CERTORA

Move fast and break nothing

2022

CERTORA

Formal Verification of Smart Contracts with The Certora Prover

Jaroslav Bendík

April 2022

Blockchain and Smart Contracts

Blockchain

- A distributed database
- Chronologically ordered data
- Decentralized
- Cryptographic security measures
- Immutable
- Usual use: digital ledger

Blockchain and Smart Contracts

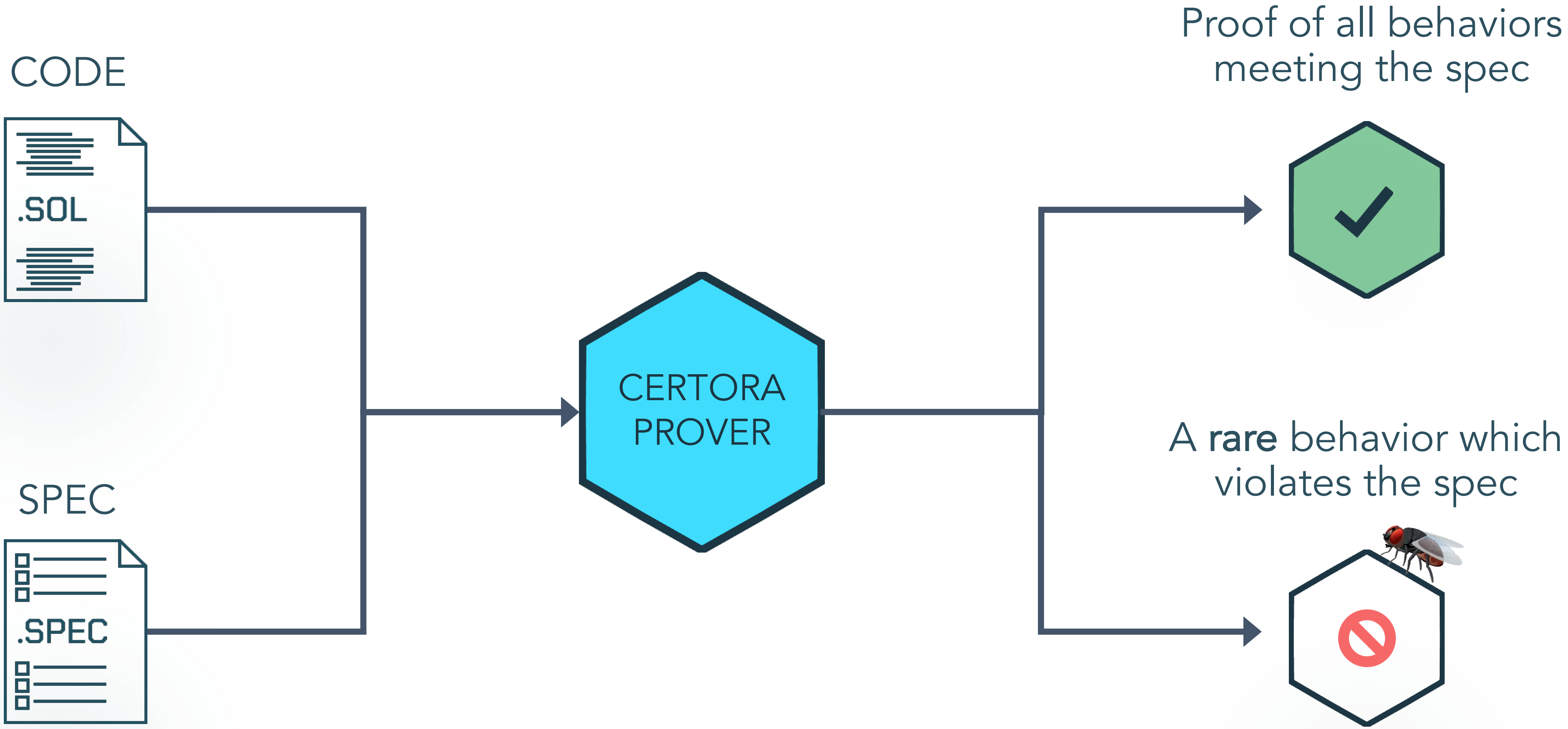
Blockchain

- A distributed database
- Chronologically ordered data
- Decentralized
- Cryptographic security measures
- Immutable
- Usual use: digital ledger

Smart Contract

- A set of functions running on Ethereum blockchain
- A user can invoke some of the functions
- Successful function invocations are irreversible
- Unsuccessful function invocations revert
- A maximum size of 24KB
- Cannot be deleted/changed once deployed

Formal Verification with Certora Prover



CODE

.SOL

SPEC

.SPEC

CERTORA PROVER

Proof of all behaviors meeting the spec

A **rare** behavior which violates the spec

Example Smart Contract

```
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];
    }
}
```

Example Smart Contract

```
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];
    }
}
```

Example Smart Contract

```
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];
    }
}
```

CERTORA

Example Smart Contract

```
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];
    }
}
```

Example Smart Contract

```
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];
    }
}
```

Example Smart Contract

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];
    }
}
```

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

Not executable but looks like Solidity!

Specification in CVL

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

Specification in CVL

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

Specification in CVL

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

Specification in CVL

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

Specification in CVL

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

Inline from contract

Specification in CVL

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

Specification in CVL

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

Must hold for ALL values of amount!

Certora Verification Language Overview

- Assumptions + assertions
- Invariants
- Ghost functions + Hooks (ghost solidity functions)
- Summary functions (replace solidity functions)
- CVL functions (to avoid repeated code in .spec files)
- Quantifiers

Certora Verification Language Overview

- Assumptions + assertions
- Invariants
- Ghost functions + Hooks (ghost solidity functions)
- Summary functions (replace solidity functions)
- CVL functions (to avoid repeated code in .spec files)
- Quantifiers

```
require forall address i. forall address j. funds(i) + funds(j) <= totalFunds();
```

Certora Verification Language Overview

- Assumptions + assertions
- Invariants
- Ghost functions + Hooks (ghost solidity functions)
- Summary functions (replace solidity functions)
- CVL functions (to avoid repeated code in .spec files)
- Quantifiers

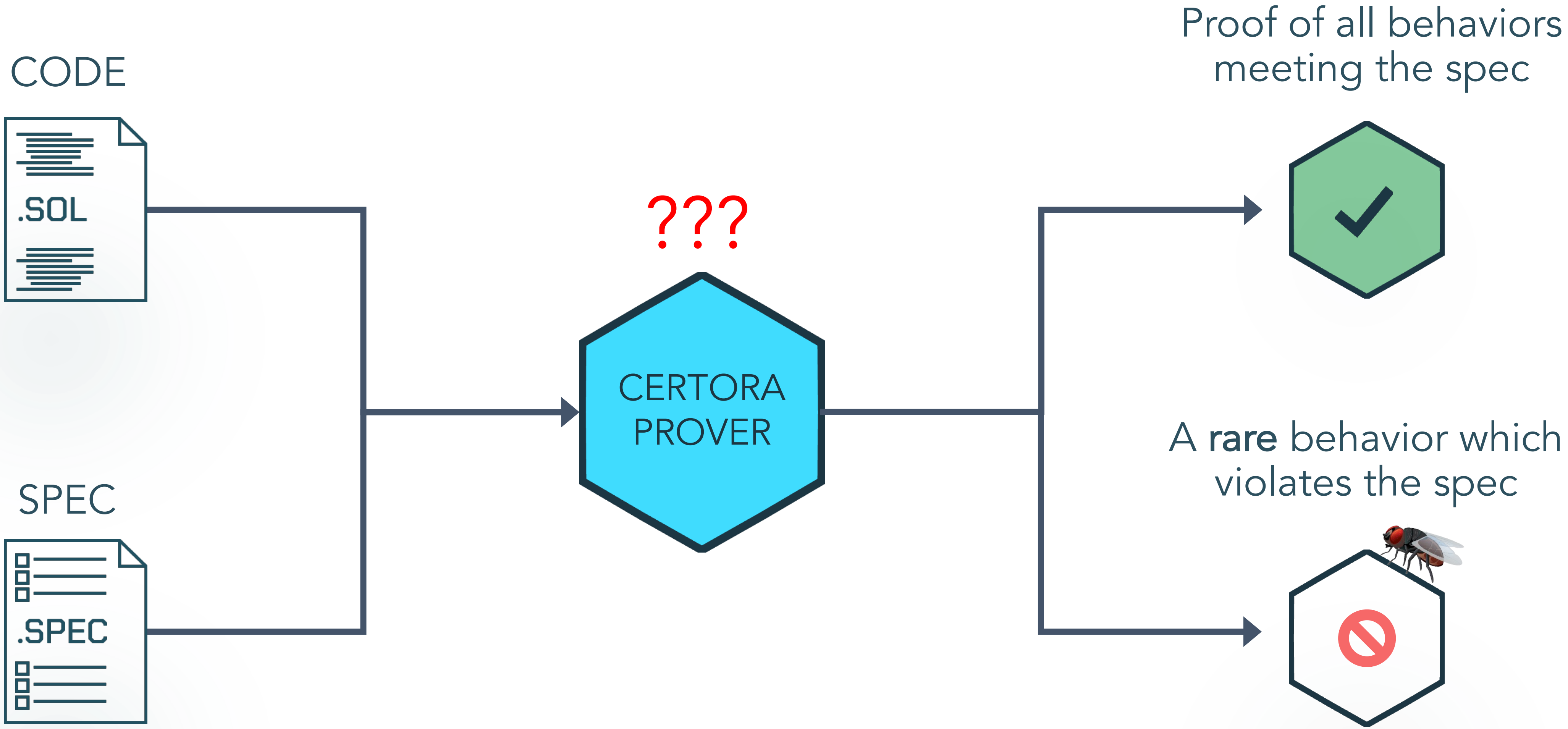
```
require forall address i. forall address j. funds(i) + funds(j) <= totalFunds();
```

Certora Verification Language Overview

- Assumptions + assertions
- Invariants
- Ghost functions + Hooks (ghost solidity functions)
- Summary functions (replace solidity functions)
- CVL functions (to avoid repeated code in .spec files)
- Quantifiers

```
require forall address i. forall address j. funds(i) + funds(j) <= totalFunds();
```

Formal Verification with Certora Prover



CODE

.SOL

SPEC

.SPEC

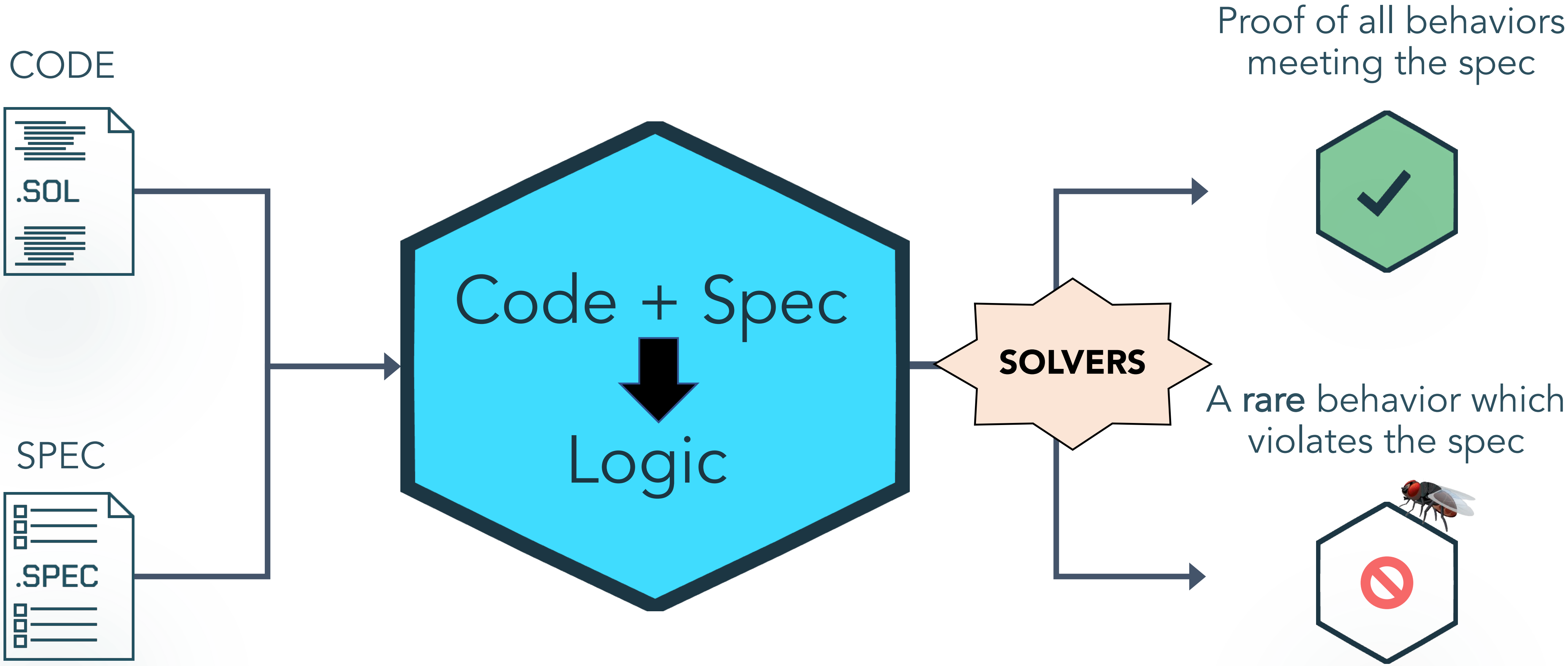
???

CERTORA PROVER

Proof of all behaviors meeting the spec

A **rare** behavior which violates the spec

Formal Verification with Certora Prover



CODE

.SOL

SPEC

.SPEC

Code + Spec

Logic

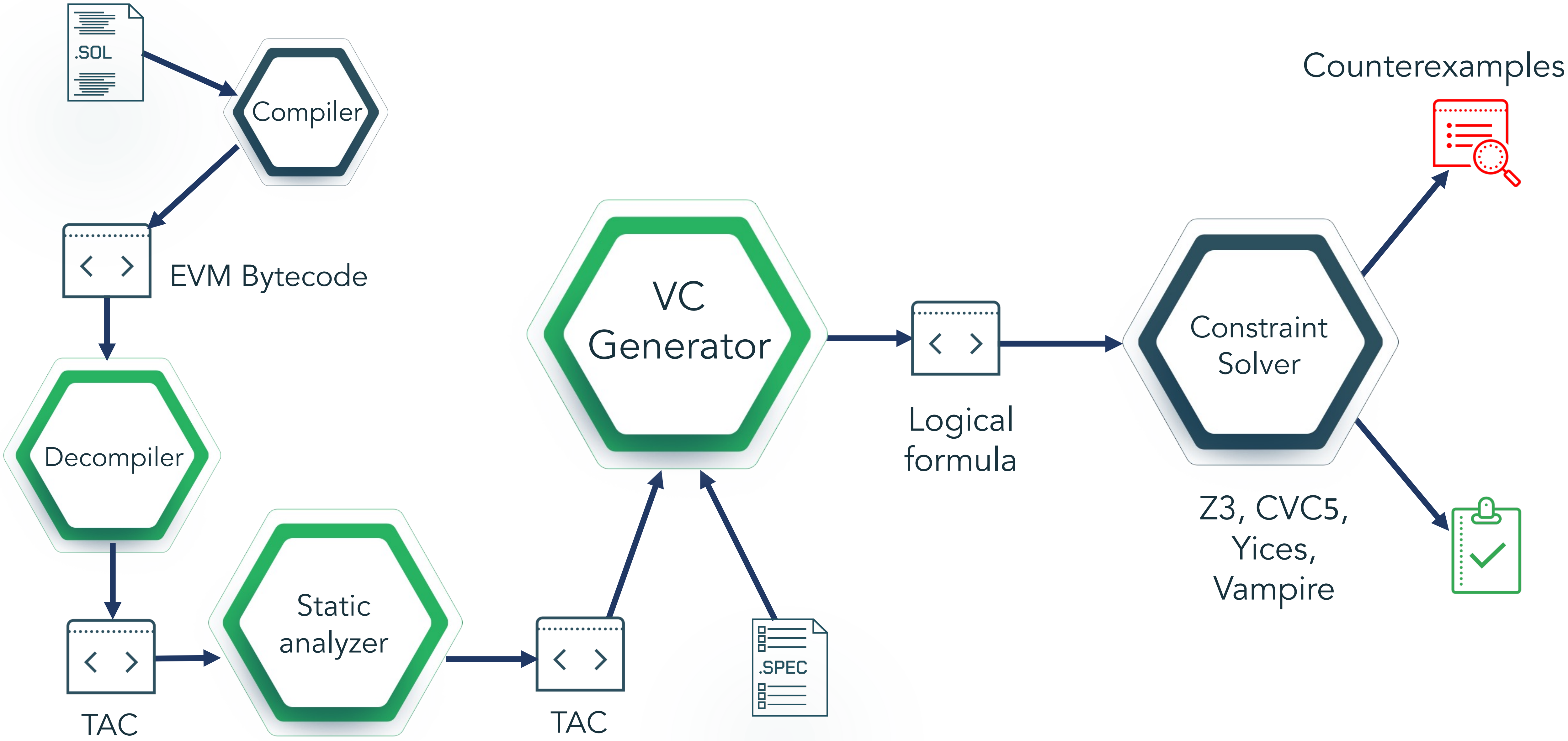
SOLVERS

Proof of all behaviors
meeting the spec

A **rare** behavior which
violates the spec

Certora Prover Architecture

CERTORA


.SOL

Compiler

EVM Bytecode

Decompiler

TAC

Static analyzer

TAC

.SPEC

VC Generator

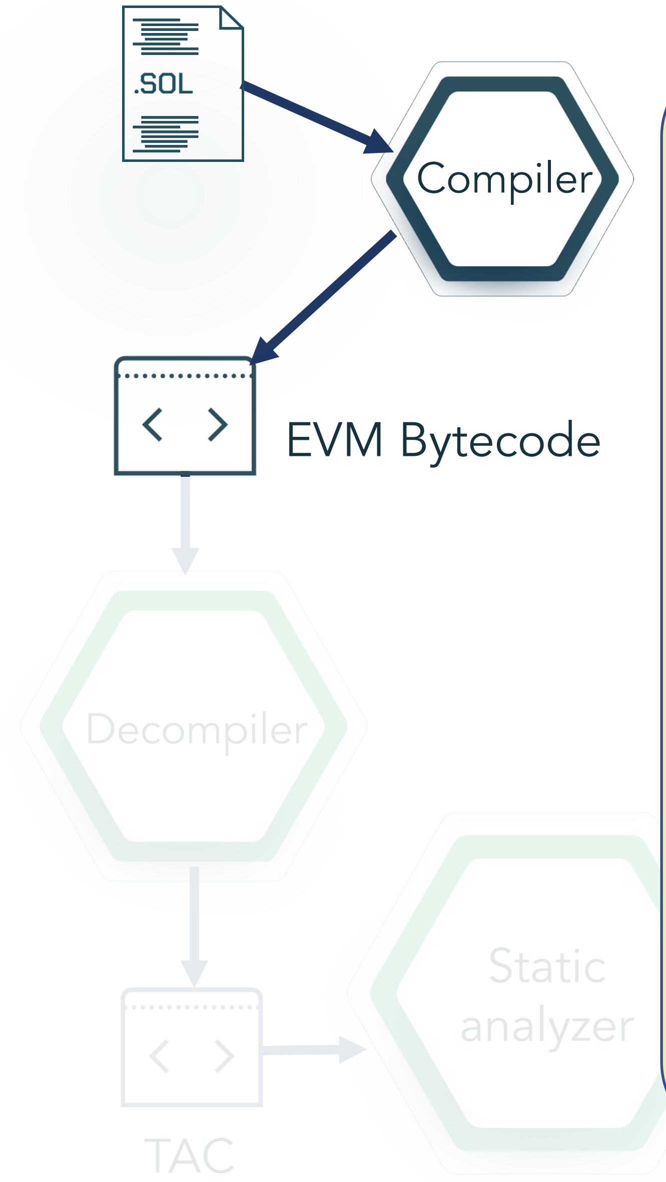
Logical formula

Constraint Solver

Z3, CVC5, Yices, Vampire

Counterexamples

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

Non-deterministic Solidity Transactions — Certora Bug Disclosure

Uri Kirstein

May 31 · 5 min read

The Solidity Compiler Silently Corrupts Storage — Certora Bug Disclosure

Uri Kirstein

May 30 · 4 min read

Memory Isolation Violation in Deserialization Code — Certora Bug Disclosure

Uri Kirstein

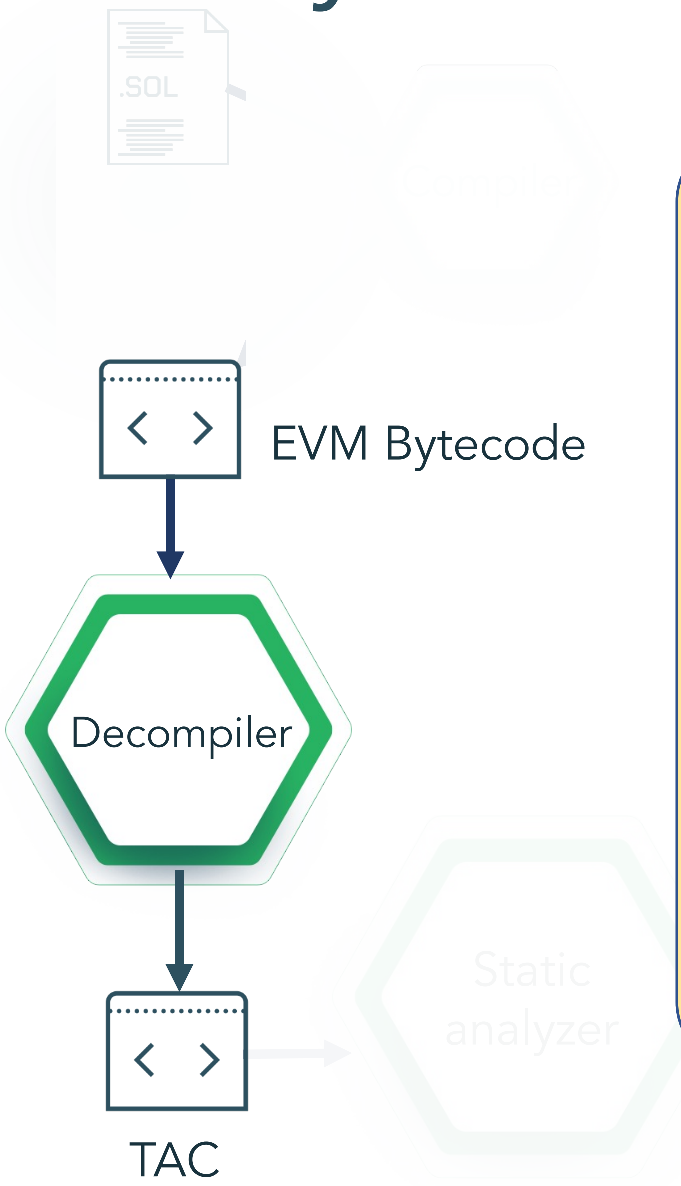
Jun 2 · 4 min read

Bug Disclosure — Solidity Code Generation Bug Can Cause Memory Corruption

Uri Kirstein

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

EVM Bytecode

Decompiler

TAC

```
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!ce4604a000000000000000000000001[R65]
R76 = R35+R68
tacS!ce4604a000000000000000000000001[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)

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)

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)

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)

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)

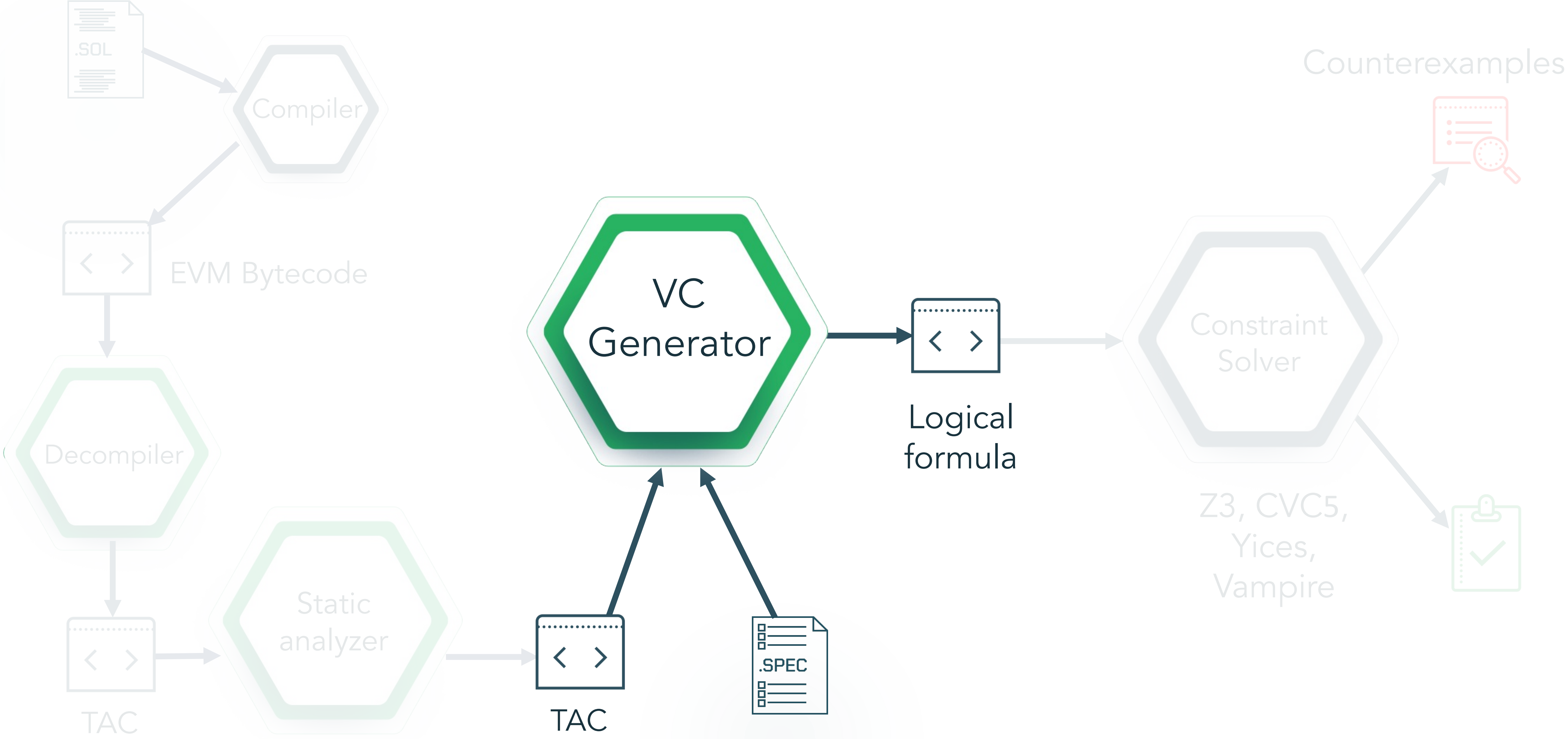
Lower burden on subsequent steps in the pipeline

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

Static Analysis on TAC Cont.

Analysis Type		Example Application
Points-to analysis	-->	reveals connections between TAC variables
Value range analysis	-->	allows us to simplify SMT axioms
Control-flow analysis	-->	split the original program into several smaller programs

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

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 ⇒ WP(S, Q) is valid</u>

Thus, if P ⇒ WP(S, Q) is valid then {P} S {Q}

Weakest Precondition Computation

Basic instructions:

- Assertions: WP(assert A, B) = $A \wedge B$
- Assumptions: WP(assume A, B) = $A \Longrightarrow B$
- Assignments = assumptions!
- Sequential composition: WP(S;T, B) = WP(S, WP(T, B))
- Choice statements: WP(S[]T, B) = WP(S, B) $\wedge$ WP(T, B)

Weakest Precondition Computation

Basic instructions:

- Assertions: WP(assert A, B) = A ∧ B
- Assumptions: WP(assume A, B) = A ⟹ B
- Assignments = assumptions!
- Sequential composition: WP(S;T, B) = WP(S, WP(T, B))
- Choice statements: WP(S[]T, B) = WP(S, B) ∧ WP(T, B)

Weakest Precondition Computation

Basic instructions:

- Assertions: WP(assert A, B) = A ∧ B
- Assumptions: WP(assume A, B) = A ⟹ B
- Assignments = assumptions!
- Sequential composition: WP(S;T, B) = WP(S, WP(T, B))
- Choice statements: WP(S[]T, B) = WP(S, B) ∧ WP(T, B)

Weakest Precondition Computation

Basic instructions:

- Assertions: WP(assert A, B) = A ∧ B
- Assumptions: WP(assume A, B) = A ⟹ B
- Assignments = assumptions!
- Sequential composition: WP(S;T, B) = WP(S, WP(T, B))
- Choice statements: WP(S[]T, B) = WP(S, B) ∧ WP(T, B)

Weakest Precondition Computation

Basic instructions:

- Assertions: WP(assert A, B) = A $\wedge$ B
- Assumptions: WP(assume A, B) = A $\Longrightarrow$ B
- Assignments = assumptions!
- Sequential composition: WP(S;T, B) = WP(S, WP(T, B))
- Choice statements: WP(S[]T, B) = WP(S, B) $\wedge$ WP(T, B)

Weakest Precondition Computation

Basic instructions:

- Assertions: WP(assert A, B) = A ∧ B
- Assumptions: WP(assume A, B) = A ⟹ B
- Assignments = assumptions!
- Sequential composition: WP(S;T, B) = WP(S, WP(T, B))
- Choice statements: WP(S[]T, B) = WP(S, B) ∧ WP(T, B)

Weakest Precondition Computation

Basic instructions:

- Assertions: $WP(\text{assert } A, B) = A \wedge B$
- Assumptions: $WP(\text{assume } A, B) = A \Longrightarrow B$
- Assignments = assumptions!
- Sequential composition: $WP(S;T, B) = WP(S, WP(T, B))$
- Choice statements: $WP(S[]T, B) = WP(S, B) \wedge WP(T, B)$

Loops

Unroll specific number of iterations +

1. Either assume loop termination condition, or
2. Assert loop termination condition

Verification Condition

If $P \Rightarrow WP(S, Q)$ is valid formula then the program satisfies the specification

Verification Condition

If $P \Rightarrow WP(S, Q)$ is valid formula then the program satisfies the specification

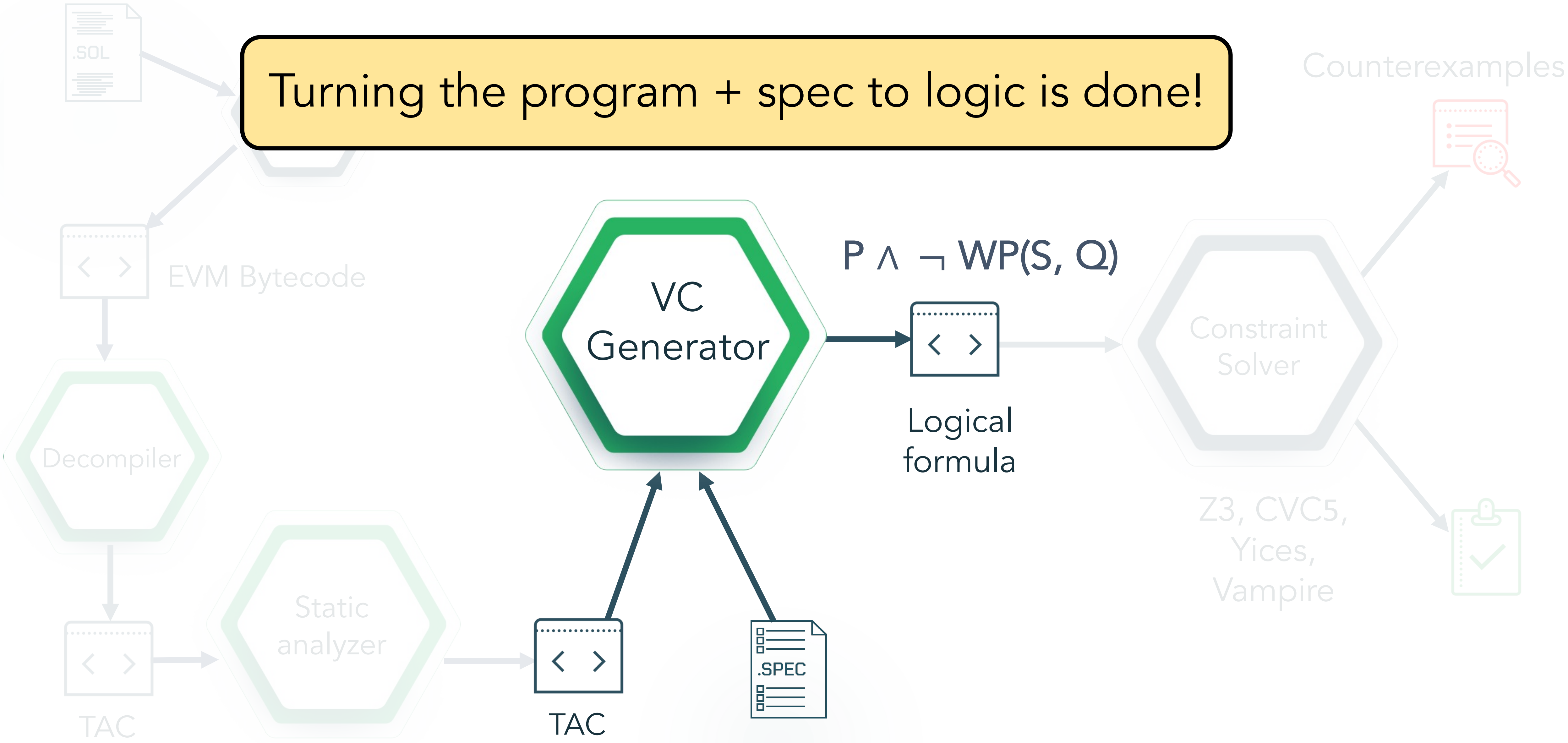
We check $P \land \neg WP(S, Q)$ for satisfiability (not validity!).

- If $P \land \neg WP(S, Q)$ is unsatisfiable then the program satisfies the spec.
- Else, if $P \land \neg WP(S, Q)$ is satisfiable, then the program might violate the spec.

CERTORA

Generate Verification Conditions

Turning the program + spec to logic is done!



.SOL

EVM Bytecode

Decompiler

TAC

Static analyzer

TAC

.SPEC

VC Generator

$P \land \neg WP(S, Q)$

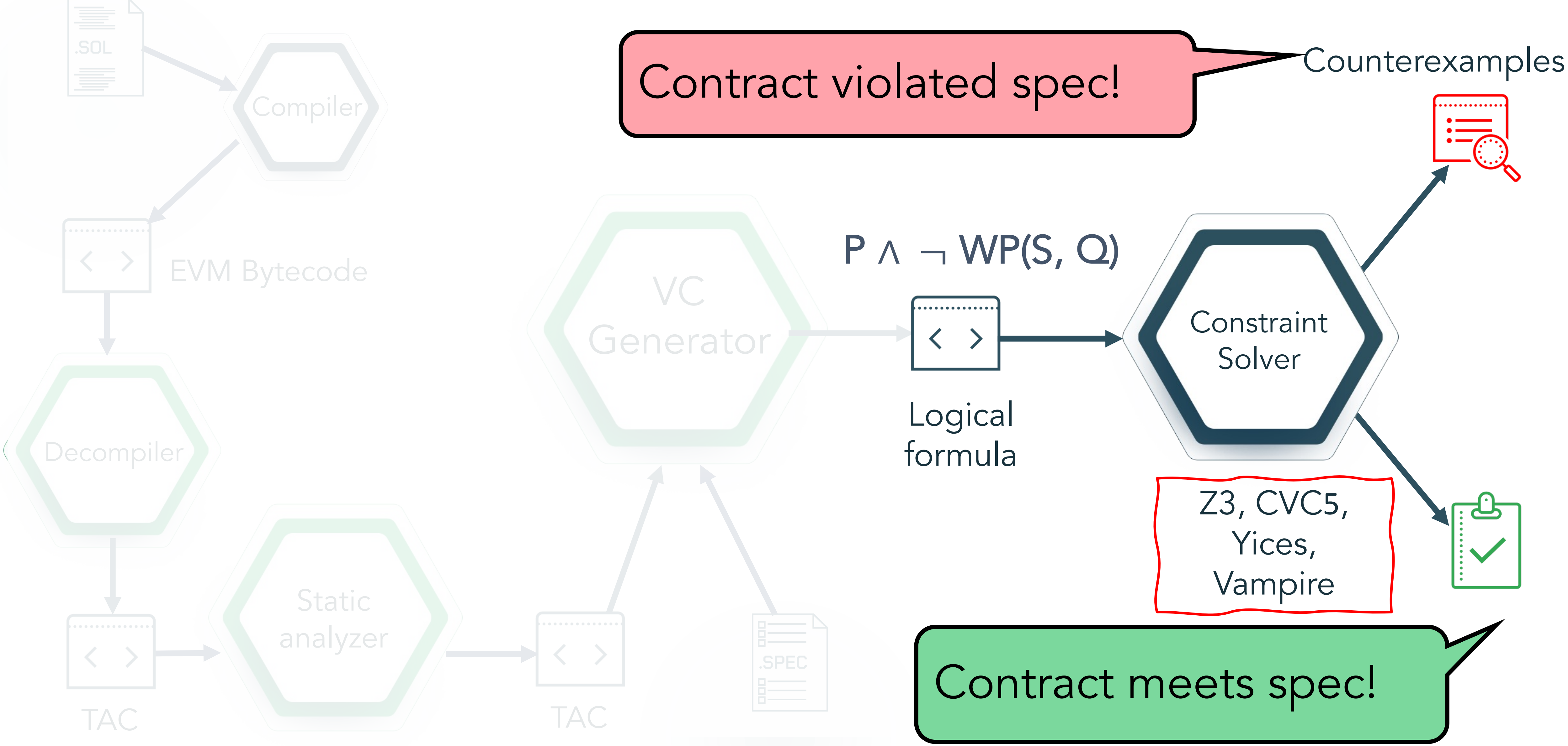
Logical formula

Constraint Solver

Z3, CVC5, Yices, Vampire

Counterexamples

Using Constraint Solvers



.SOL

Compiler

EVM Bytecode

Decompiler

TAC

Static analyzer

TAC

.SPEC

VC Generator

$P \land \neg WP(S, Q)$

Logical formula

Constraint Solver

Z3, CVC5, Yices, Vampire

Counterexamples

Contract violated spec!

Contract meets spec!

SMT Machinery

Encodings of $P \wedge \neg WP(S, Q)$

SMT Machinery

Encodings of $P \wedge \neg WP(S, Q)$

- Precise NIA
- LIA Overraproximation

SMT Machinery

Encodings of $P \wedge \neg WP(S, Q)$

- Precise NIA
- LIA Overraproximation

SMT solvers

- z3, cvc4, cvc5, vampire, yices
- 1-4 configs per solver
- Choosen configurations run in parallel

SMT Machinery

Encodings of $P \wedge \neg WP(S, Q)$

- Precise NIA
- LIA Overraproximation

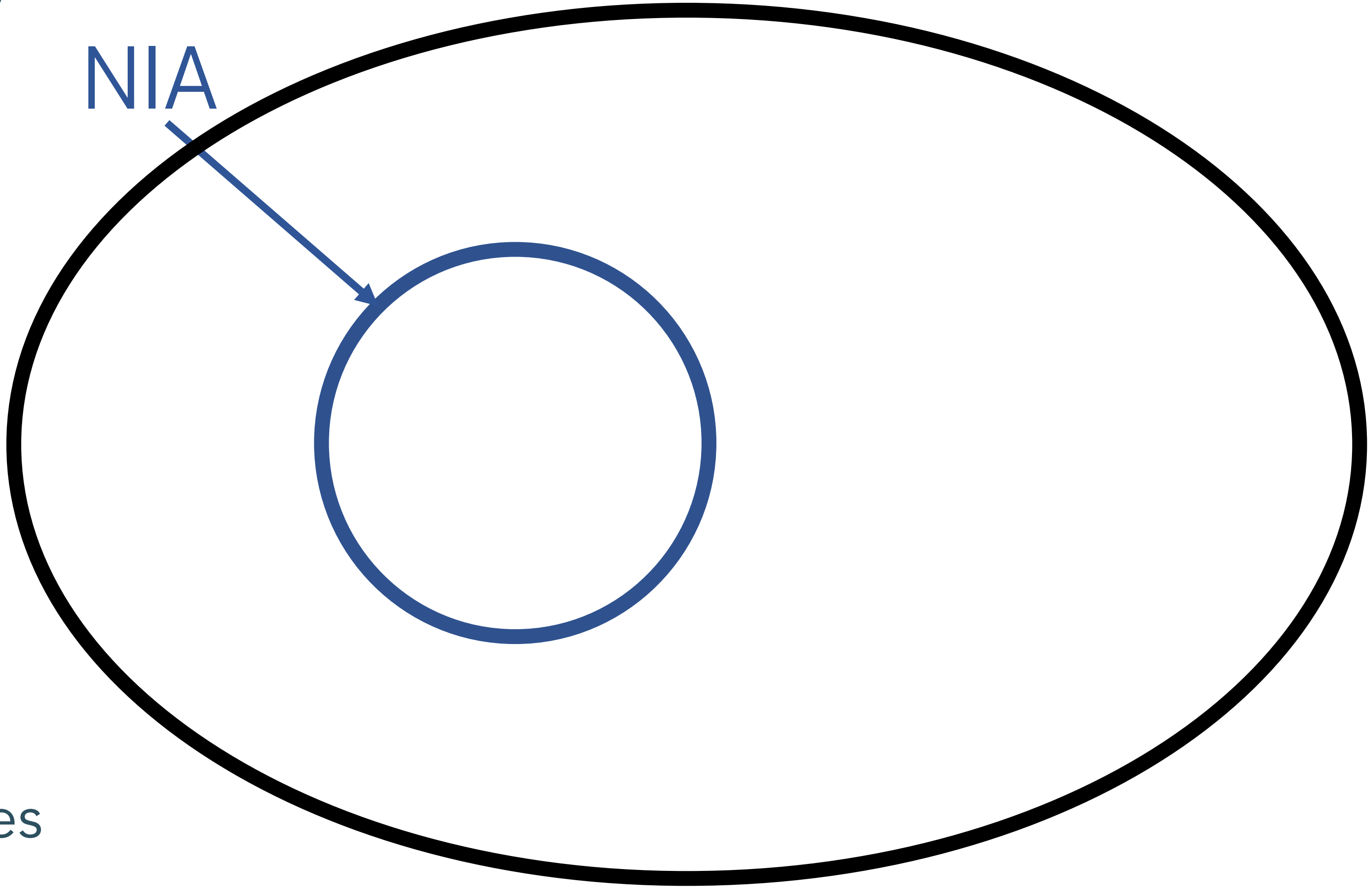
SMT solvers

- z3, cvc4, cvc5, vampire, yices
- 1-4 configs per solver
- Choosen configurations run in parallel

SMT Machinery

Encodings of $P \wedge \neg WP(S, Q)$

- Precise NIA
- LIA Overraproximation



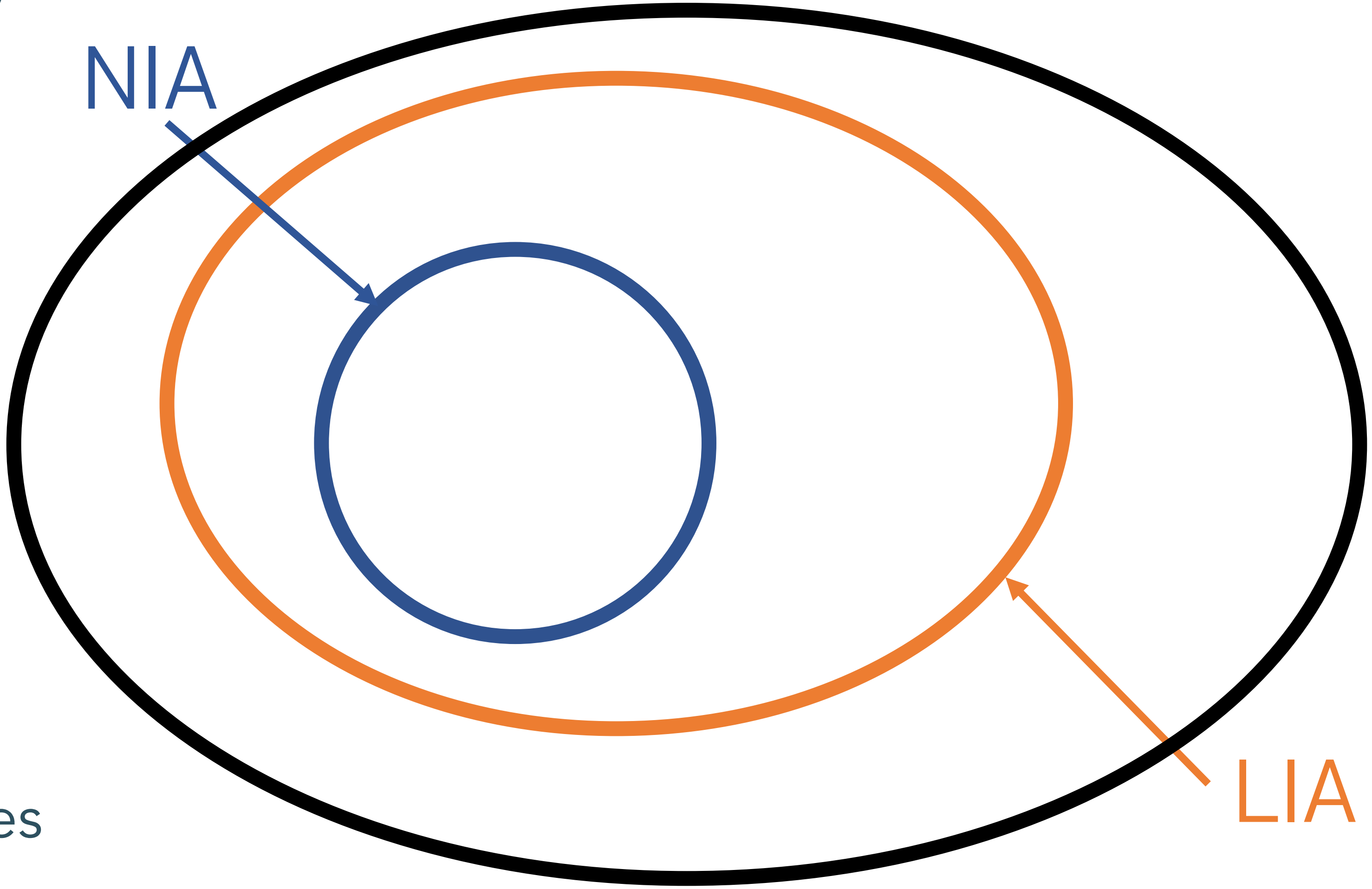
SMT solvers

- z3, cvc4, cvc5, vampire, yices
- 1-4 configs per solver
- Choosen configurations run in parallel

SMT Machinery

Encodings of $P \wedge \neg WP(S, Q)$

- Precise NIA
- LIA Overraproximation



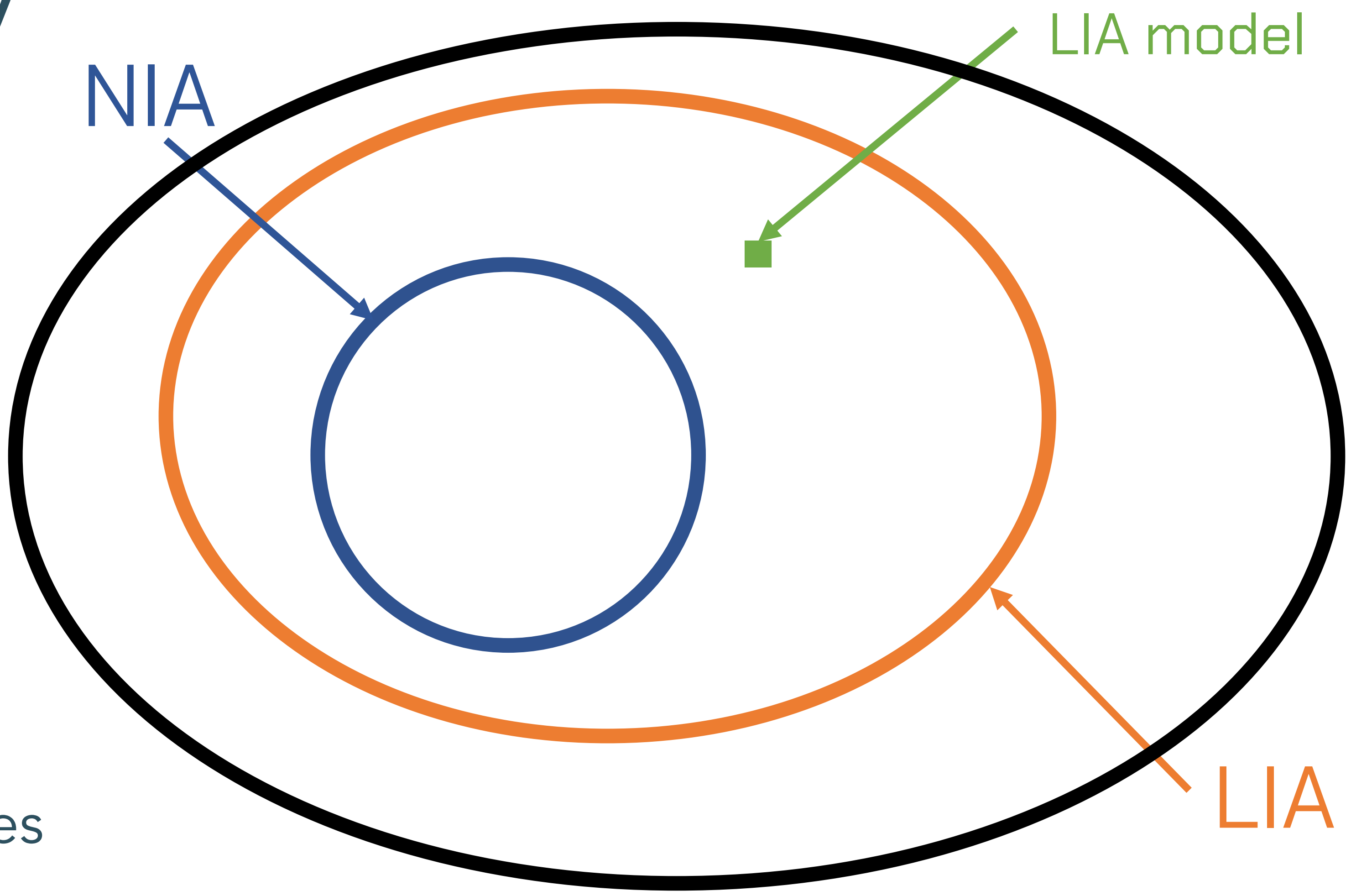
SMT solvers

- z3, cvc4, cvc5, vampire, yices
- 1-4 configs per solver
- Choosen configurations run in parallel

SMT Machinery

Encodings of P ∧ ¬ WP(S, Q)

- Precise NIA
- LIA Overraproximation



SMT solvers

- z3, cvc4, cvc5, vampire, yices
- 1-4 configs per solver
- Choosen configurations run in parallel

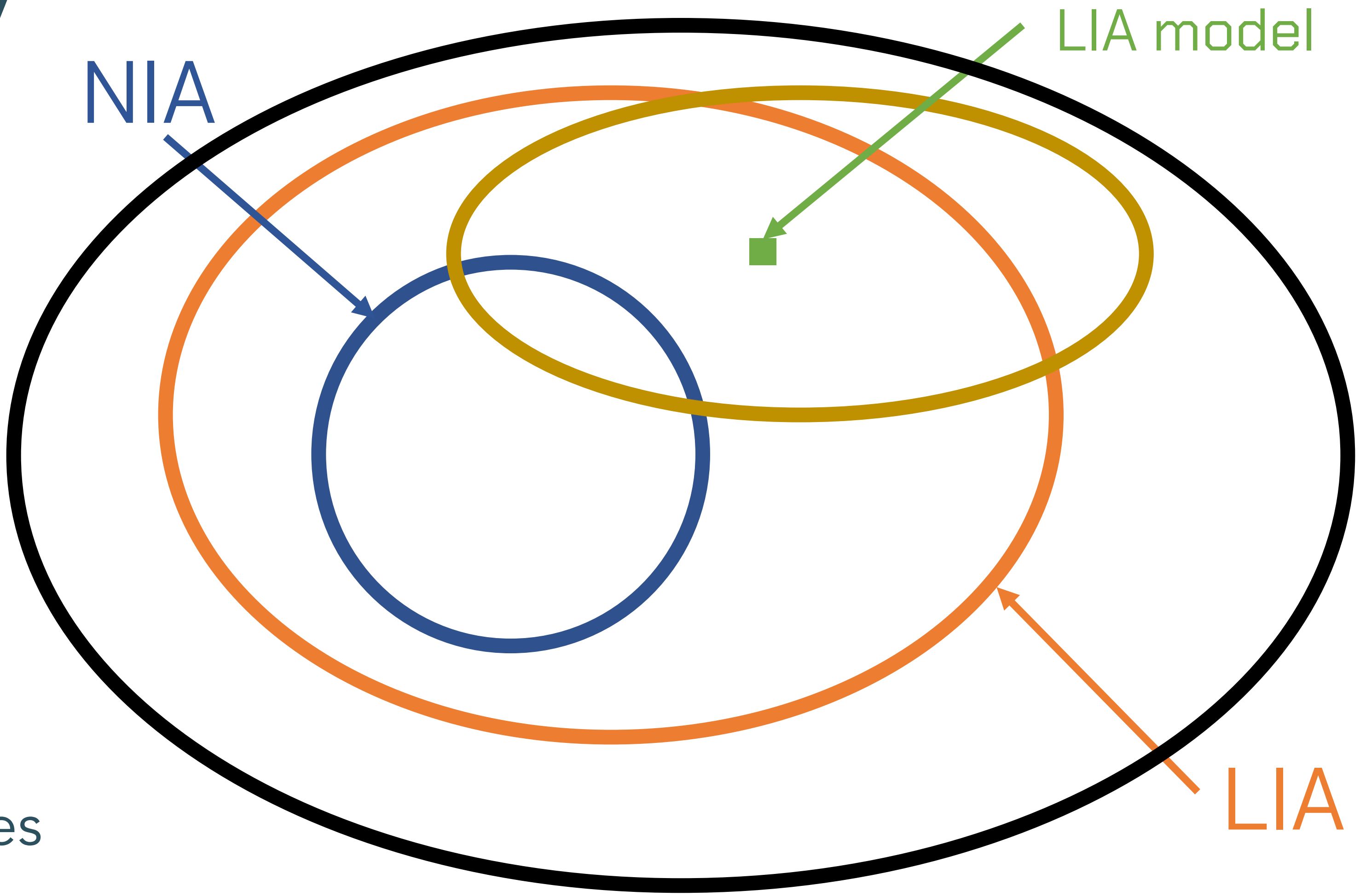
SMT Machinery

Encodings of P ∧ ¬ WP(S, Q)

- Precise NIA
- LIA Overraproximation

SMT solvers

- z3, cvc4, cvc5, vampire, yices
- 1-4 configs per solver
- Choosen configurations run in parallel



LIA Multiplication Axiomatization

a*b modelled with an unintepreted function $a \cdot b$

LIA Multiplication Axiomatization

a*b modelled with an unintepreted function a·b

- $a \cdot 0 = 0$
- $a \cdot b = b \cdot a$
- $a > 0, b > 0 \rightarrow a \cdot b > 0$
- $a > 0, b < 0 \rightarrow a \cdot b < 0$
- $a < 0, b > 0 \rightarrow a \cdot b < 0$
- $a < 0, b < 0 \rightarrow a \cdot b > 0$
- $a > 0, b > 0 \rightarrow a \cdot b >= a, a \cdot b >= b$
- $0 <= a1 <= a2, 0 <= b1 <= b2 \rightarrow a1 \cdot b1 <= a2 \cdot b2$

CERTORA

LIA Multiplication Axiomatization

a*b modelled with an unintepreted function a∙b

- a∙0 = 0
- a∙b = b∙a
- a > 0, b > 0 → a∙b > 0
- a > 0, b < 0 → a∙b < 0
- a < 0, b > 0 → a∙b < 0
- a < 0, b < 0 → a∙b > 0
- a > 0, b > 0 → a∙b >= a, a∙b >= b
- 0 <= a1 <= a2, 0 <= b1 <= b2 → a1∙b1 <= a2∙b2

LIA Multiplication Axiomatization

a*b modelled with an unintepreted function a·b

- $a \cdot 0 = 0$
- $a \cdot b = b \cdot a$
- $a > 0, b > 0 \rightarrow a \cdot b > 0$
- $a > 0, b < 0 \rightarrow a \cdot b < 0$
- $a < 0, b > 0 \rightarrow a \cdot b < 0$
- $a < 0, b < 0 \rightarrow a \cdot b > 0$
- $a > 0, b > 0 \rightarrow a \cdot b >= a, a \cdot b >= b$
- $0 <= a1 <= a2, 0 <= b1 <= b2 \rightarrow a1 \cdot b1 <= a2 \cdot b2$

LIA Multiplication Axiomatization

a*b modelled with an unintepreted function a·b

- $a \cdot 0 = 0$
- $a \cdot b = b \cdot a$
- $a > 0, b > 0 \rightarrow a \cdot b > 0$
- $a > 0, b < 0 \rightarrow a \cdot b < 0$
- $a < 0, b > 0 \rightarrow a \cdot b < 0$
- $a < 0, b < 0 \rightarrow a \cdot b > 0$
- $a > 0, b > 0 \rightarrow a \cdot b >= a, a \cdot b >= b$
- $0 <= a1 <= a2, 0 <= b1 <= b2 \rightarrow a1 \cdot b1 <= a2 \cdot b2$

LIA Multiplication Axiomatization

a*b modelled with an uninterpreted function a·b

- $a \cdot 0 = 0$
- $a \cdot b = b \cdot a$
- $a > 0, b > 0 \rightarrow a \cdot b > 0$
- $a > 0, b < 0 \rightarrow a \cdot b < 0$
- $a < 0, b > 0 \rightarrow a \cdot b < 0$
- $a < 0, b < 0 \rightarrow a \cdot b > 0$
- $a > 0, b > 0 \rightarrow a \cdot b >= a, a \cdot b >= b$
- $0 <= a1 <= a2, 0 <= b1 <= b2 \rightarrow a1 \cdot b1 <= a2 \cdot b2$

LIA Multiplication Axiomatization

a*b modelled with an uninterpreted function $a \cdot b$

- $a \cdot 0 = 0$
- $a \cdot b = b \cdot a$
- $a > 0, b > 0 \rightarrow a \cdot b > 0$
- $a > 0, b < 0 \rightarrow a \cdot b < 0$
- $a < 0, b > 0 \rightarrow a \cdot b < 0$
- $a < 0, b < 0 \rightarrow a \cdot b > 0$
- $a > 0, b > 0 \rightarrow a \cdot b >= a, a \cdot b >= b$
- $0 <= a1 <= a2, 0 <= b1 <= b2 \rightarrow a1 \cdot b1 <= a2 \cdot b2$

Learned Literals

Given a formula F, an SMT solver says:

- F is SAT, or
- F is UNSAT, or
- timeout

Learned Literals

Given a formula F, an SMT solver says:

- F is SAT, or
- F is UNSAT, or
- timeout, but learned F => L for some L

Learned Literals

Given a formula F, an SMT solver says:

- F is SAT, or
- F is UNSAT, or
- timeout, but learned F => L for some L

Example

$$
\begin{aligned}
L \equiv\ & (x = 5) \land \\
& (y <= 10 \lor y > 20) \land \\
& (y < 100) \land \\
& (z = x \lor z > 10)
\end{aligned}
$$

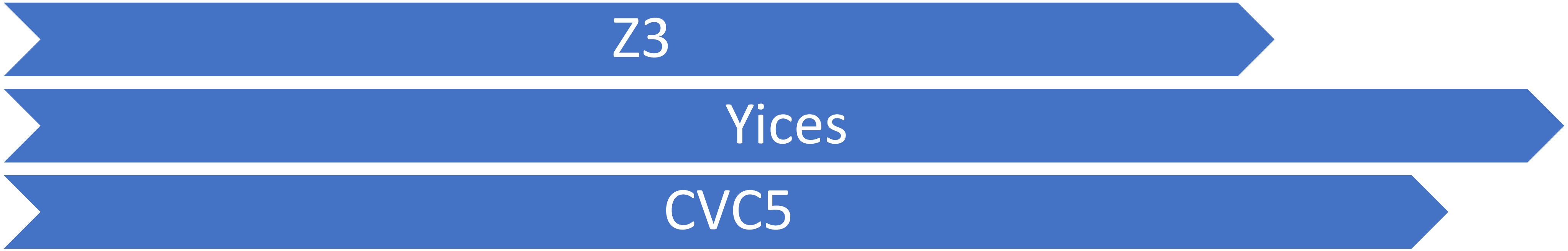
Learned Literals

Given a formula F, an SMT solver says:

- F is SAT, or
- F is UNSAT, or
- timeout, but learned F => L for some L

Example

$$
\begin{aligned}
L \equiv\ & (x = 5) \land \\
& (y <= 10 \lor y > 20) \land \\
& (y < 100) \land \\
& (z = x \lor z > 10)
\end{aligned}
$$



Z3

Yices

CVC5

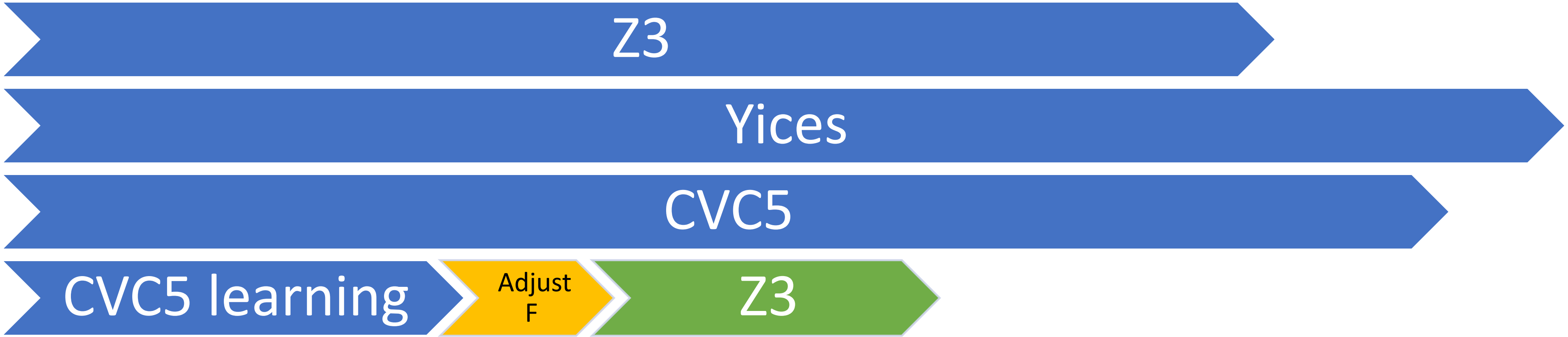
Learned Literals

Given a formula F, an SMT solver says:

- F is SAT, or
- F is UNSAT, or
- timeout, but learned F => L for some L

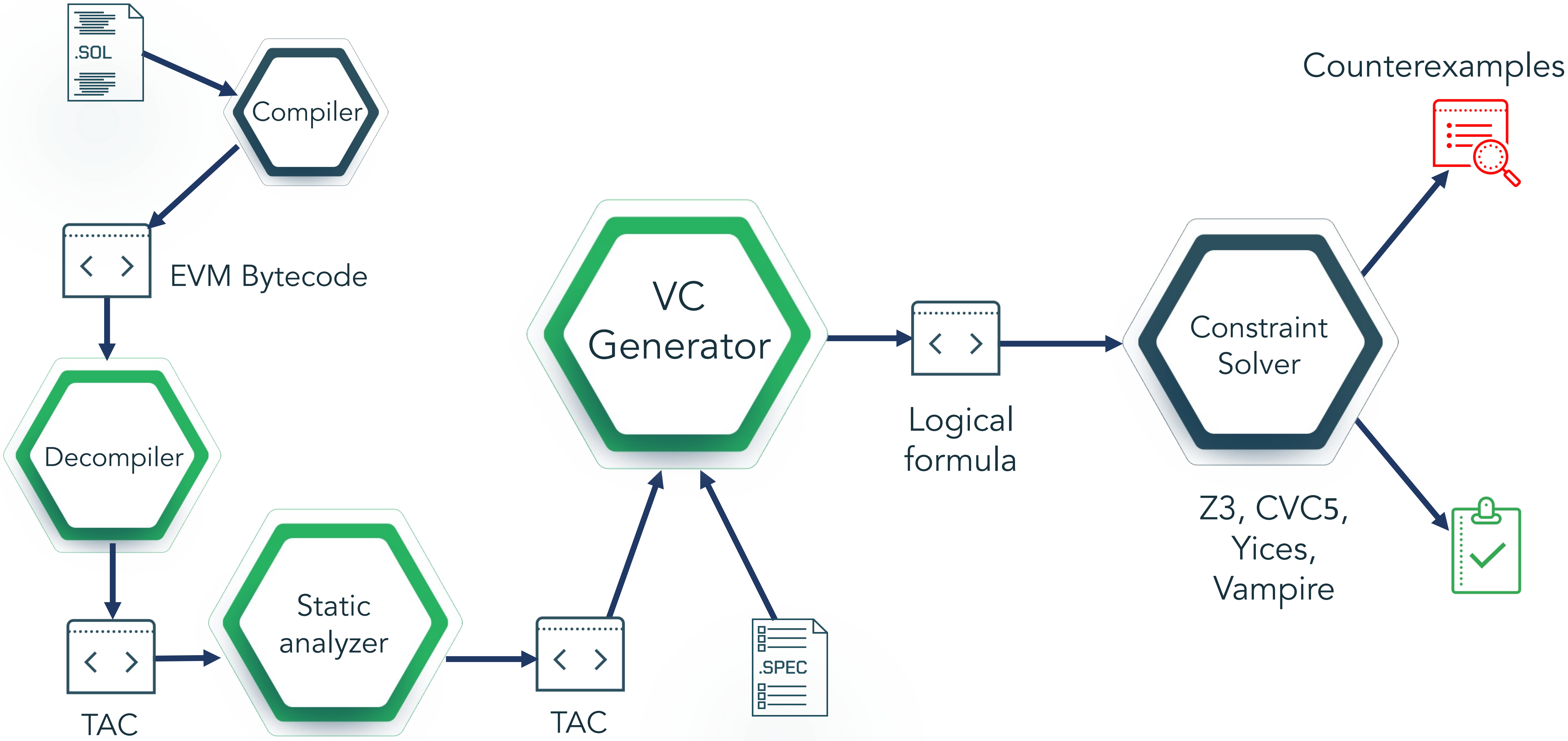
Example

$$
\begin{aligned}
L \equiv\ & (x = 5) \land \\
& (y <= 10 \lor y > 20) \land \\
& (y < 100) \land \\
& (z = x \lor z > 10)
\end{aligned}
$$



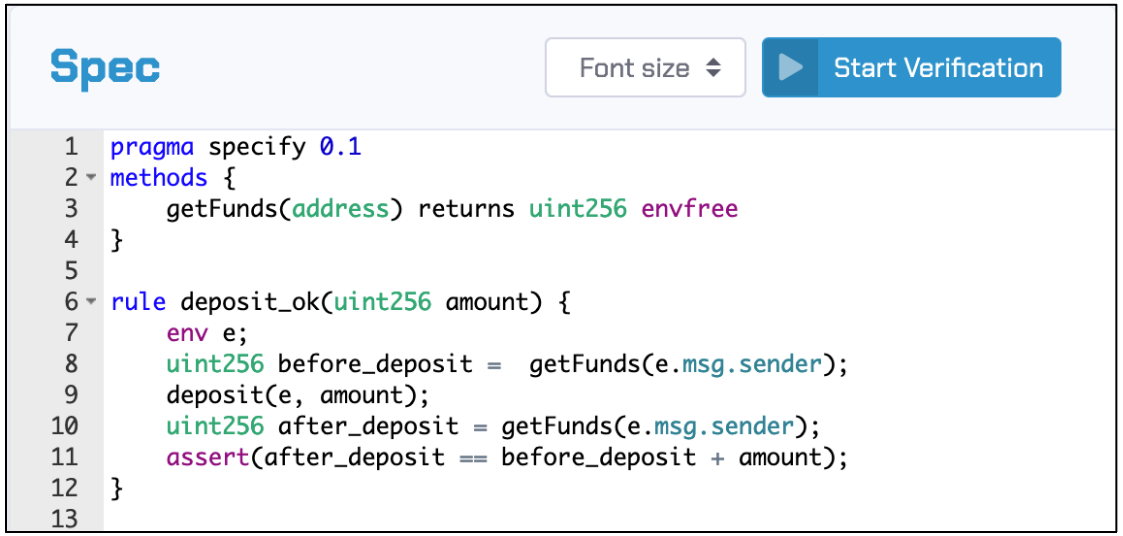
The Certora Prover Pipeline

CERTORA

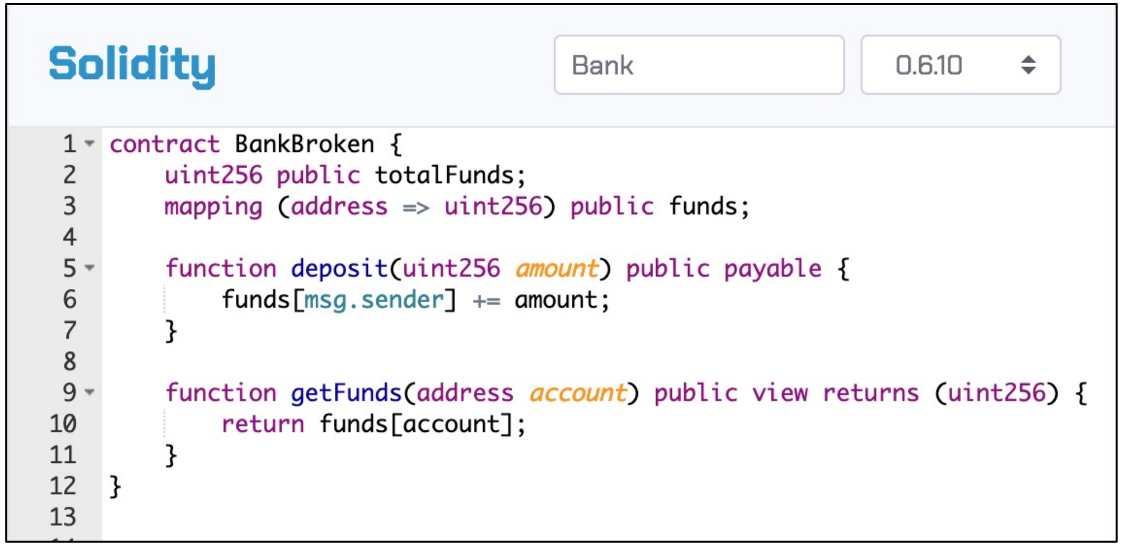


Putting It All Together

CERTORA



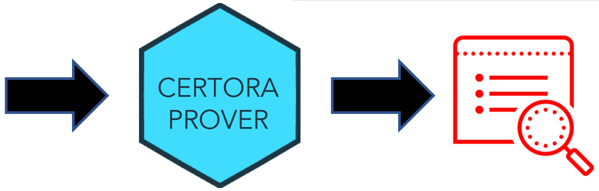
```
pragma specify 0.1
methods {
    getFunds(address) returns uint256 envfree
}

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);
}
```

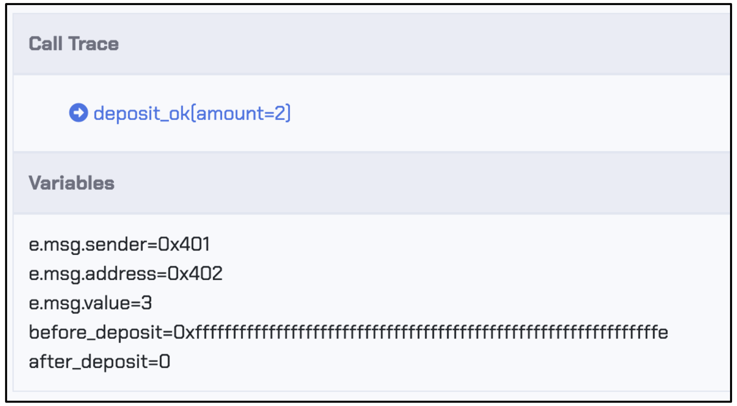


```
contract BankBroken {
    uint256 public totalFunds;
    mapping (address => uint256) public funds;

    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!





deposit_ok(amount=2)

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

Certora Inc.

- Founded in 2019
- 60 software engineers including 13 PhDs
- Offices in Tel Aviv and Seattle
- Teams:
    - Static analysis
    - SMT
    - Frontend
    - Rulewriters
    - Fuzzing and mutation testing
    - Security Engineers (white hat hackers)

CERTORA

Certora Inc.

- Founded in 2019
- 60 software engineers including 13 PhDs
- Offices in Tel Aviv and Seattle
- Teams:
    - Static analysis
    - SMT
    - Frontend
    - Rulewriters
    - Fuzzing and mutation testing
    - Security Engineers (white hat hackers)

WE ARE HIRING

full time, part time, internship

(contact me, jaroslav@certora.com,
or see https://www.certora.com)