Deductive Verification of Smart Contracts

Mooly Sagiv

TEL AVIV UNIVERSITY אוניברסיטת תל אביב

vmware®

CERTORA

OCTOBER 2018

Software verification

- The programmer defines what is the desired behavior
- Ensures there is a proof of correctness
- Proof covers all scenarios

Why verify smart contracts?

Smart Contracts are hard to get right

alex van de sande
@avsa

I repeat. There was an attack on the DAO so we launched our white hat counter attack. More updates will follow

RETWEETS 46 LIKES 51

8:11 PM - 21 Jun 2016

Manuel Aráoz
@maraoz

Someone stole ~$32M (~153k ether) from three multisig wallets. More info and blog post coming soon. etherscan.io/address/0xb376 …

12:08 PM - 19 Jul 2017

Parity Technologies
@ParityTech

UPDATE: A user exploited an issue and thus removed the library code, as it seems unaware of the consequences.

2:51 AM - 7 Nov 2017

SpankChain
@SpankChain

At 6pm PST Saturday, an unknown attacker drained 165.38 ETH (~$38k) from our payment channel smart contract which also resulted in $4,000 worth of BOOTY on the contract becoming immobilized. Here is what we know so far:

We Got Spanked: What We Know So Far – SpankChain – M…
At 6pm PST Saturday, an unknown attacker drained 165.38 ETH (~$38,000) from our payment channel smart contract which also resulted in…
medium.com

8:49 PM - 8 Oct 2018

A barrier to trust!

Correctness is essential for Smart Contracts

Traditional software

- Buggy code is a reality
- Mechanisms for reverting effects of erroneous code execution
- Continuous code maintenance is standard practice

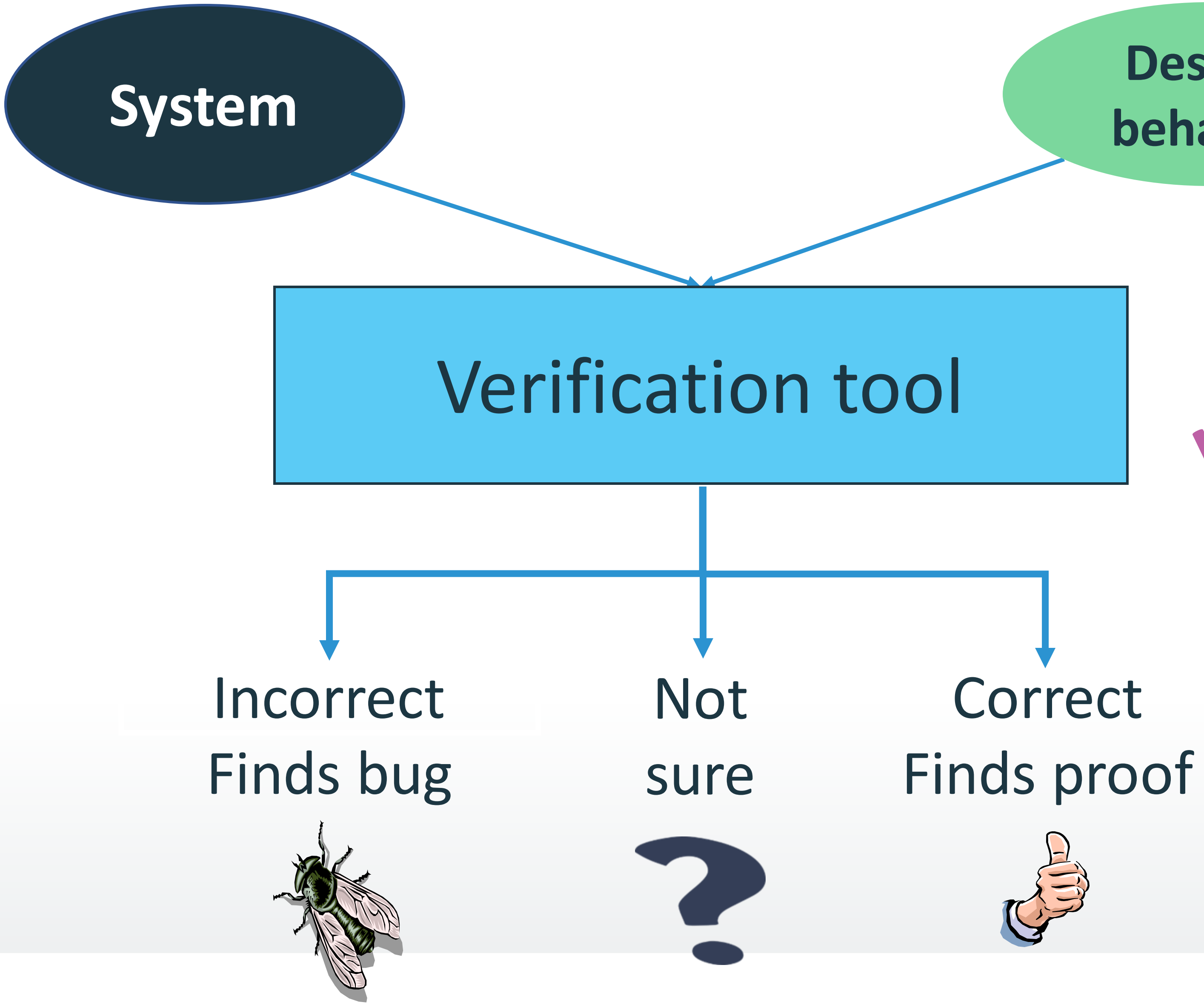
Smart Contracts on Blockchains

- Code as law
- Transactions are irreversible, often anonymous
- Smart Contracts are unpatchable
    - Upgrade is tricky

Auditing

- The standard procedure for checking contracts
- Expensive $$$
- Quality depends on the auditors
- Miss bugs
- Not decentralized
  - Auditor reputation is the trust authority

Automatic deductive verification



System

Desired behavior

Verification tool

Incorrect
Finds bug

Not
sure

Correct
Finds proof

UNDECIDABILITY ☹

What We Do: Technology for certifying contracts

Find bugs or prove their absence

- No false alarms or missed errors

Define what is required from contracts

- Generic properties
  - No overflow
  - Isolation between contracts [POPL’18]

- Standard requirements
  - ERC20, ERC721
    Money market, Exchanges…

- Contract-specific correctness
  - Wallet should have sufficient number of signers
  - Correct libraries

[POPL’18] S. Grossman et. al. [Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts]()

What Customers Say

“Compound worked with Certora to verify the correctness of a preliminary-version of a core contract.

The tool demonstrated a unique capability to discover not just the obvious corner-cases, but also subtle cases that would have been difficult, if not impossible, to find through standard unit-testing.

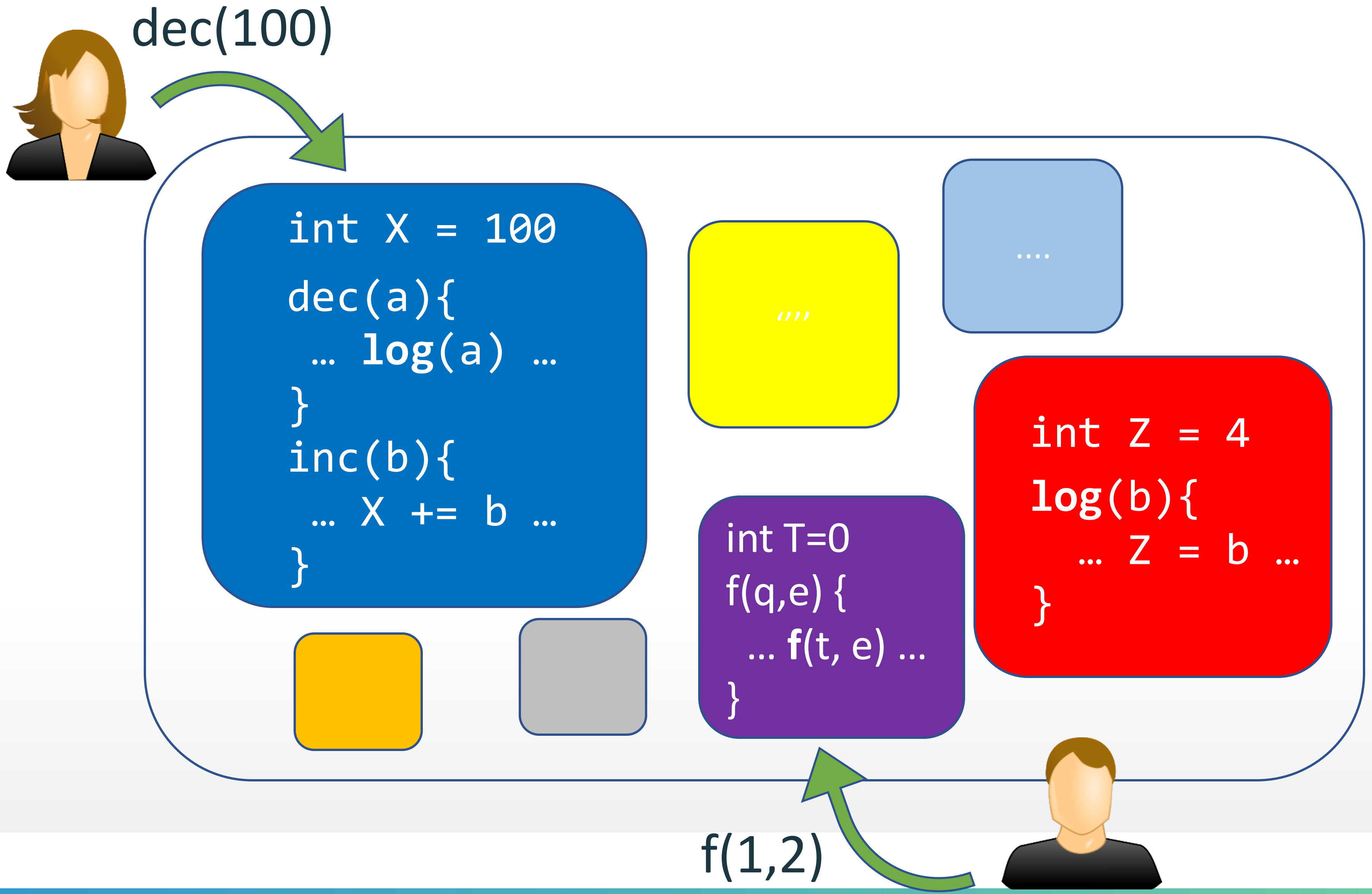
The Certora team discovered two subtle bugs in the contract which were patched, as well definitively proving a conjecture which influenced an important design decision.

Certora's collection of properties proven to hold for all inputs and environments greatly increased our confidence in the correctness of our contract.”

Geoff Hayes | CTO **Compound**

Modular Systems

Ethereum



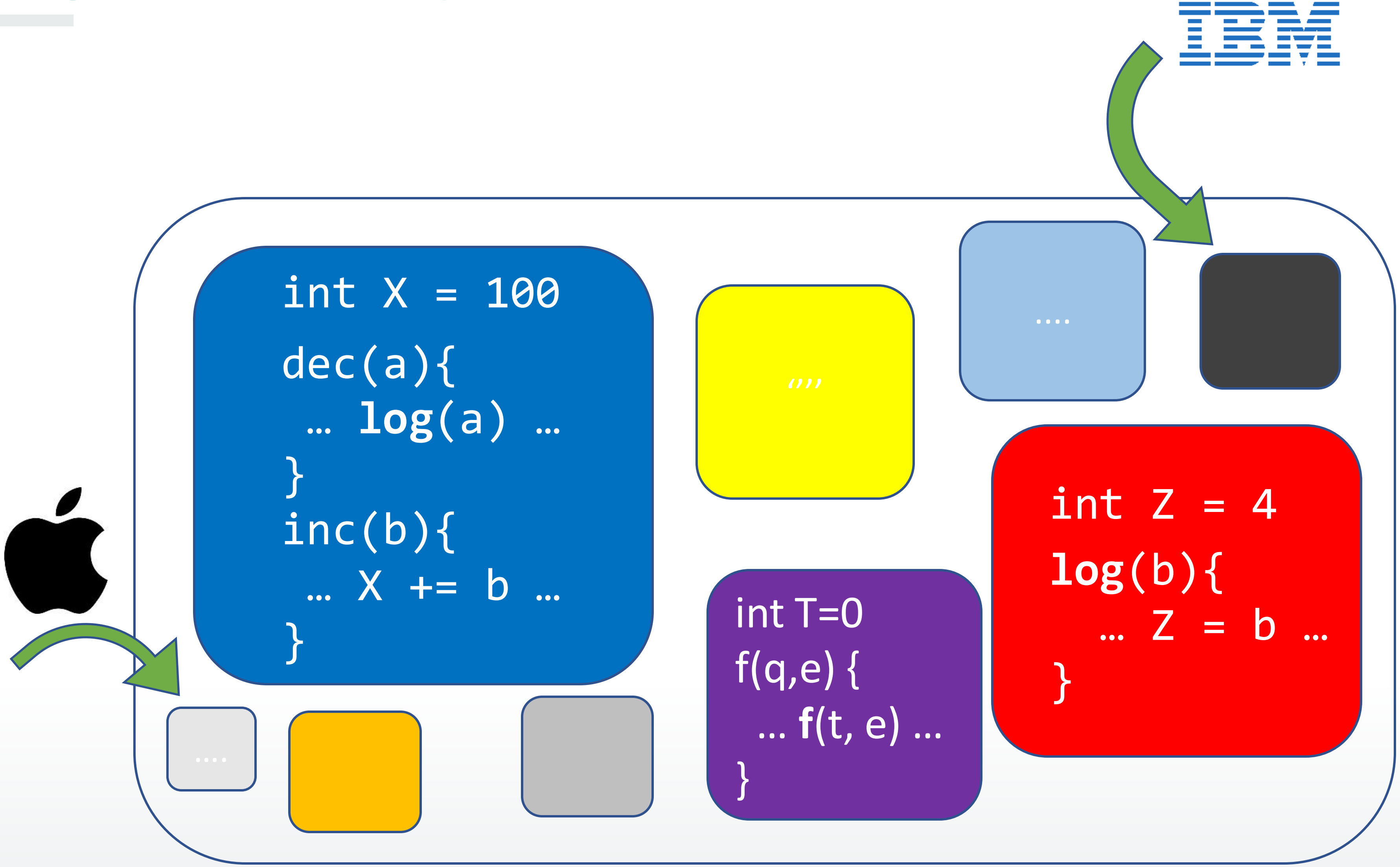
dec(100)

```
int X = 100
dec(a){
 … log(a) …
}
inc(b){
 … X += b …
}
```

~~~

….

```
int Z = 4
log(b){
  … Z = b …
}
```

```
int T=0
f(q,e) {
  … f(t, e) …
}
```

f(1,2)
~~~

Evolving Modular Systems

Ethereum



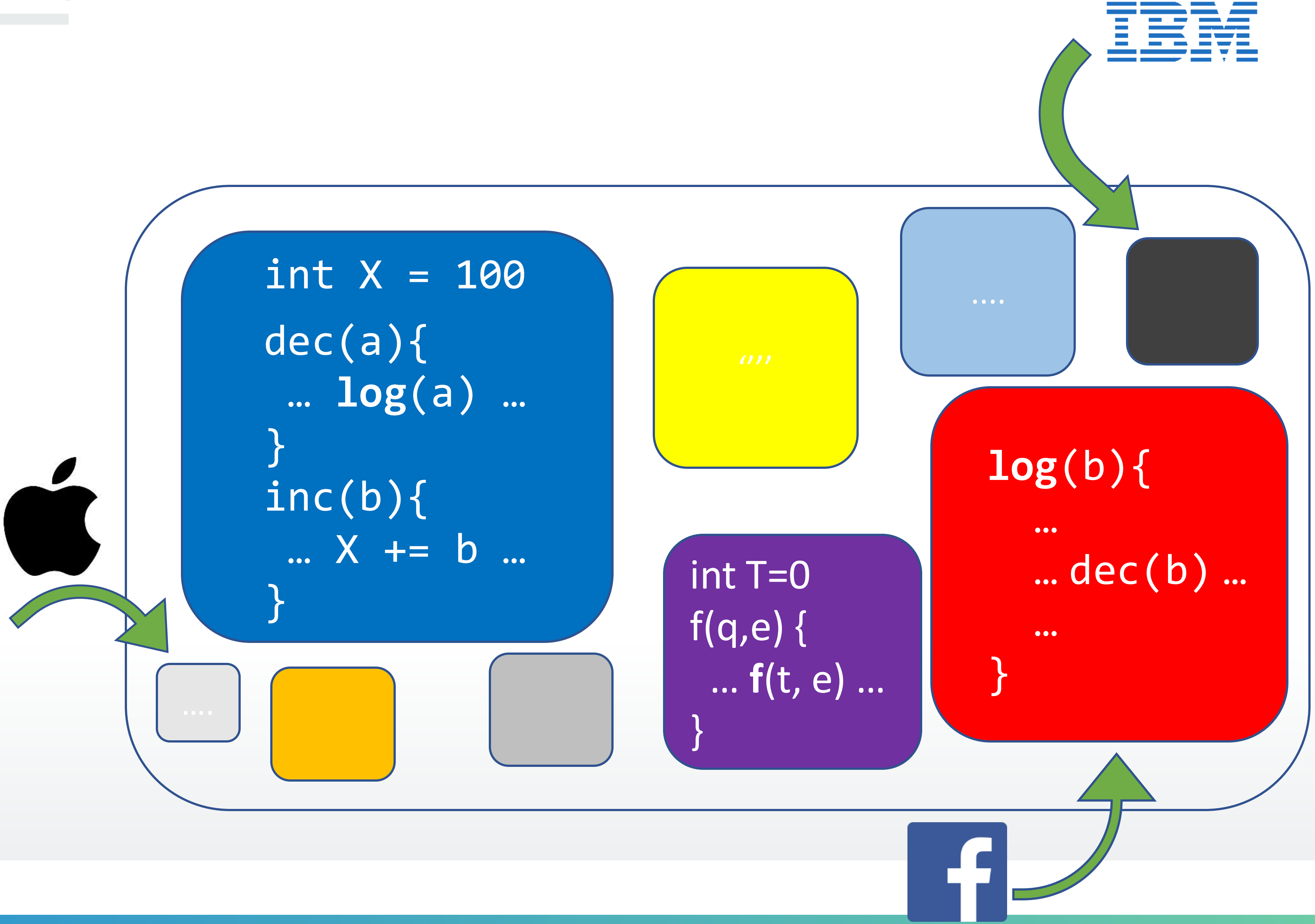
```
int X = 100
dec(a){
 … log(a) …
}
inc(b){
 … X += b …
}
```

```
int T=0
f(q,e) {
 … f(t, e) …
}
```

```
int Z = 4
log(b){
  … Z = b …
}
```

Evolving Modular Systems

Ethereum



```
int X = 100
dec(a){
 … log(a) …
}
inc(b){
 … X += b …
}
```

```
int T=0
f(q,e) {
  … f(t, e) …
}
```

```
log(b){
   …
   … dec(b) …
   …
}
```

Encapsulation → Isolation?

```
module Blue
  int X := 100

  void dec(a)
    requires X >= 0
    if (X >= a)
      log(a)
      X := X - a
    ensures X >= 0
```

Encapsulation → Isolation?

```
module Blue
 int X := 100

 void dec(a)
  requires X >= 0
  if (X >= a)
   ~~log(a)~~
   X := X - a
  ensures X >= 0
```

Encapsulation → Isolation?

```
module Blue
  int X := 100
  void dec(a)
    requires X >= 0
    if (X >= a)
      ~~log(a)~~
      X := X - a
    ensures X >= 0
```

$X \geq 0$

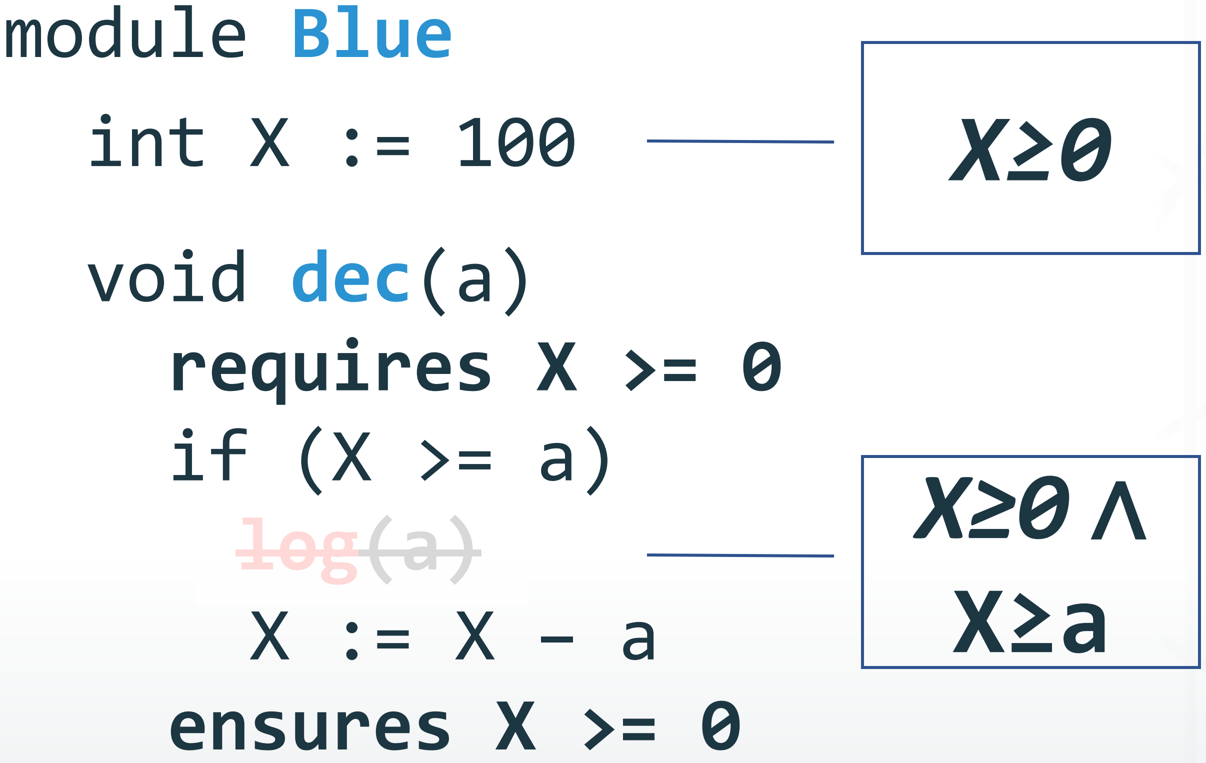
Encapsulation → Isolation?

```
module Blue
   int X := 100
   void dec(a)
      requires X >= 0
      if (X >= a)
         ~~log(a)~~
         X := X - a
      ensures X >= 0
```

$X \geq 0$

$X \geq 0 \land X \geq a$

Encapsulation → Isolation?



```
module Blue
   int X := 100
   void dec(a)
      requires X >= 0
      if (X >= a)
         ~~log(a)~~
         X := X - a
      ensures X >= 0
```

$X \geq 0$

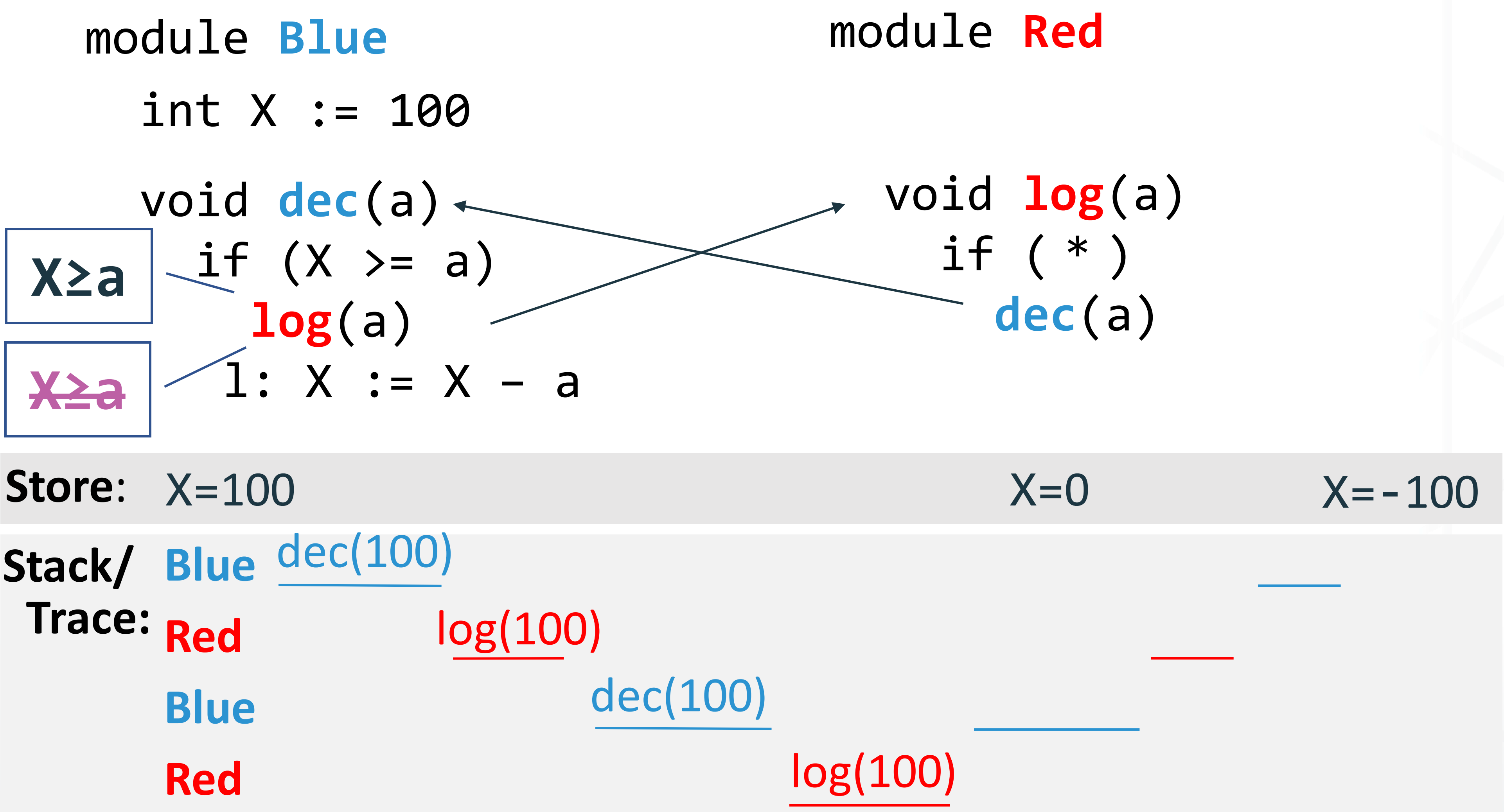
$X \geq 0 \wedge X \geq a$

Encapsulation ↛ Isolation

```
module Blue
  int X := 100

  void dec(a)
    requires X >= 0
    if (X >= a)
      log(a)
      X := X - a
    ensures X >= 0
```

```
module Red


  void log(a)
    if ( * )
      dec(a)
```

Encapsulation ↛ Isolation



```
module Blue
  int X := 100

  void dec(a)
    if (X >= a)
      log(a)
    l: X := X - a
```

```
module Red

  void log(a)
    if ( * )
      dec(a)
```

Store: X=100 X=0 X=-100

Stack/ Trace:

Blue dec(100)

Red log(100)

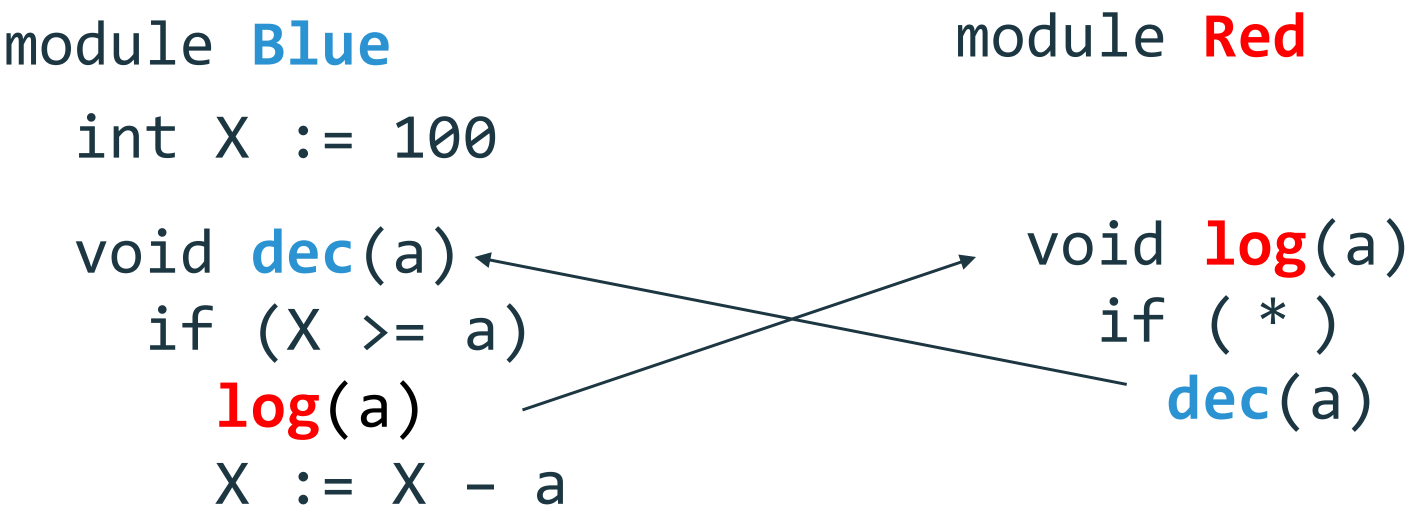
Blue dec(100)

Red log(100)

Effective Callback Freedom (ECF)

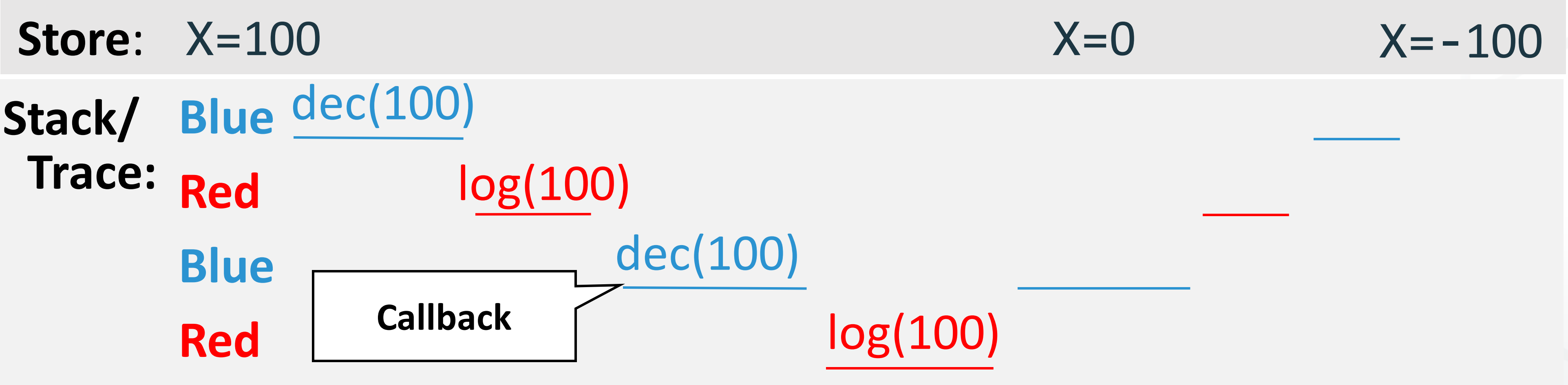
- Correctness condition for modularity
- Tames callbacks – do not add behaviors
- Enforces isolation between contracts
- Not syntactic
- Does not preclude good behaviors

A Non ECF Execution for `Blue`



```
module Blue
   int X := 100

   void dec(a)
     if (X >= a)
       log(a)
       X := X - a
```

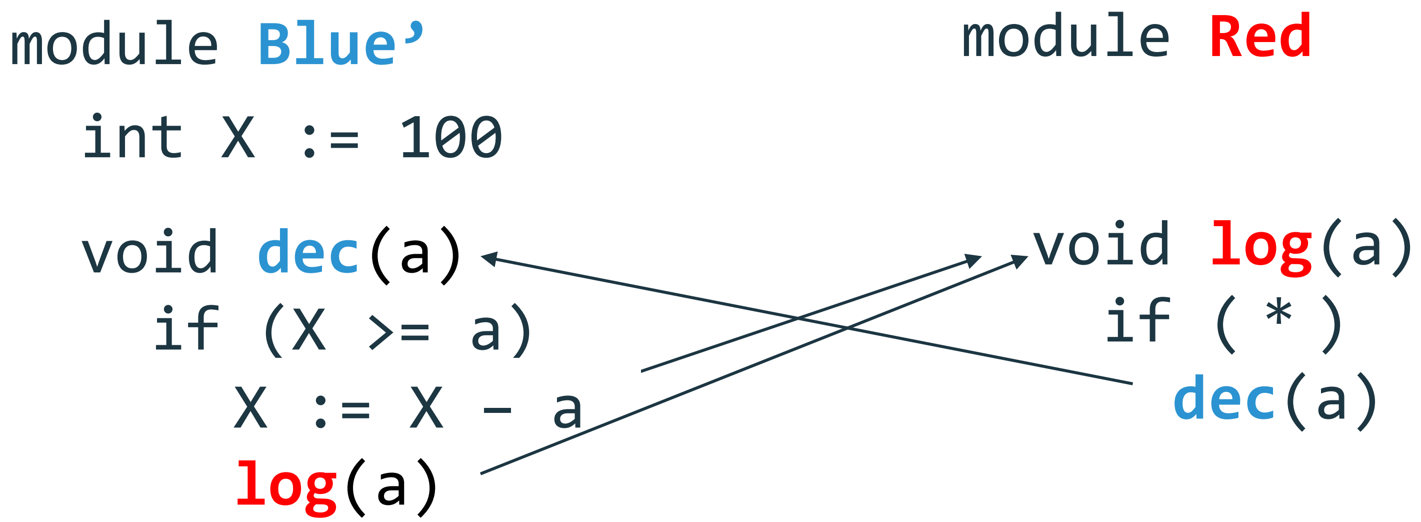
```
module Red

   void log(a)
     if ( * )
       dec(a)
```



Store: X=100 X=0 X=-100

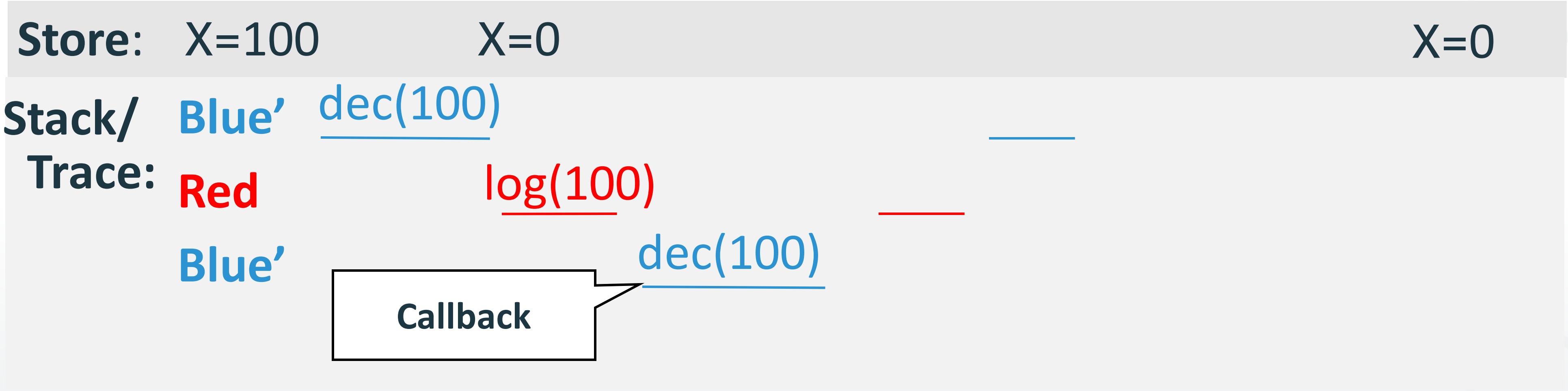
Stack/Trace:

Any Execution for Blue’



```
module Blue’
  int X := 100

  void dec(a)
    if (X >= a)
      X := X - a
      log(a)
```

```
module Red

  void log(a)
    if ( * )
      dec(a)
```



Store:	X=100	X=0				X=0
Stack/ Trace: Blue’	dec(100)				____	
Red		log(100)		____		
Blue’			dec(100)			

Callback

Analysis of Ethereum Transactions (7/2015 — 6/2017)

Contracts	Invocations	Callbacks	Non-ECF$_C$ callbacks
342,316	96,409,071	284,332	3,312

- Breakdown

  3298: DAO attack

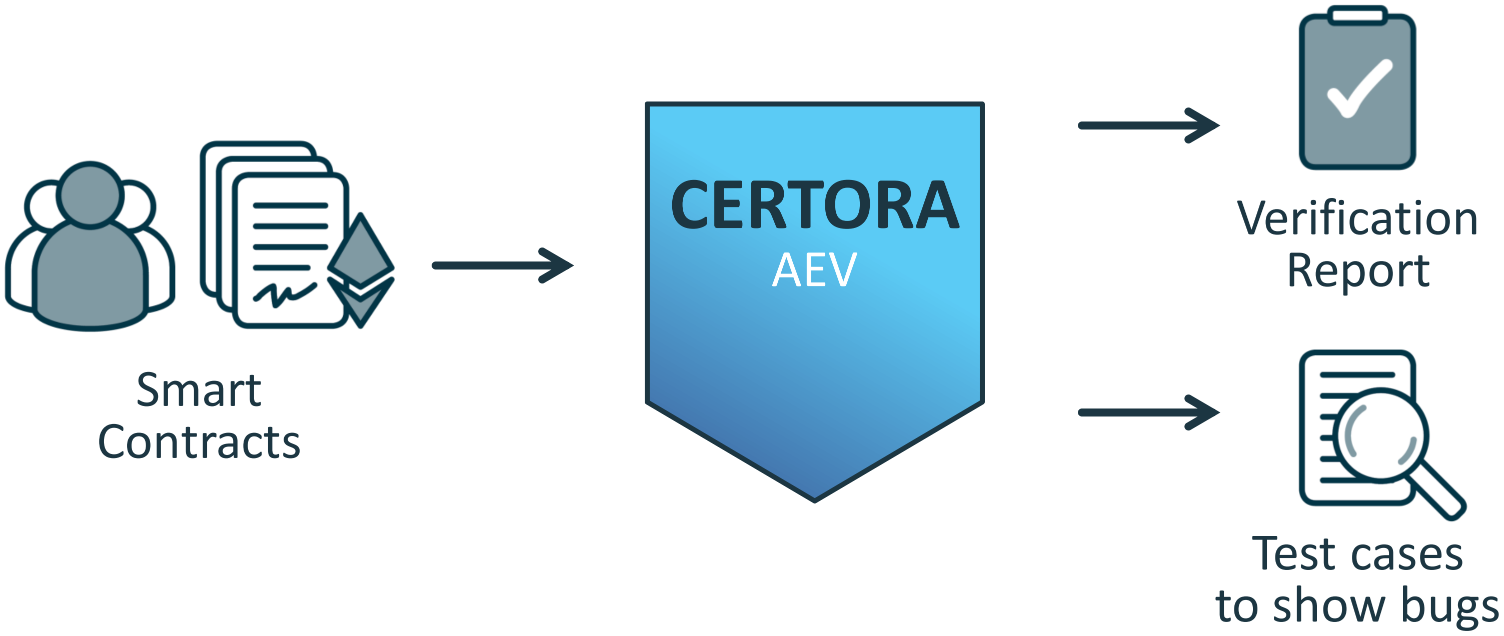
  10: DAO-like vulnerability

  3: Dummy DAO training exercise

  1: DAO demonstration

- Non-ECF callbacks = Attack
  - No “false positive”

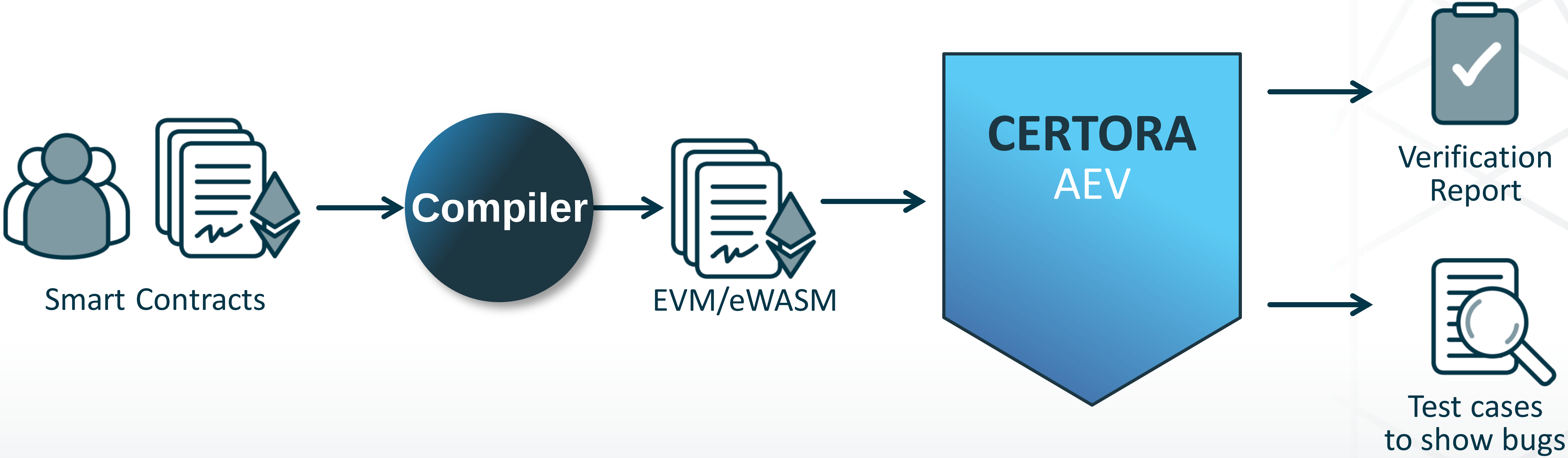
Certora - Automatic Exact Verification (AEV)



Benefits

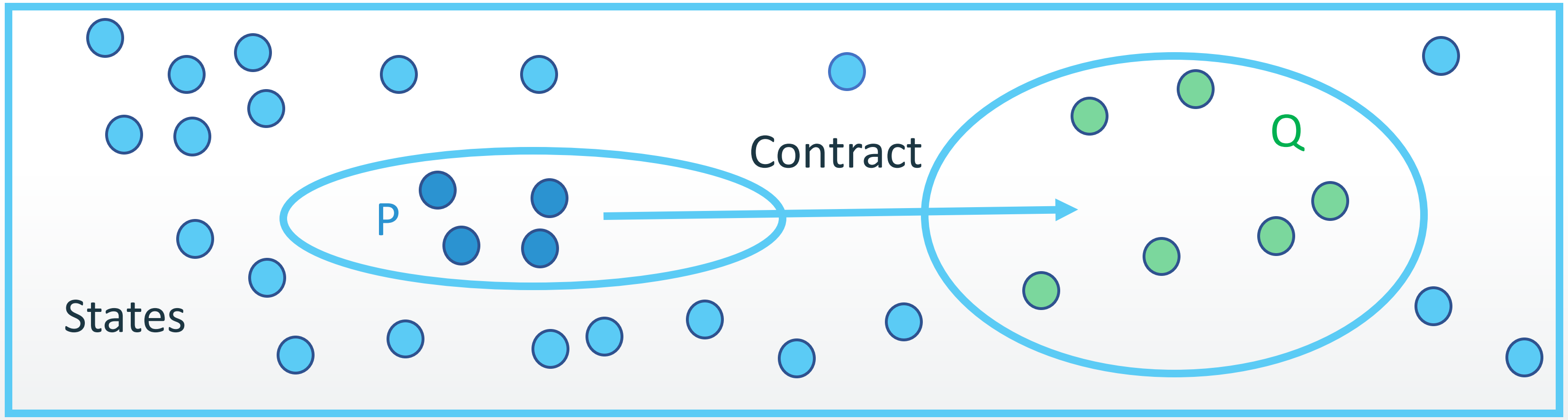
Superior Accuracy	Automatic	Zero False Alarms	Zero Missed Errors
Most accurate method to detect bugs	No customization per contract or services are required	All reported errors are real and come with risk explanation	All errors are eventually detected and come with formal checkable proofs

How does Certora-AEV work?



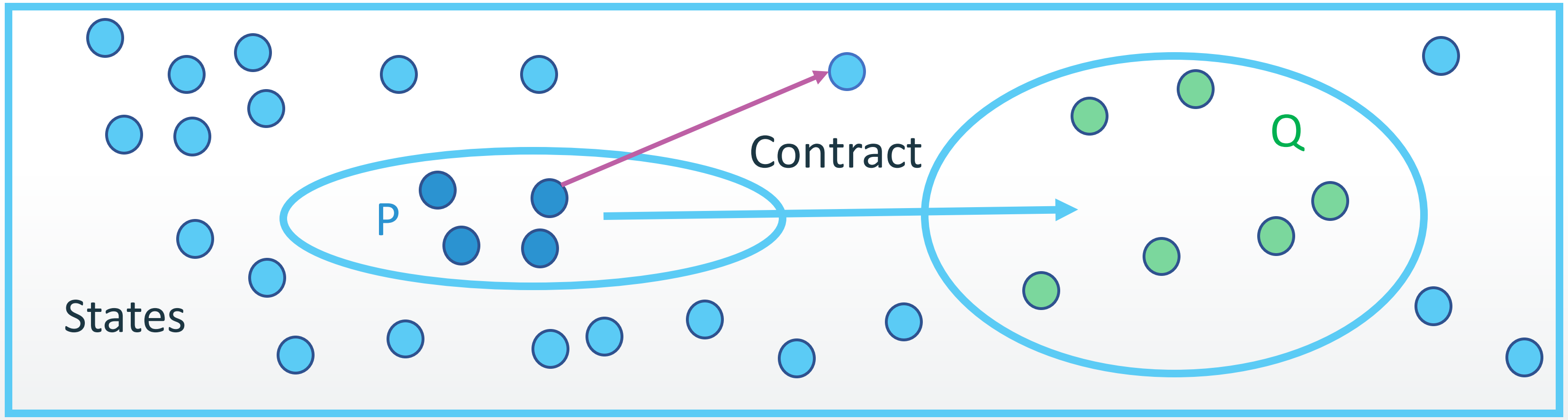
Hoare Triples

- Useful to explain verification
- Annotate the code with assertions
- **{P}** Contract **{Q}**
  - Every execution of the contract starting in a state in P results in a state in Q



Hoare Triples

- Useful to explain verification
- Annotate the code with assertions
- {P} Contract {Q}
  - Every execution of the contract starting in a state in P results in a state in Q



```
if P then {
        Contract;
        assert Q;
}
```

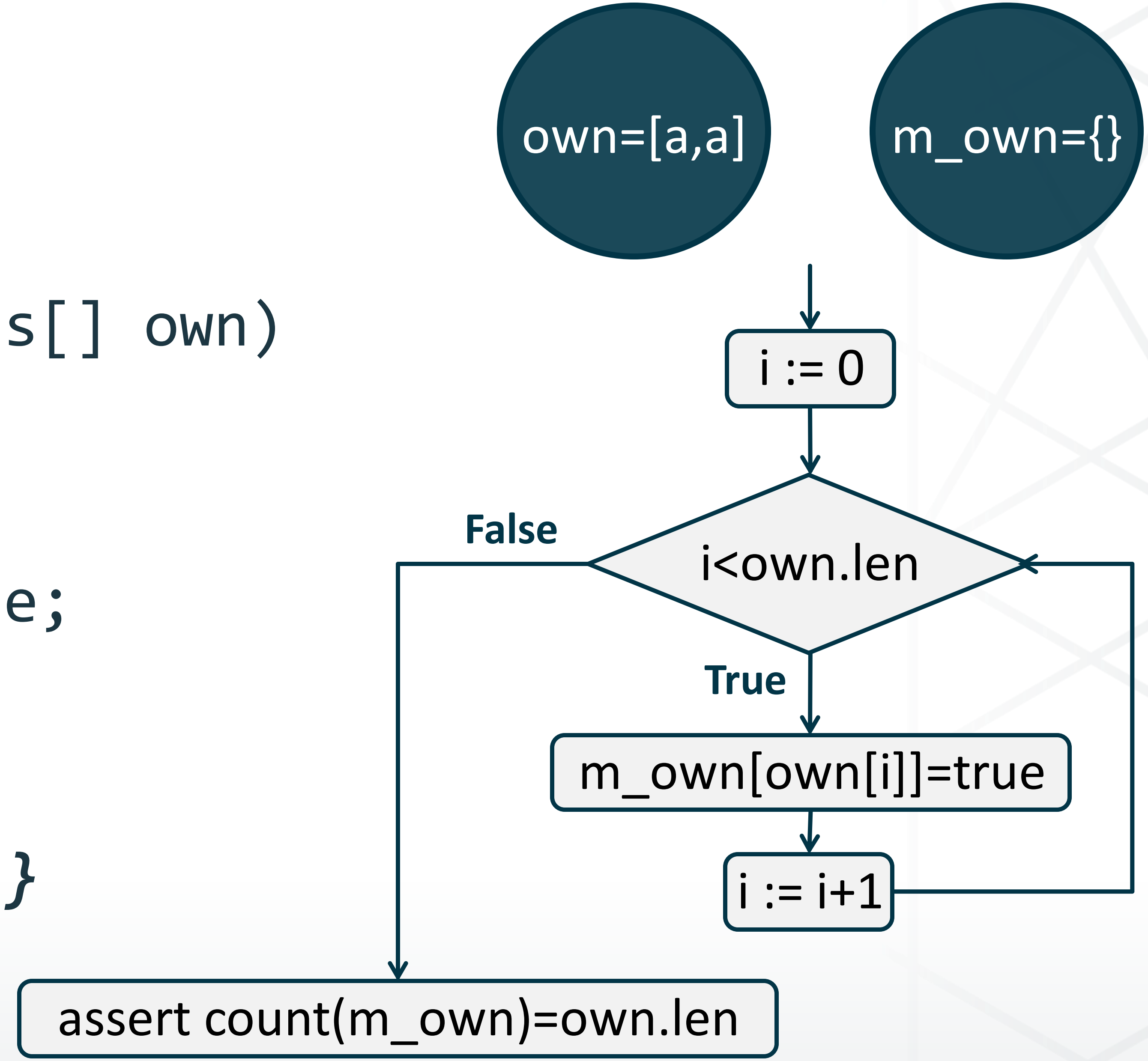
Example Hoare Triples

- **{ $x=0$ }** $x := x+2$ **{ $x=2$ }** ✓*valid*
- **{ $x=0$ }** $x := x+2$ **{ $x>0$ }** ✓*valid*
- **{ $x=1$ }** $x := x+y$ **{ $x>0$ }** ✗*invalid* *Test $y = -3$*
- **{ $x=1 \wedge y\geq 0$ }** $x := x+y$ **{ $x>0$ }** ✓*valid*
- **{ true }** if $x<0$ then $y:=-x$ else $y:=x$ **{ $y\geq 0$ }** ✓*valid*
- **{ $y\geq 0$ }** $t:=y$; $z:=1$; while $t>0$ do $z:=z*x$; $t:=t-1$; done **{ $z=x^y$ }** ✓*valid*

Example Wallet

```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```

```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



own=[a,a]

m_own={}

i := 0

i<own.len

False

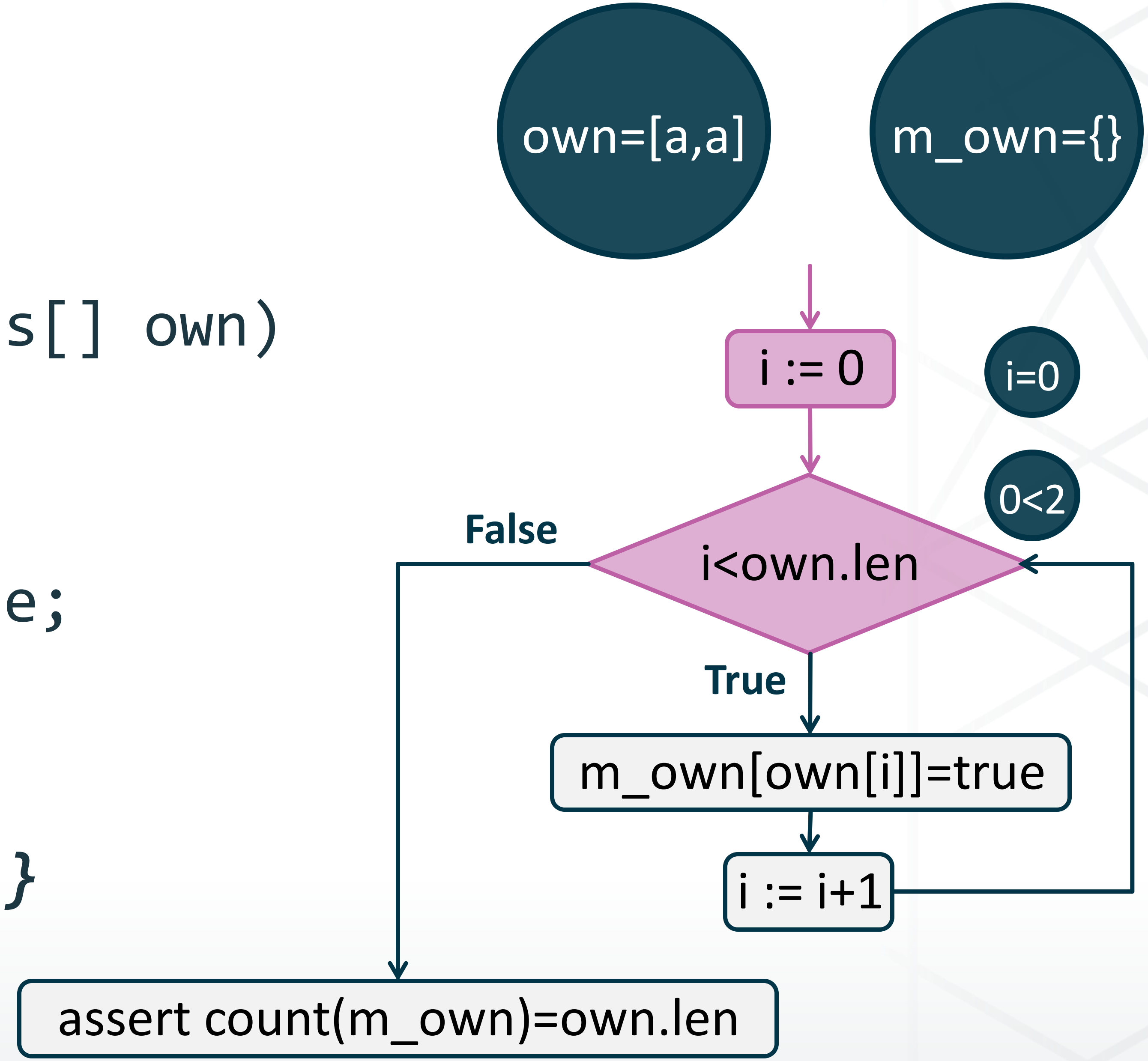
True

m_own[own[i]]=true

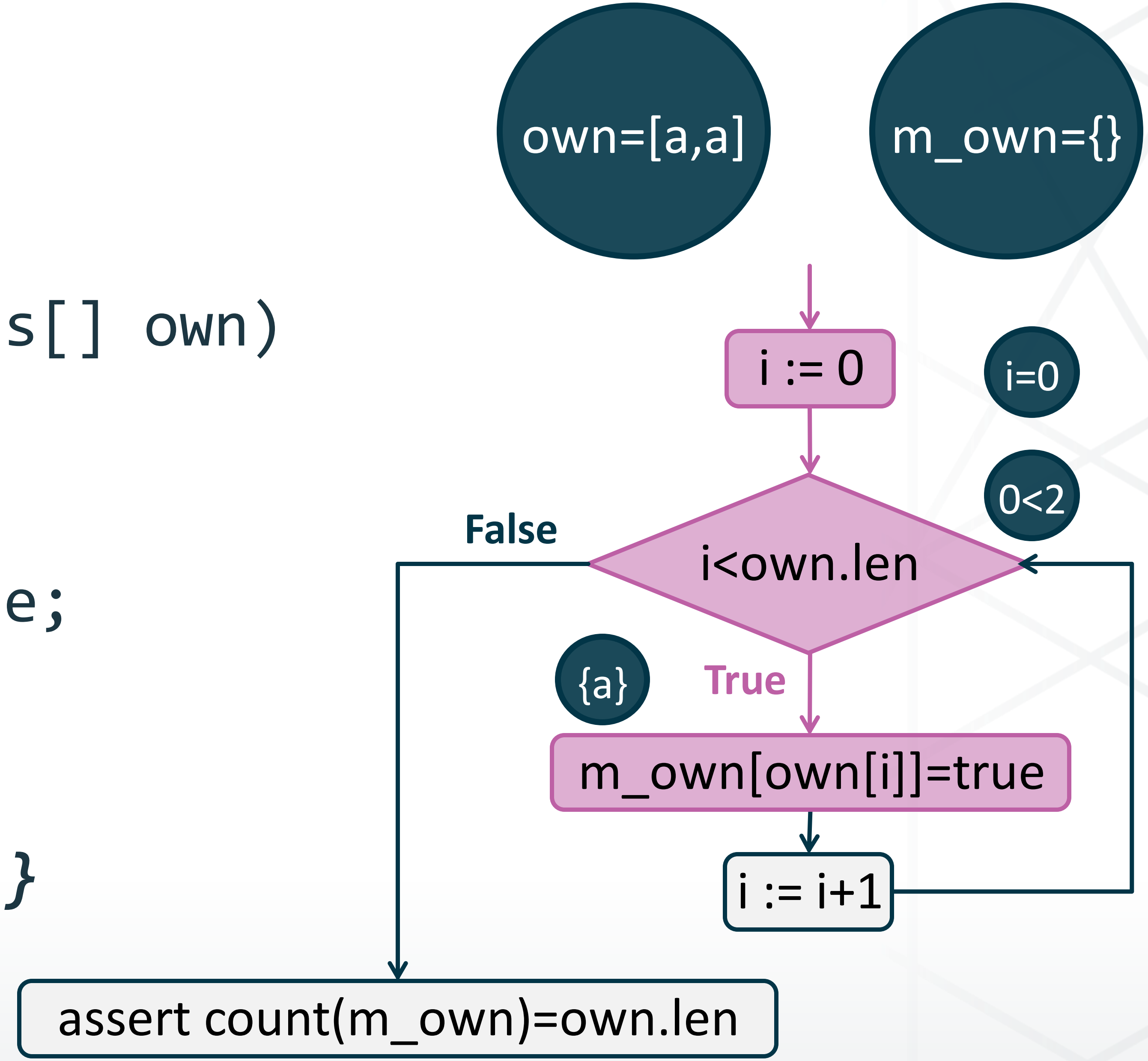
i := i+1

assert count(m_own)=own.len

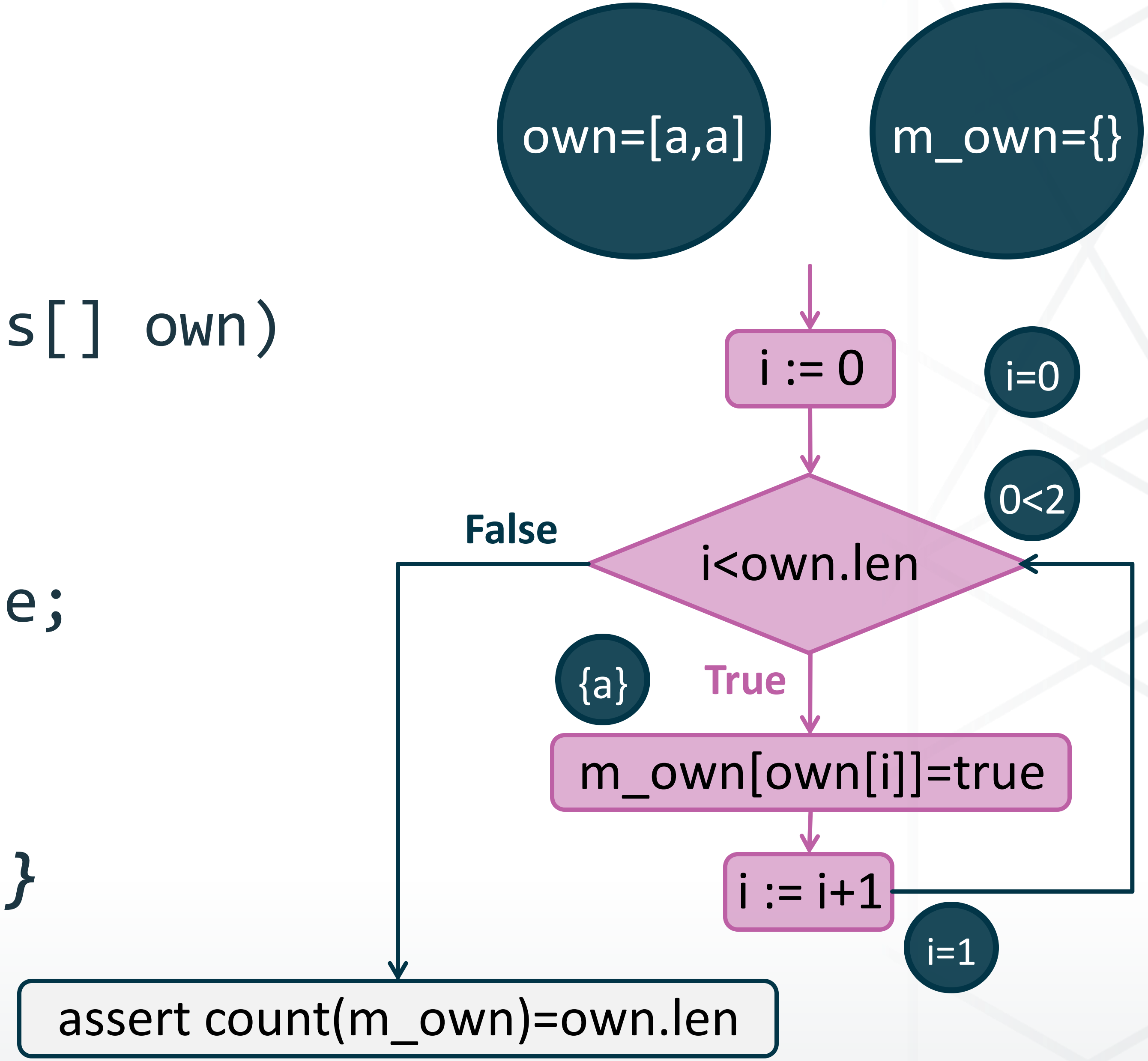
```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
```



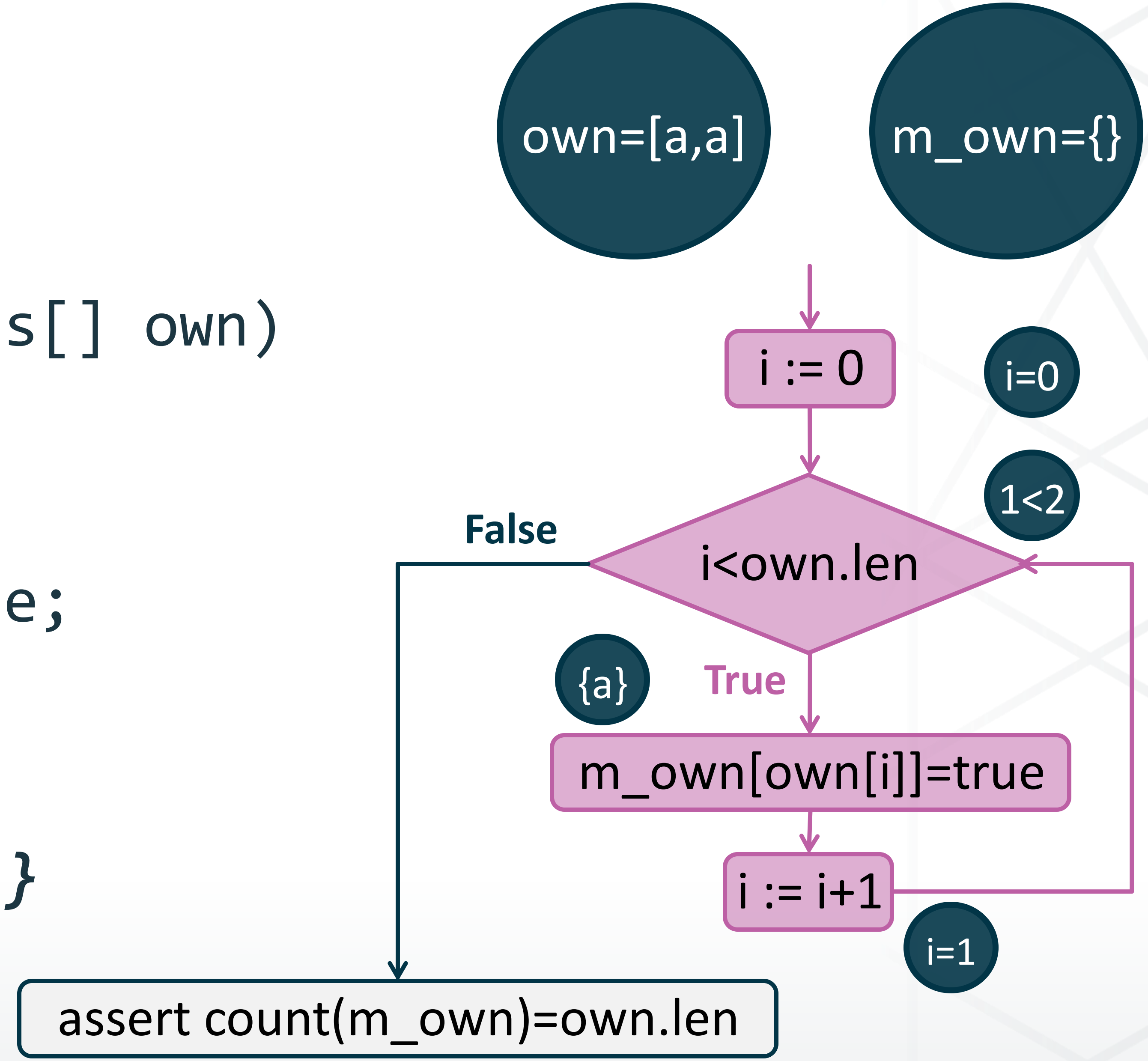
```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



own=[a,a]

m_own={}

i := 0

i=0

1<2

False

i<own.len

{a}

True

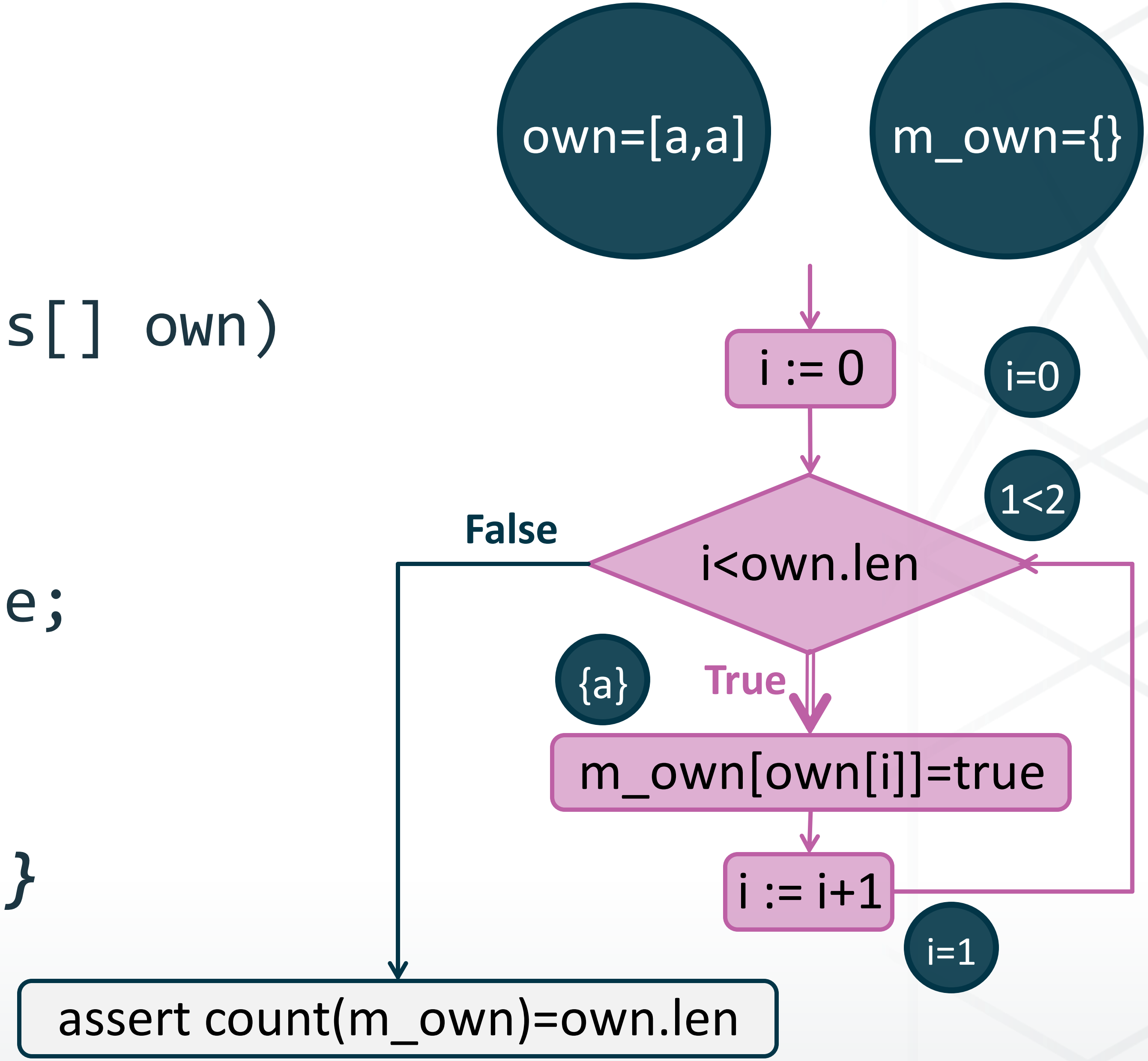
m_own[own[i]]=true

i := i+1

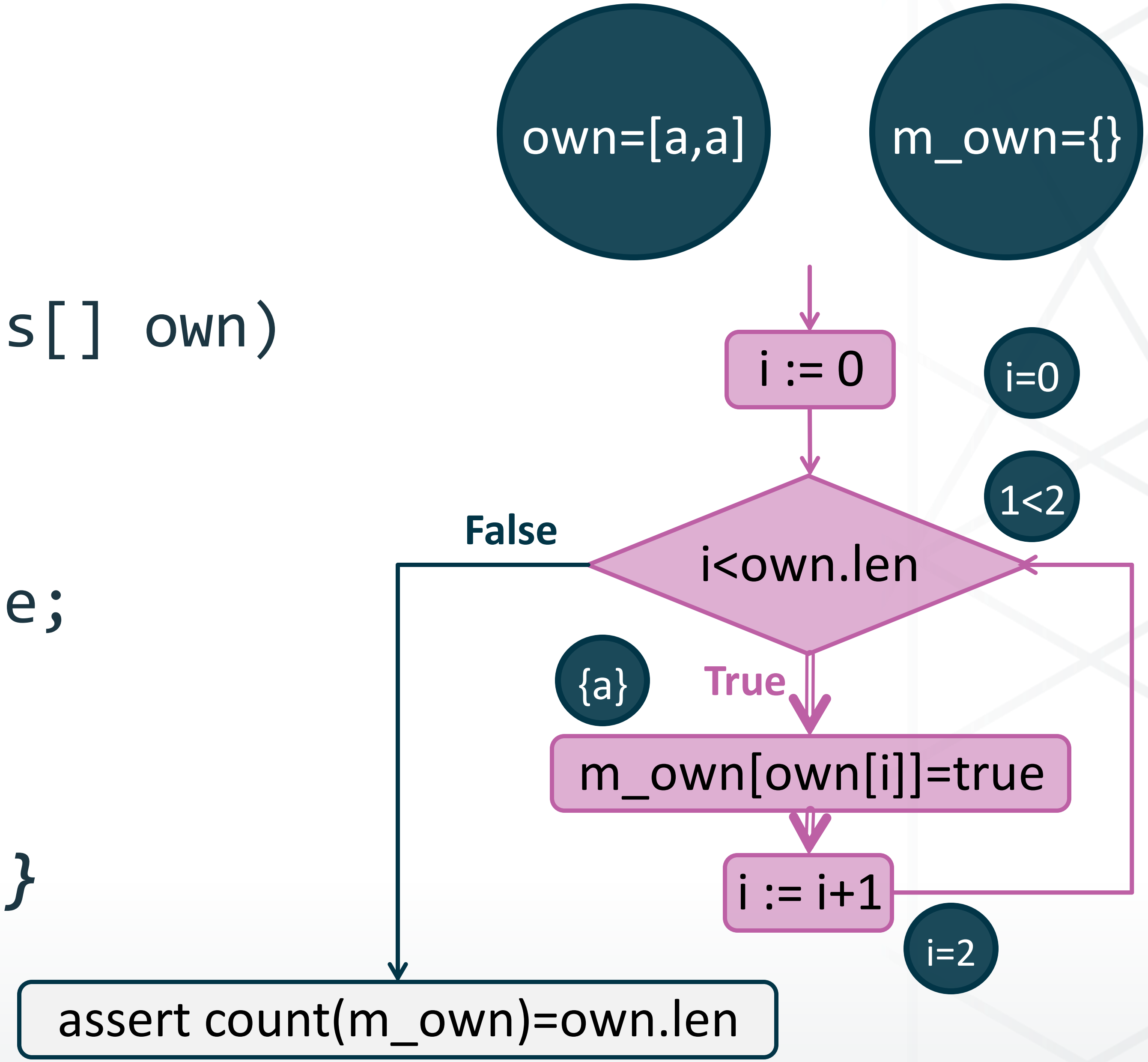
i=1

assert count(m_own)=own.len

```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



own=[a,a]

m_own={}

i := 0

i=0

1<2

False

i<own.len

{a}

True

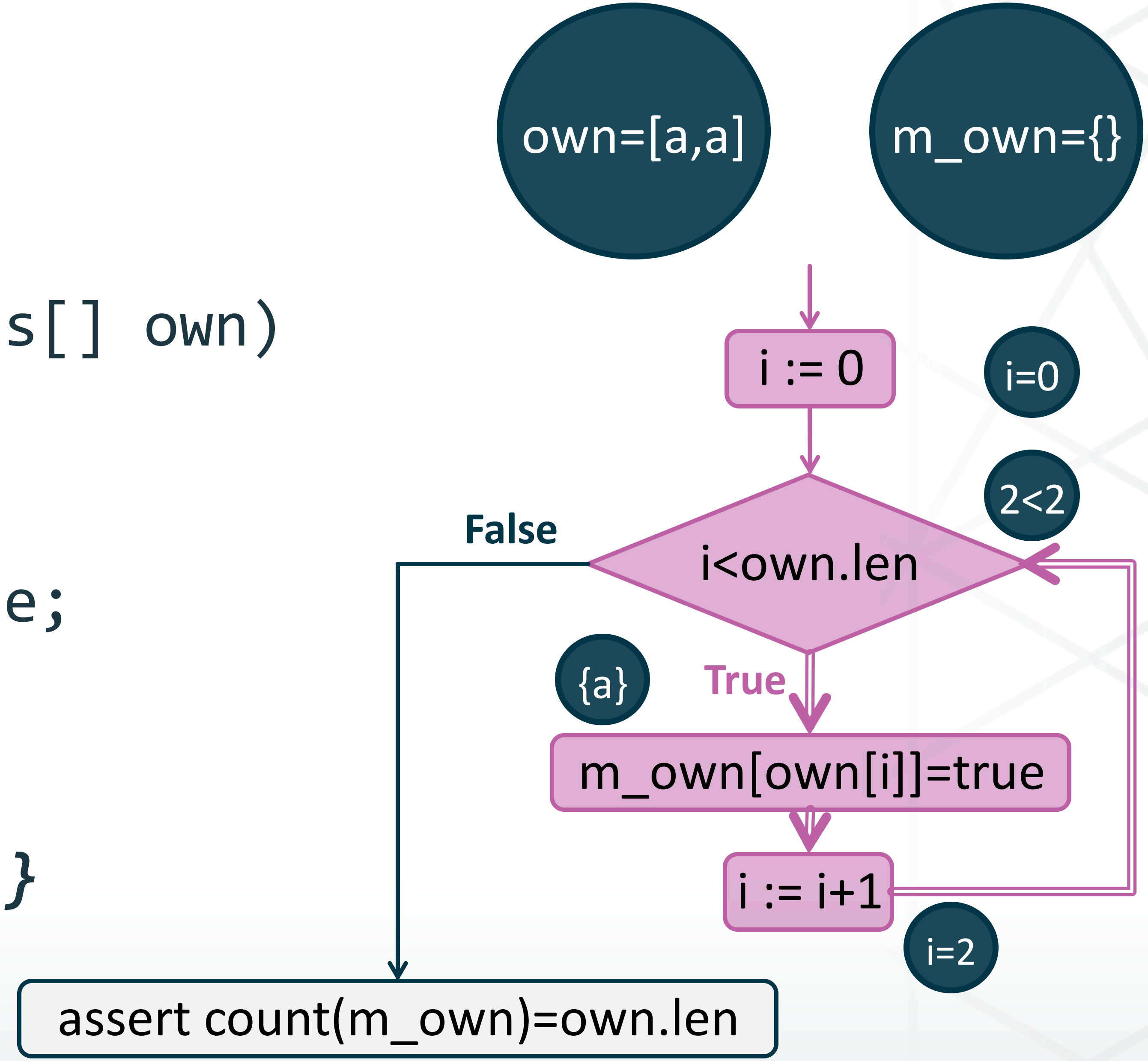
m_own[own[i]]=true

i := i+1

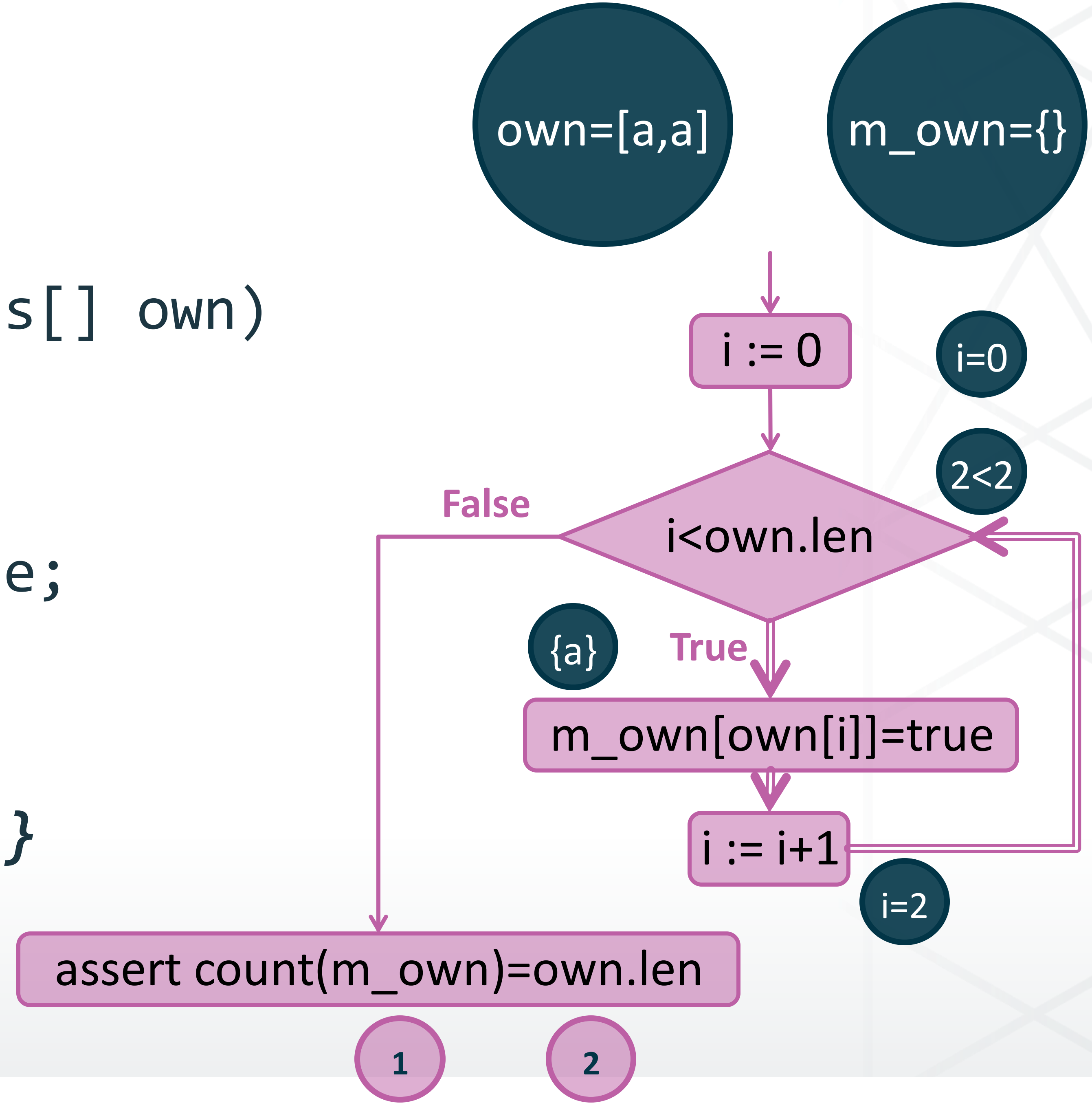
i=2

assert count(m_own)=own.len

```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
   int i = 0;
   while (i < own.len)
      m_own[own[i]] = true;
      ++i;

{ count(m_own) = own.len }
```



own=[a,a]

m_own={}

i := 0

i=0

i<own.len

2<2

False

True

{a}

m_own[own[i]]=true

i := i+1

i=2

assert count(m_own)=own.len

1

2

Fixed Wallet

```
{ count(m_own) = 0 }

wallet_constructor_fixed(address[] own)
    int i = 0;
    while (i < own.len)
        if (m_own[own[i]])
            abort;
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
```

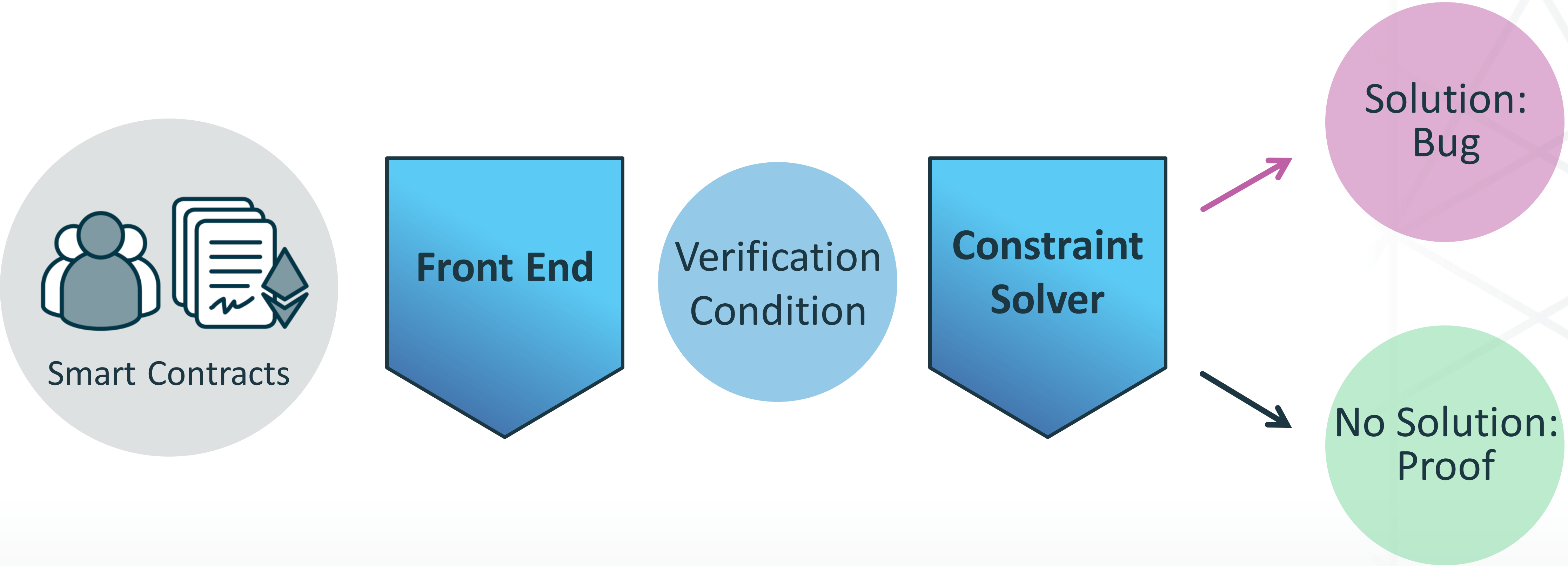
Fixed Wallet

```
{ count(m_own) = 0 }

wallet_constructor_fixed(address[] own)
    int i = 0;
    while (i < own.len)
        if (m_own[own[i]])
            abort;
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
```

```
{ count(m_own) = 0 }
   int i = 0;
   while (i < own.len) { own.len>i≥ 0 ∧ count(m_own) = i }
      if (m_own[own[i]])
         abort;
      { own.len>i ≥ 0 ∧ count(m_own) = i ∧ ¬m_own[own[i]] }
      m_own[own[i]] = true;
      { own.len>i ≥ 0 ∧ count(m_own) = i+1 }
      ++i;
      { own.len ≥ i ≥ 0 ∧ count(m_own) = i }
{ count(m_own) ≥ own.len }
```

How does Certora-AEV automatically check correctness?

Secret Sauce – Compilation and Constraint Solving


Smart Contracts

Front End

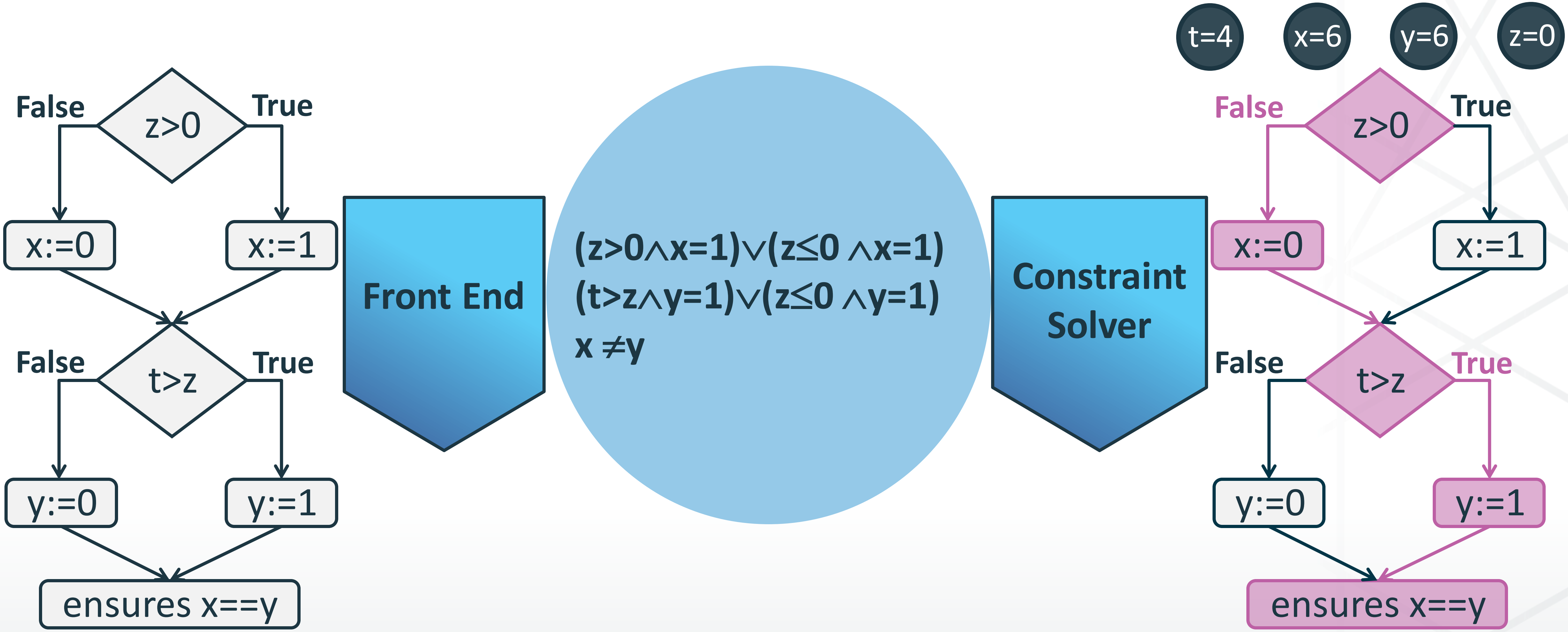
Verification Condition

Constraint Solver

Solution: Bug

No Solution: Proof

Secret Sauce – Compilation and Constraint Solving



z>0

False

True

x:=0

x:=1

t>z

False

True

y:=0

y:=1

ensures x==y

Front End

$(z>0 \wedge x=1) \vee (z \leq 0 \wedge x=1)$

$(t>z \wedge y=1) \vee (z \leq 0 \wedge y=1)$

$x \neq y$

Constraint Solver

t=4

x=6

y=6

z=0

z>0

False

True

x:=0

x:=1

t>z

False

True

y:=0

y:=1

ensures x==y

Summary

- Ensured correctness is critical for the adoption of Smart Contracts
- Formal verification is the tool we have
- Enabling technologies
  - Modularity
  - Mature tools