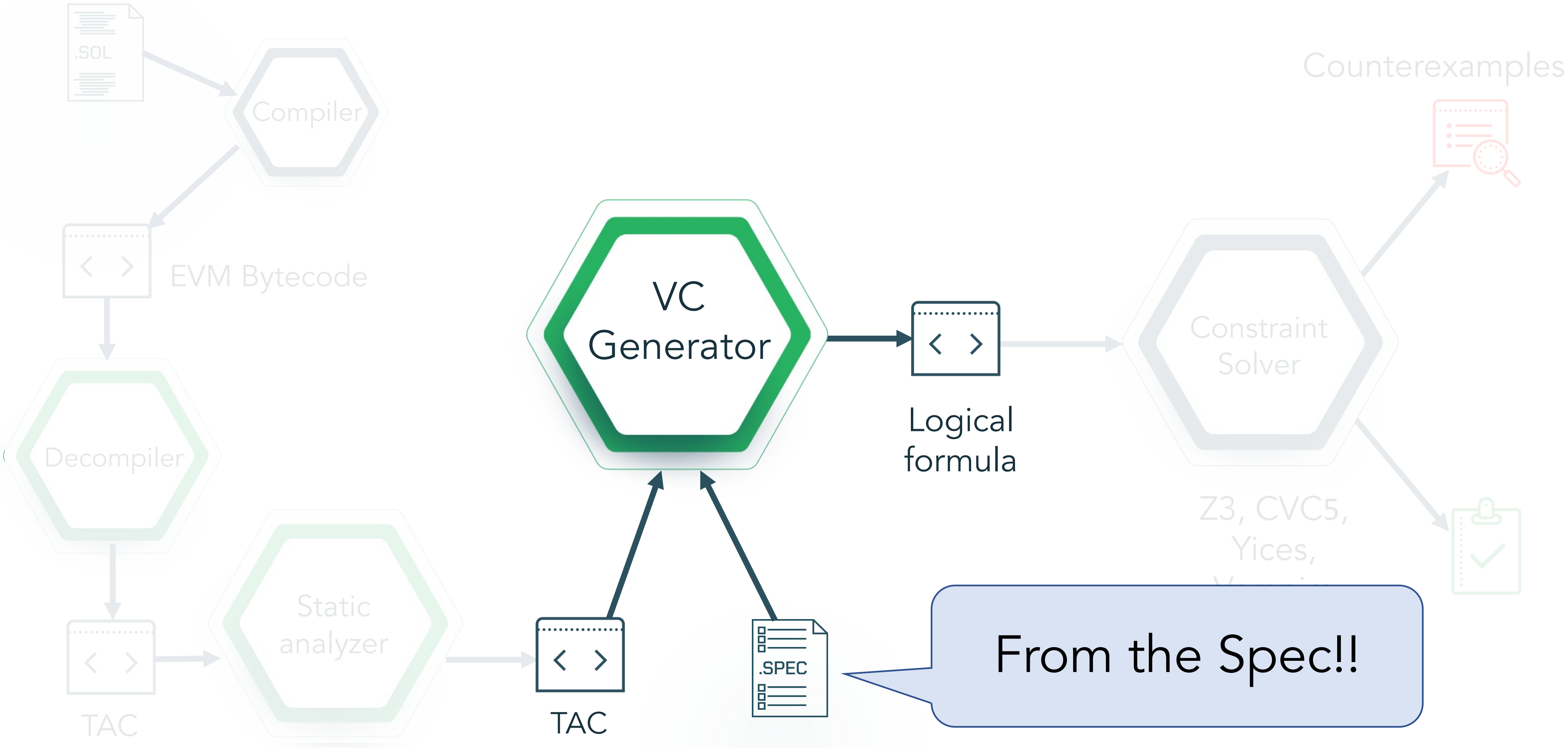
WP (S, Q): weakest predicate such that Q holds after executing S

{WP (S, Q)} S {Q}

Then to prove the triple, just show that <u>$P \Rightarrow WP(S, Q)$</u>

Thus, if $P \Rightarrow WP(S, Q)$ then {P} S {Q}

Generate Verification Conditions



Writing the Specification

How do we know that **deposit** increases **funds** by **amount?**

```
function deposit (uint256 amount) public payable {
    funds[msg.sender] += amount;
}
```

Need to first write "deposit increases funds by amount" more formally so that we can automatically check it!

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

CERTORA

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

CERTORA

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Inline from contract

CERTORA

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Allows us to get pre and post conditions!

CERTORA

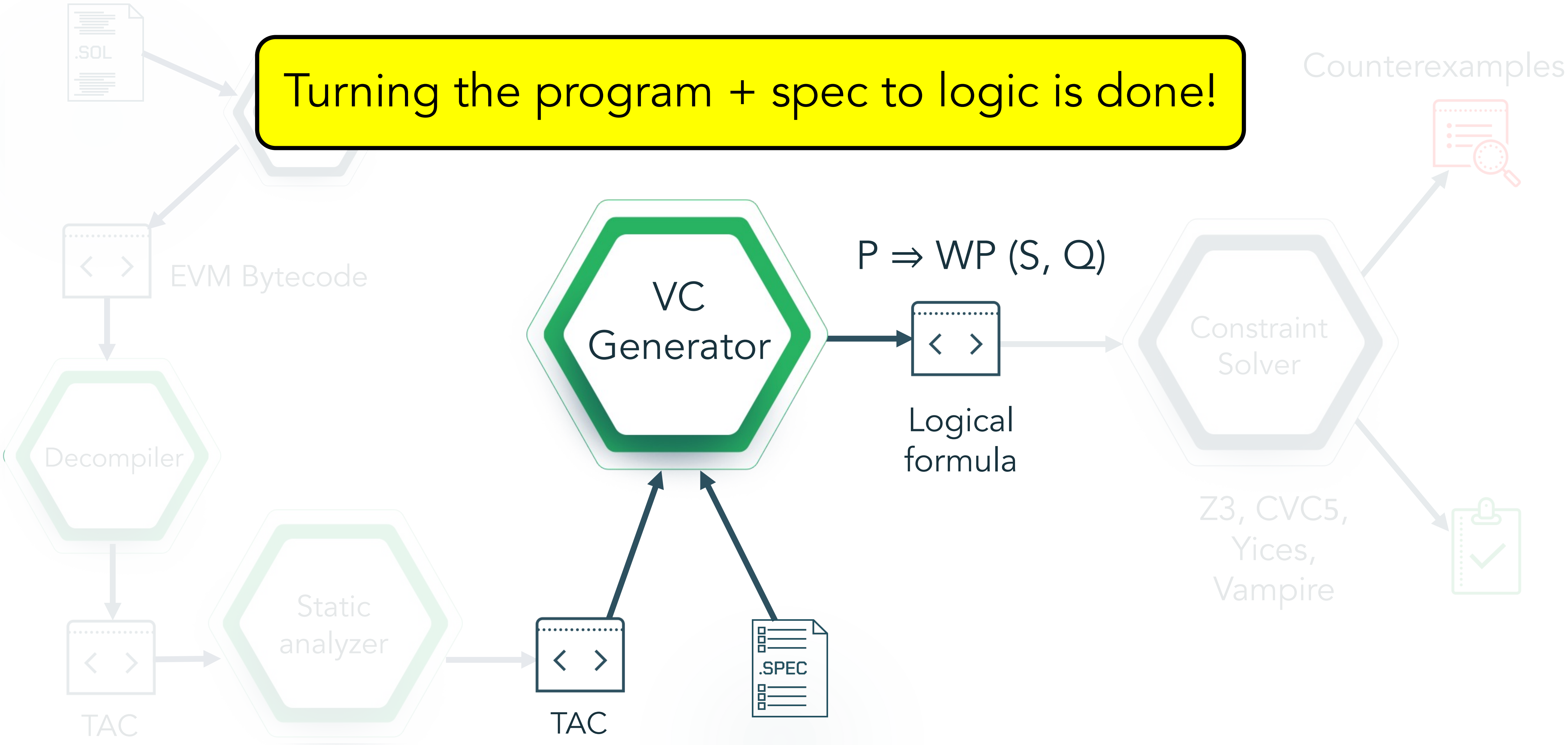
Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

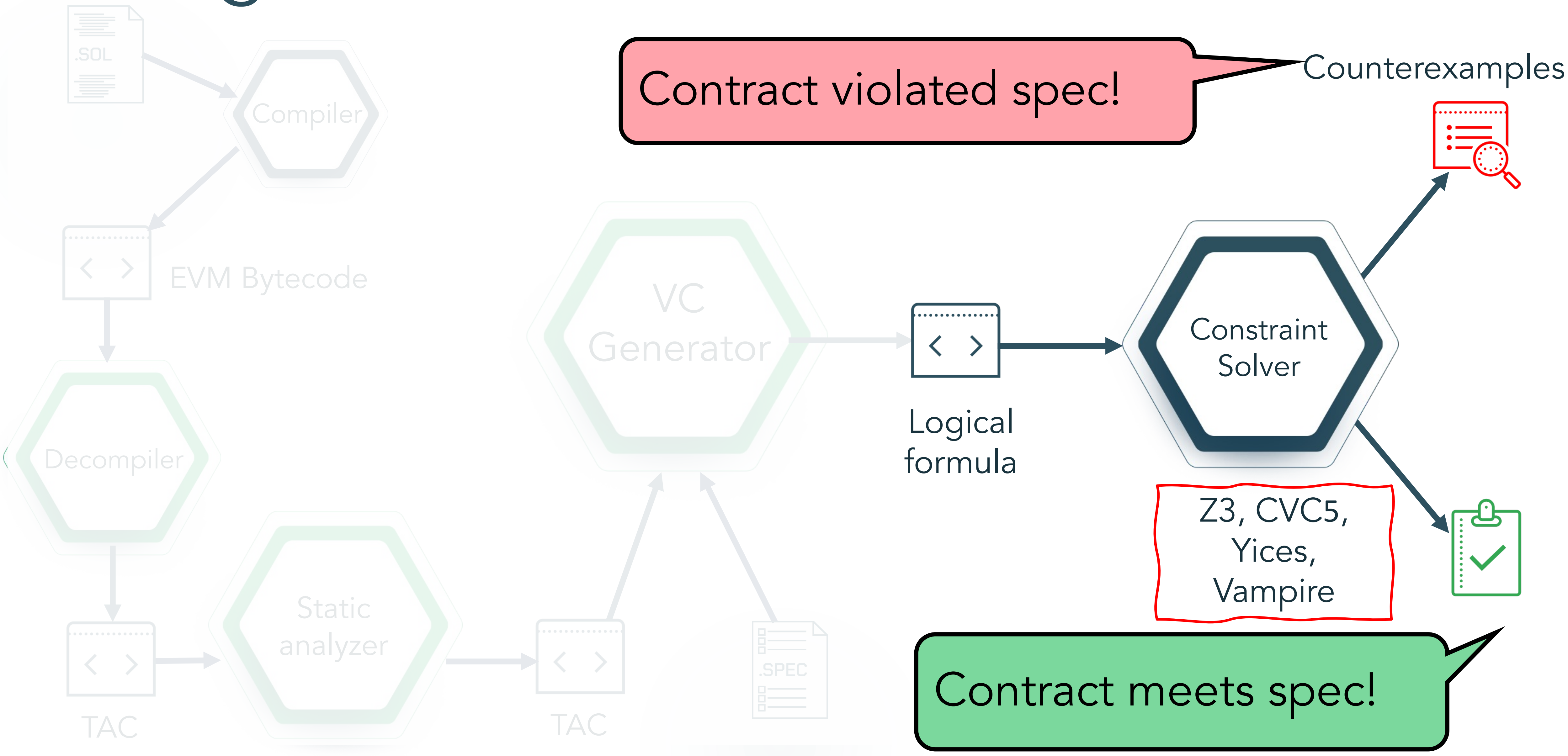
Must hold for ALL values of amount!

Generate Verification Conditions

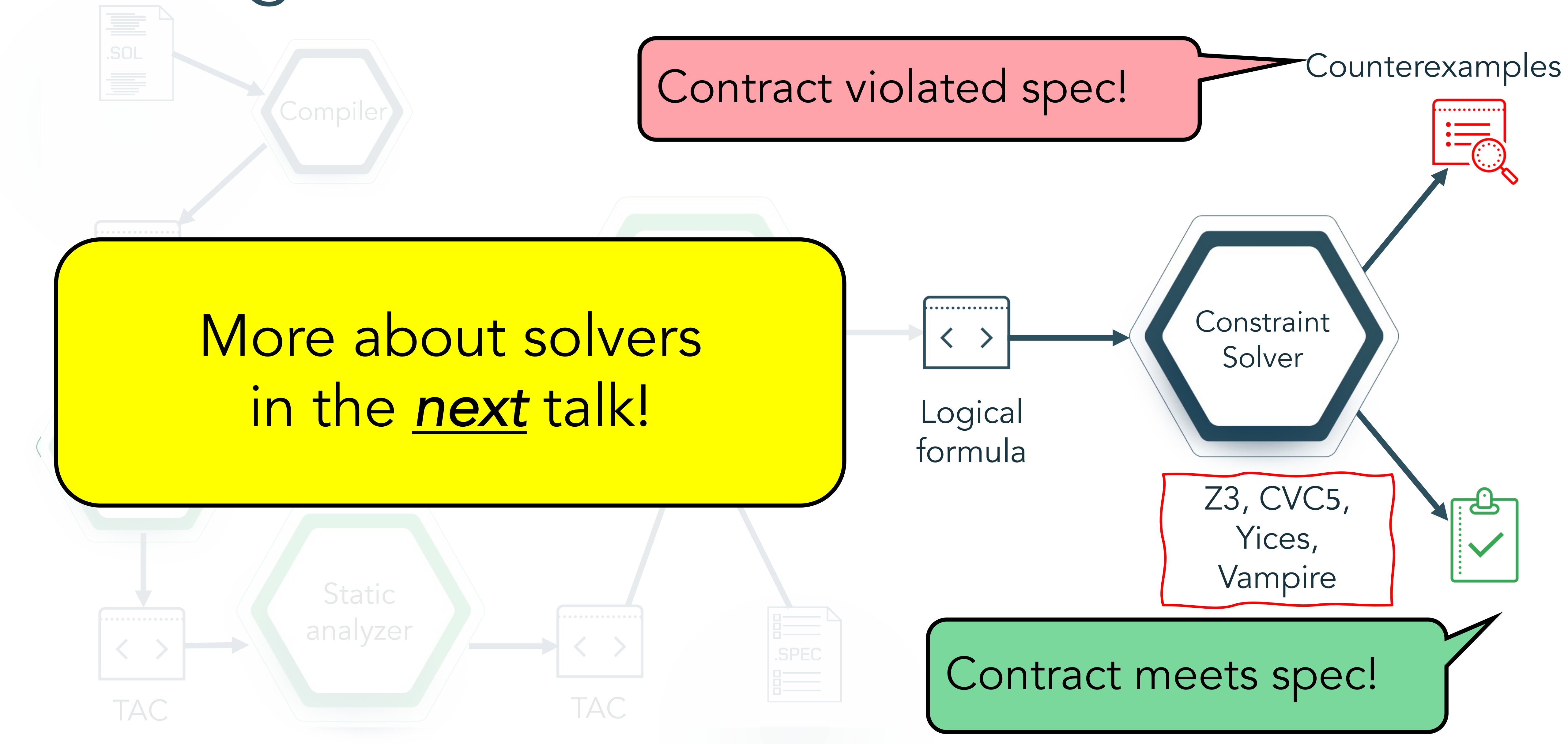
Turning the program + spec to logic is done!



Using Constraint Solvers



Using Constraint Solvers



Putting It All Together

Spec

Font size | Start Verification

```
pragma specify 0.1
methods {
     getFunds(address) returns uint256 envfree
}

rule deposit_ok(uint256 amount) {
     env e;
     uint256 before_deposit =  getFunds(e.msg.sender);
     deposit(e, amount);
     uint256 after_deposit = getFunds(e.msg.sender);
     assert(after_deposit == before_deposit + amount);
}
```

Solidity

Bank | 0.6.10

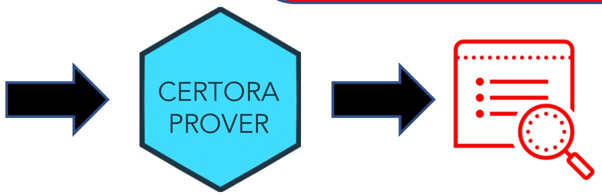
```
contract BankBroken {
     uint256 public totalFunds;
     mapping (address => uint256) public funds;

     function deposit(uint256 amount) public payable {
          funds[msg.sender] += amount;
     }

     function getFunds(address account) public view returns (uint256) {
          return funds[account];
     }
}
```

https://demo.certora.com

Overflow!



Call Trace

deposit_ok(amount=2)

Variables

e.msg.sender=0x401
e.msg.address=0x402
e.msg.value=3
before_deposit=0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe
after_deposit=0

Quis custodiet ipsos custodes?

Is the spec itself trustworthy?

Is the Spec Trustworthy?

CERTORA

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Spec

```
contract Bank {
    mapping (address => uint256) public funds;
    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }
   function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Code

Is the Spec Trustworthy?

CERTORA

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Is it vacuously true?
Does it catch errors?

```
contract Bank {
    mapping (address => uint256) public funds;
    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }
   function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Code

Mutation *Verification*

CERTORA

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Is it vacuously true?
Does it catch errors?

```
contract Bank {
    mapping (address => uint256) public funds;
    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }
   function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Code

Mutation *Verification*

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

```
contract Bank {
    mapping (address => uint256) public funds;
    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }
    function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Is it vacuously true?
<u>**Does it catch errors?**</u>

Spec must catch mutants

Mutation *Verification*

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}


contract Bank {
    mapping (address => uint256) public funds;
    function deposit (uint256 amount) public payable {
        funds[msg.sender] += 1;
    }
   function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Is it vacuously true?
<u>Does it catch errors?</u>

Spec must catch mutants

Mutation *Verification*

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

```
contract Bank {
    mapping (address => uint256) public funds;
    function deposit (uint256 amount) public payable {
        // funds[msg.sender] += amount;
    }
   function getFunds (address account) public view returns (uint256) {
        return funds[account];
    }
}
```

Is it vacuously true?
Does it catch errors?

Spec must catch mutants

Mutation *Verification*

CERTORA

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Is it vacuously true?

Does it catch errors?

```
contract Bank {
    mapping (address => uint256) public funds;
    function deposit (uint256 amount) public payable {
        funds[msg.sender] += amount;
    }
    function getFunds (address account) public view returns (uint256) {
        return funds[account] - 1;
    }
}
```

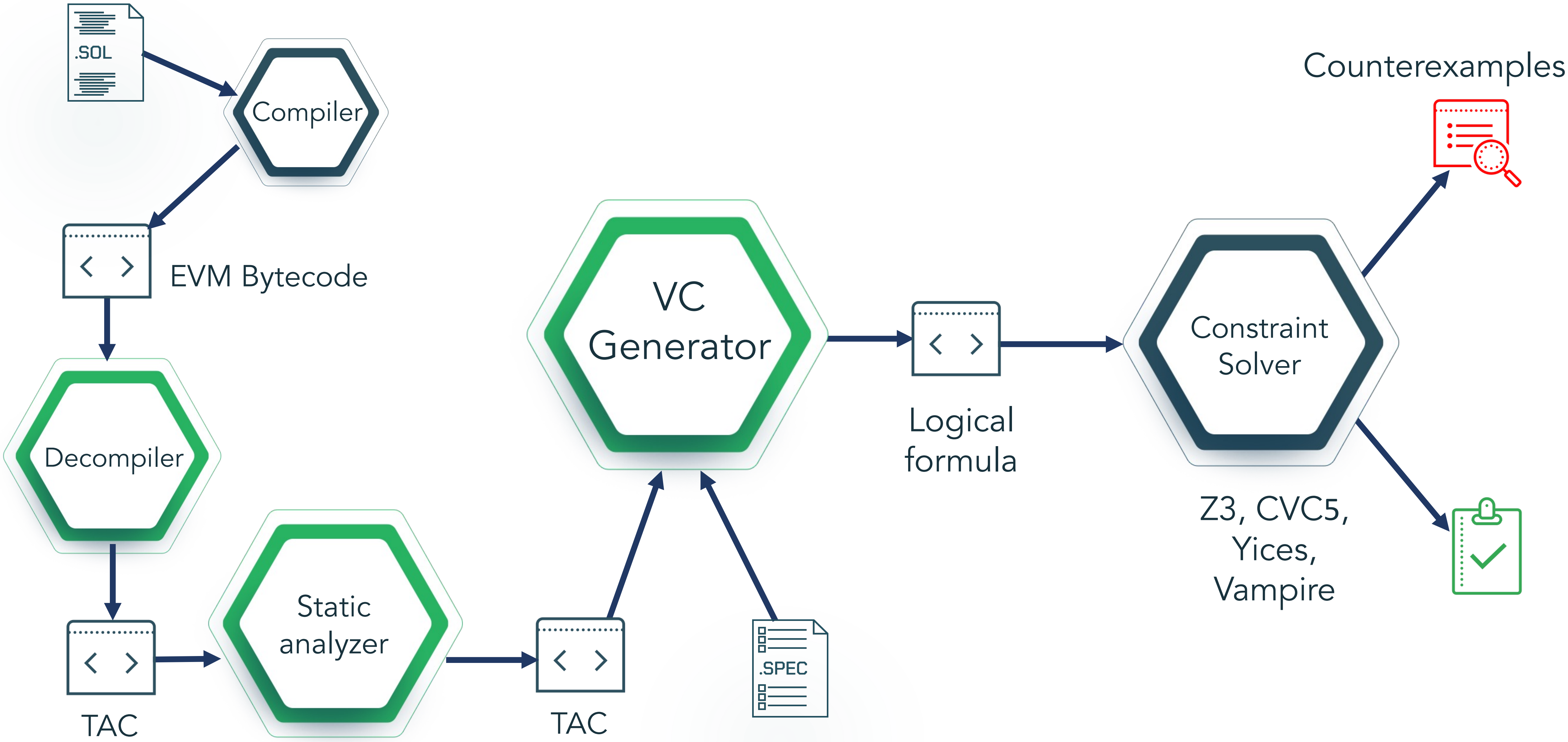
Spec must catch mutants

Mutation *Verification*

Improve spec based on mutants that “got away”

Assign spec a “score” based on #mutants caught

The Certora Prover Pipeline



Thank You!

Specification in CVL

```
rule deposit_ok (uint256 amount) {
    env e;
    uint256 before_deposit =  getFunds (e.msg.sender);
    deposit (e, amount);
    uint256 after_deposit = getFunds (e.msg.sender);
    assert (after_deposit == before_deposit + amount);
}
```

Not executable but looks like Solidity!