Peter Müller

Joint work with Marco Eilers and Thibault Dardinier

PROVING INFORMATION FLOW SECURITY FOR CONCURRENT PROGRAMS

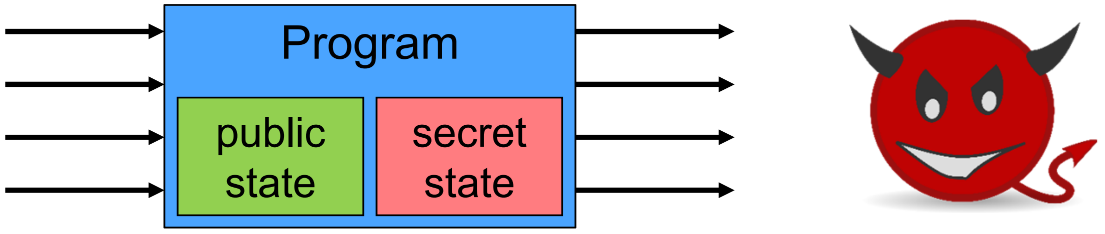
ETH zürich

Microsoft Azure Breach, April 2021

> *"Our investigation found that a consumer signing system crash in April of 2021 resulted in a snapshot of the crashed process ("crash dump"). The crash dumps, which redact sensitive information, should not include the signing key. In this case, a race condition allowed the key to be present in the crash dump (this issue has been corrected).“*
>
> Microsoft Security Response Center

Secure Information Flow

- Programs maintain secret state such as crypto keys



Program

public state

secret state

- High-level goal:
  Verify that attackers cannot learn secrets by interacting with the implementation

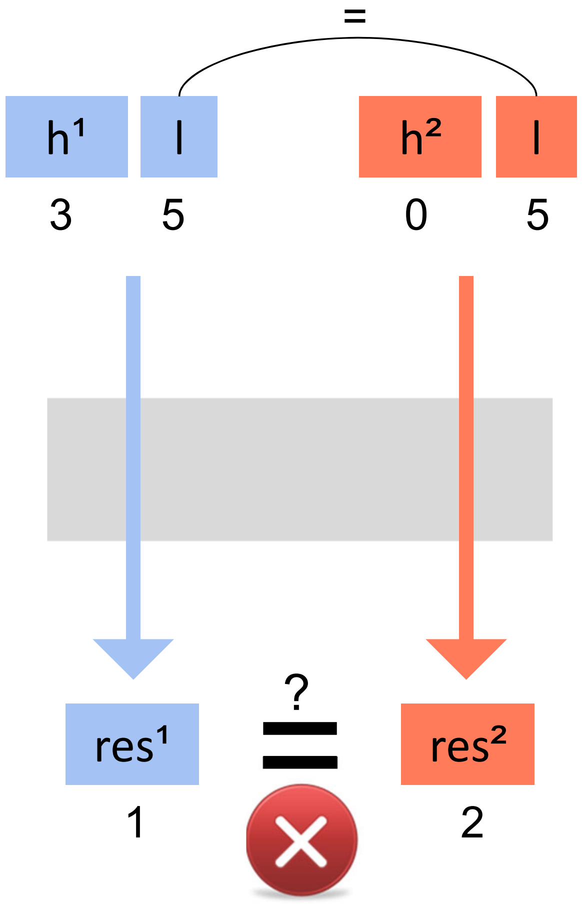
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

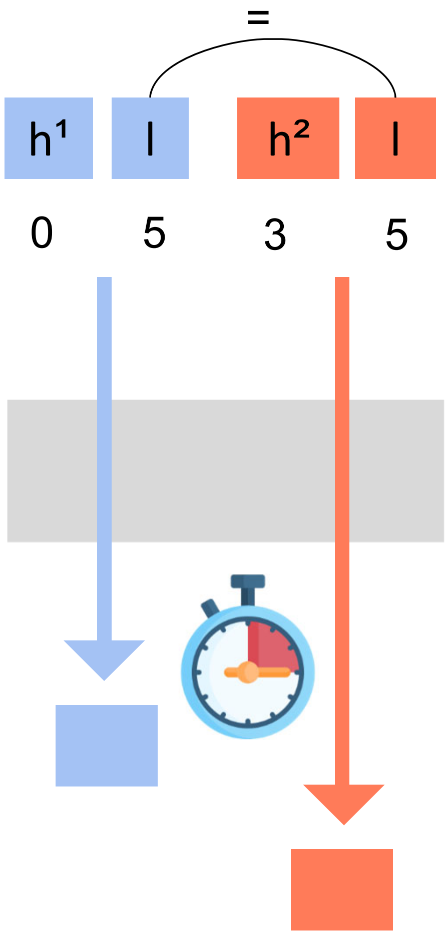
Does *res* leak information about *h*?



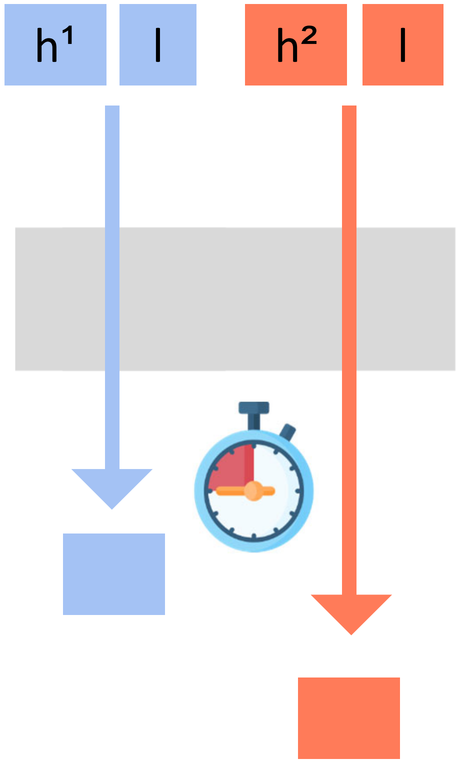
Secure Information Flow: Timing Channel

```
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

Does the execution time leak information about *h*?



Reasoning About Timing Channels



- Proving the absence of timing channels is extremely difficult
  - Compiler optimizations
  - Value-dependent duration of CPU instructions
  - Complex hardware: pipelining, caching, etc.
- In many scenarios, attackers cannot observe execution time
  - Data is published only after computation
  - Time measurement is too imprecise
    (e.g., due to a laggy network)

Our attacker model:

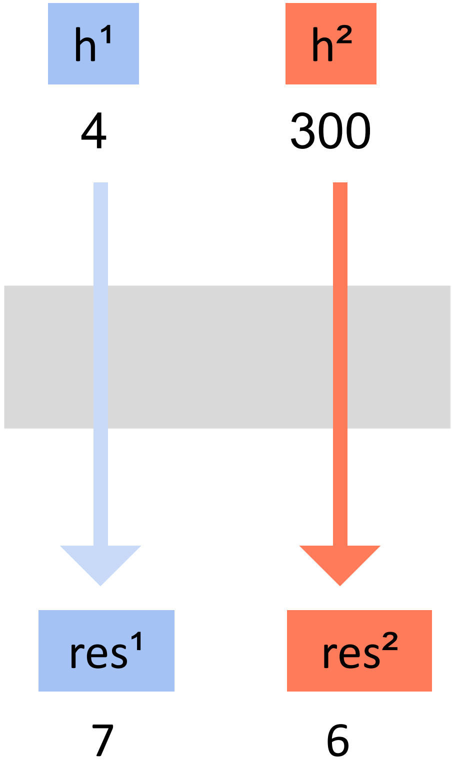
Attacker may observe final results,
but not intermediate states or timing

Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

```
i = 0          ||  j = 0
while i < h:   ||  while j < 100:
  i += 1       ||    j += 1
shared = 6     ||  shared = 7
          return shared
```



h^1 4

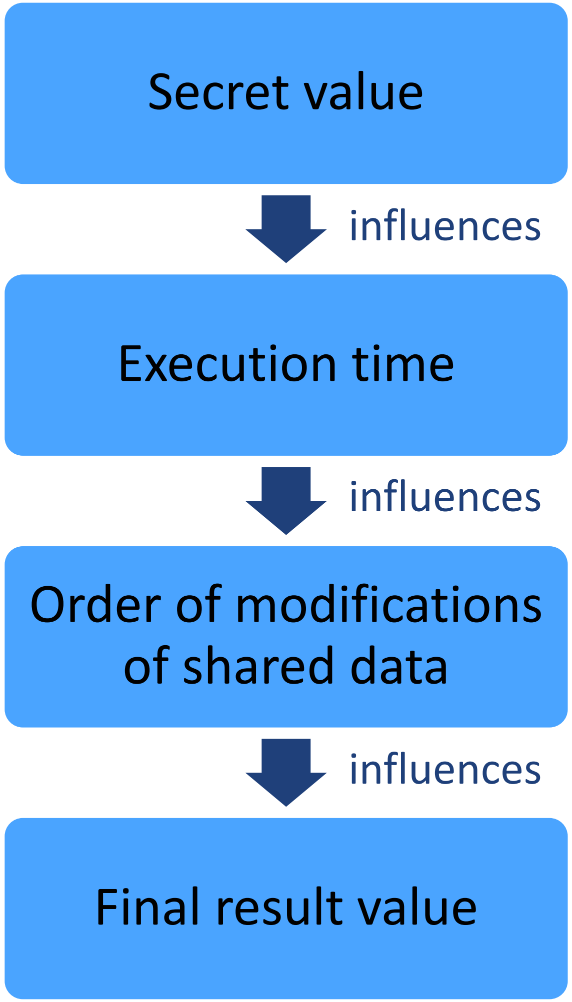
h^2 300

res^1 7

res^2 6

Shared-Memory Concurrency Ruins Everything

```
i = 0             || j = 0
while i < h:      || while j < 100:
  i += 1          ||   j += 1
shared = 6        || shared = 7
          return shared
```


Secret value

influences

Execution time

influences

Order of modifications of shared data

influences

Final result value

Our goal:
Verify the absence of value channels
without reasoning about timing

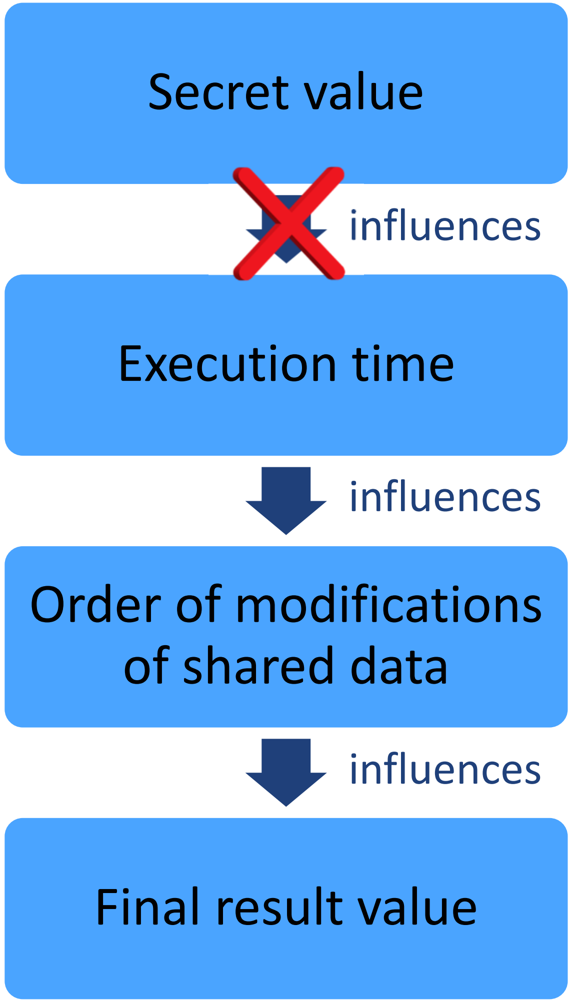
Existing (Modular) Solutions

Insecure

```
i = 0               ||  j = 0
while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
shared = 6          ||  shared = 7
          return shared
```

Secure

```
            shared = 1

i = 0               ||  j = 0
while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
shared += 6         ||  shared += 7
          return shared
```



Secret value

influences

Execution time

influences

Order of modifications of shared data

influences

Final result value

Key idea:
The thread schedule does not influence the final result if modifications commute

Our Solution: Commutativity

Insecure

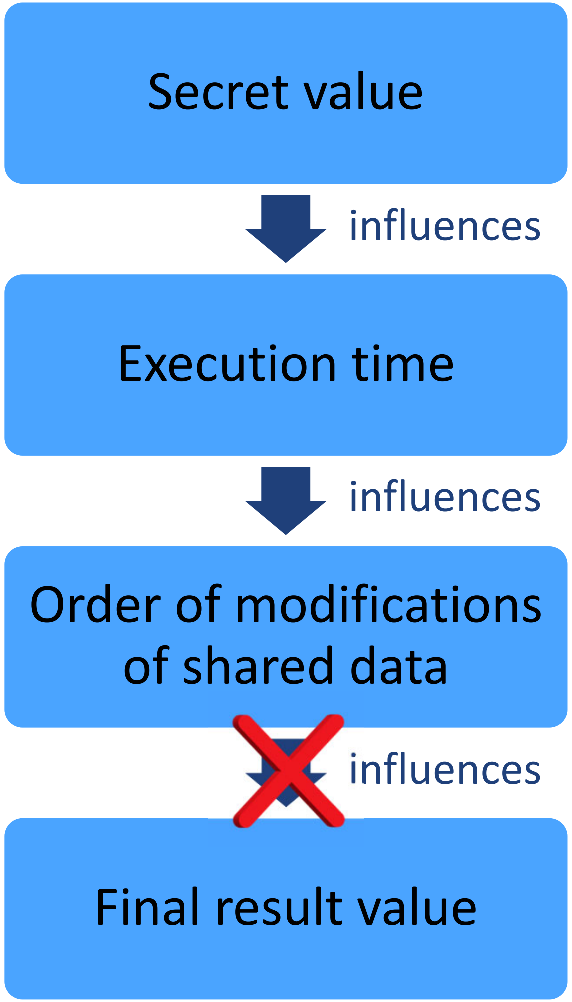
```
i = 0              ||  j = 0
while i < h:       ||  while j < 100:
  i += 1           ||    j += 1
shared = 6         ||  shared = 7
              return shared
```

✗

Secure

```
              shared = 1

i = 0              ||  j = 0
while i < h:       ||  while j < 100:
  i += 1           ||    j += 1
shared += 6        ||  shared += 7
              return shared
```

✓


Secret value

influences

Execution time

influences

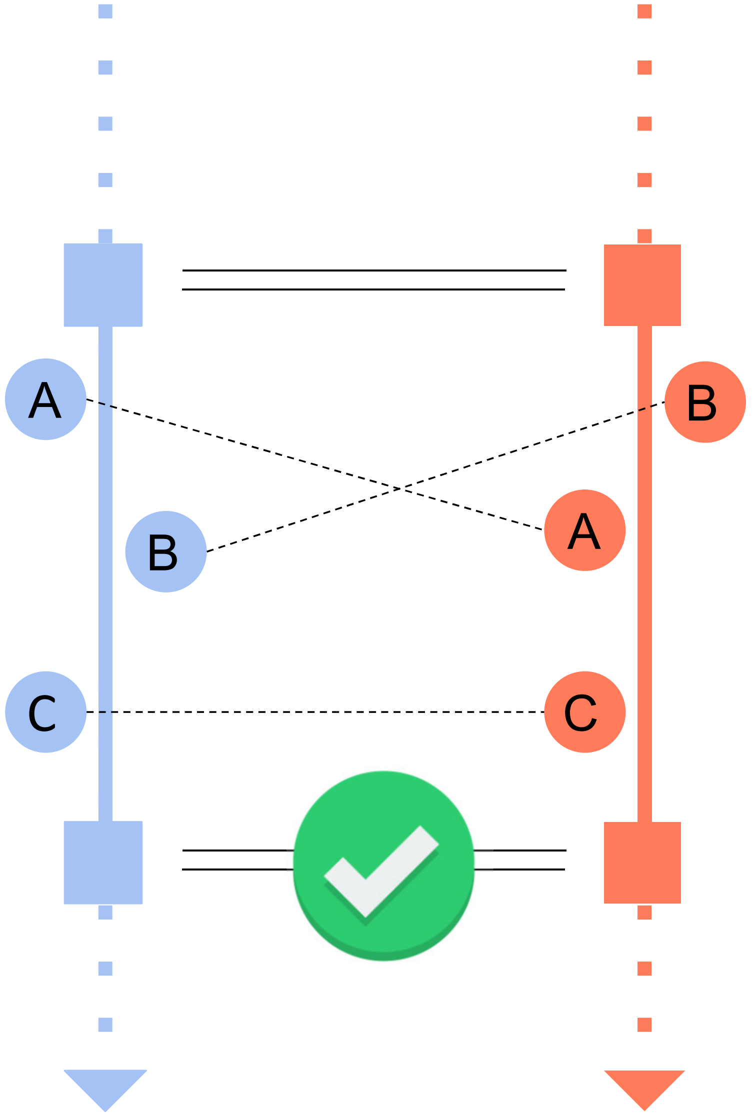
Order of modifications of shared data

influences

Final result value

Basic Solution

```
            shared = …
atomic:            ||  atomic:
  shared.A()       ||    shared.B()
atomic:            ||
  shared.C()       ||
                …
```



(1) **Prove**: *shared* has the same initial value in both executions

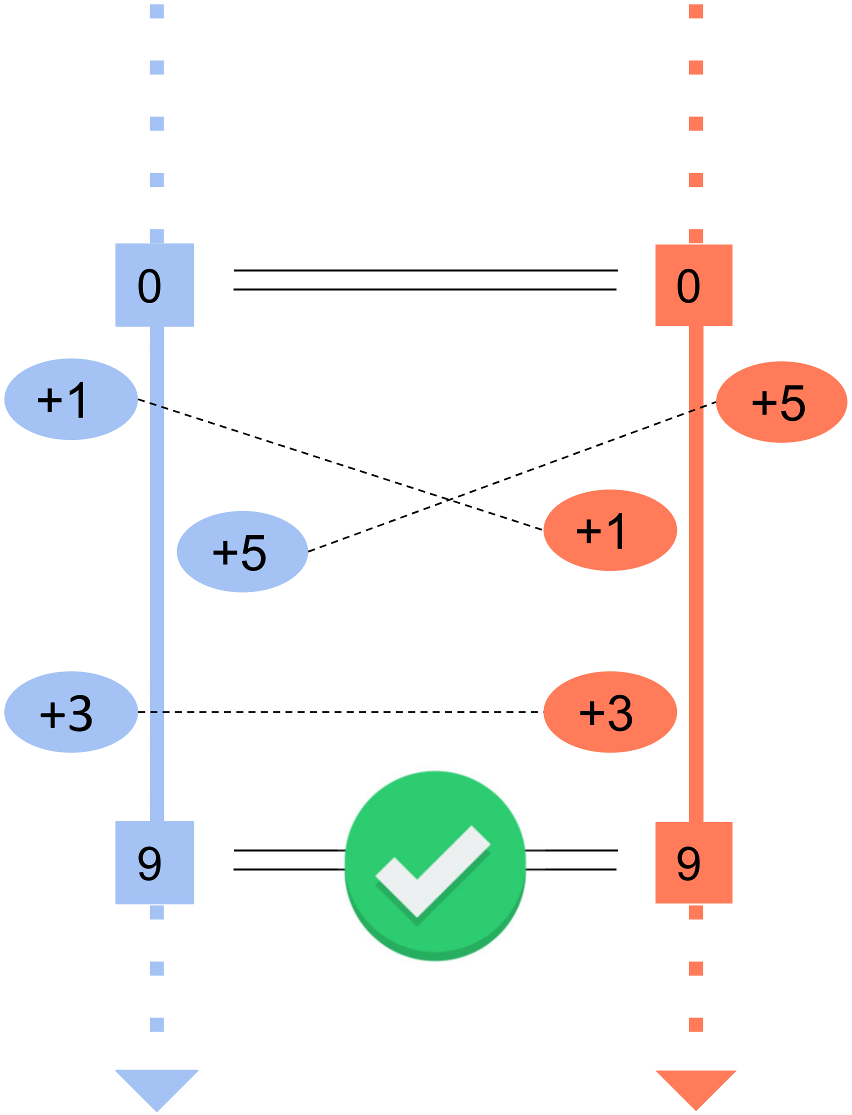
(2) **Prove**: the two executions perform the “same” updates

(3) **Prove**: the updates commute

Assume: *shared* has the same final value in both executions

Basic Solution

```
          shared = 0
atomic:            || atomic:
  shared += 1      ||   shared += 5
atomic:            ||
  shared += 3      ||
                 …
```



- ✓ **Prove**: *shared* has the same initial value in both executions
- ✓ **Prove**: the two executions perform the “same” updates
- ✓ **Prove**: the updates commute

Assume: *shared* has the same final value in both executions

Basic Solution

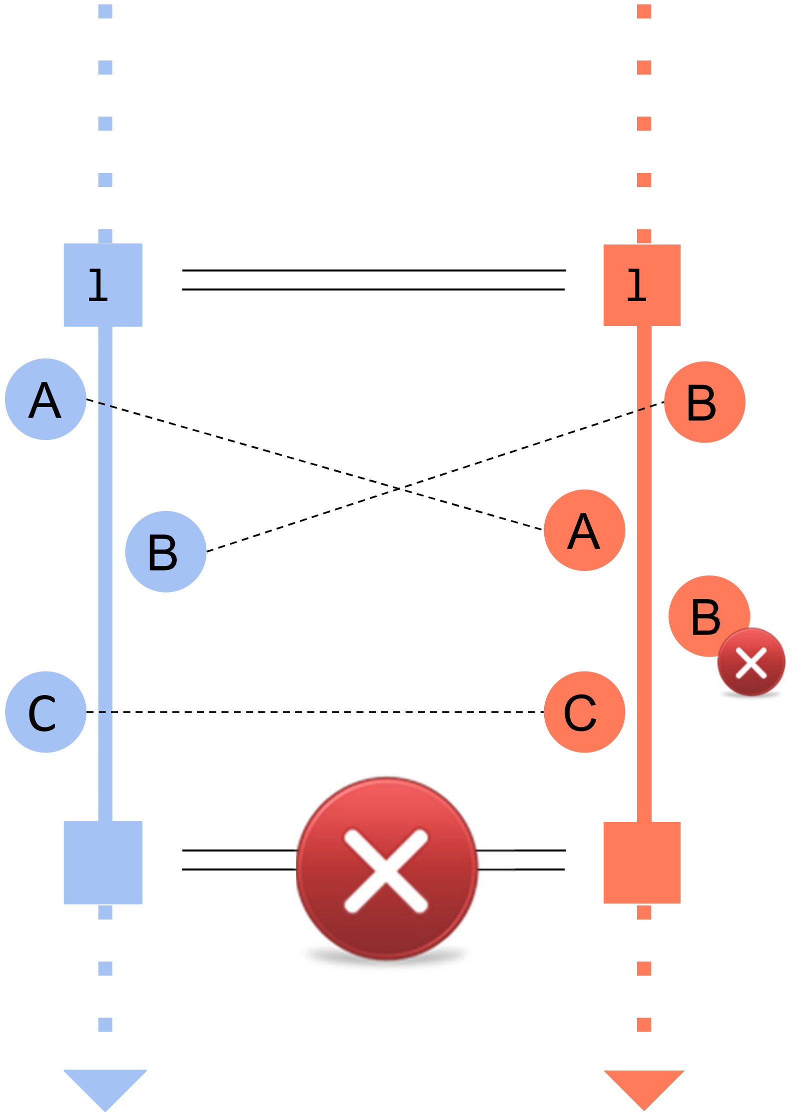
```
          shared = 1
atomic:          || atomic:
  shared.A()     ||   shared.B()
atomic:          || if h > 0:
  shared.C()     ||   atomic:
                 ||     shared.B()
              …
```

✓ **Prove**: *shared* has the same initial value in both executions

✗ **Prove**: the two executions perform the “same” updates

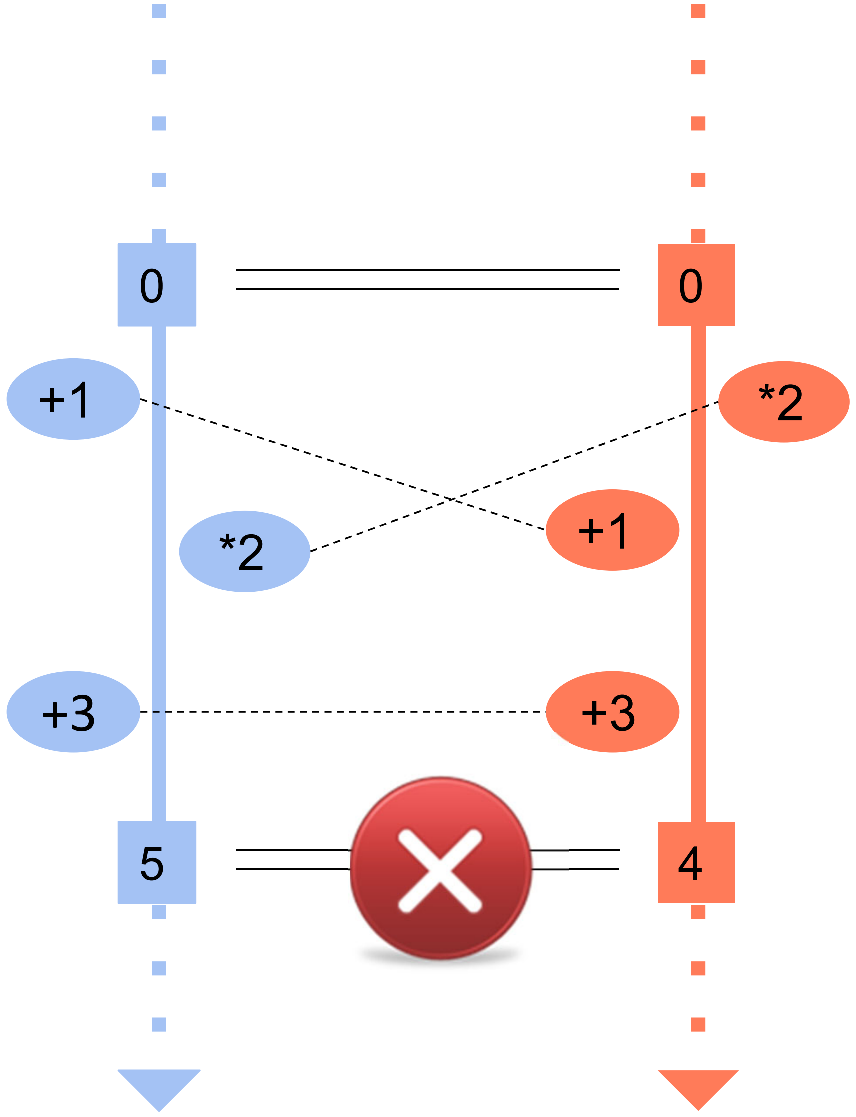
(3) **Prove**: the updates commute

Assume: *shared* has the same final value in both executions



Basic Solution

```
        shared = 0
atomic:               ||atomic:
  shared += 1         ||  shared *= 2
atomic:               ||
  shared += 3         ||
             …
```



✓ **Prove**: *shared* has the same initial value in both executions

✓ **Prove**: the two executions perform the “same” updates

✗ **Prove**: the updates commute

Assume: *shared* has the same final value in both executions

CommCSL:
A concurrent separation logic with commutativity reasoning

Relational Reasoning

- Assertions relate two states

P,Q ::= b | emp | e ↦ e | P * Q | … | Low(e)

Unary assertions need to hold individually in both states

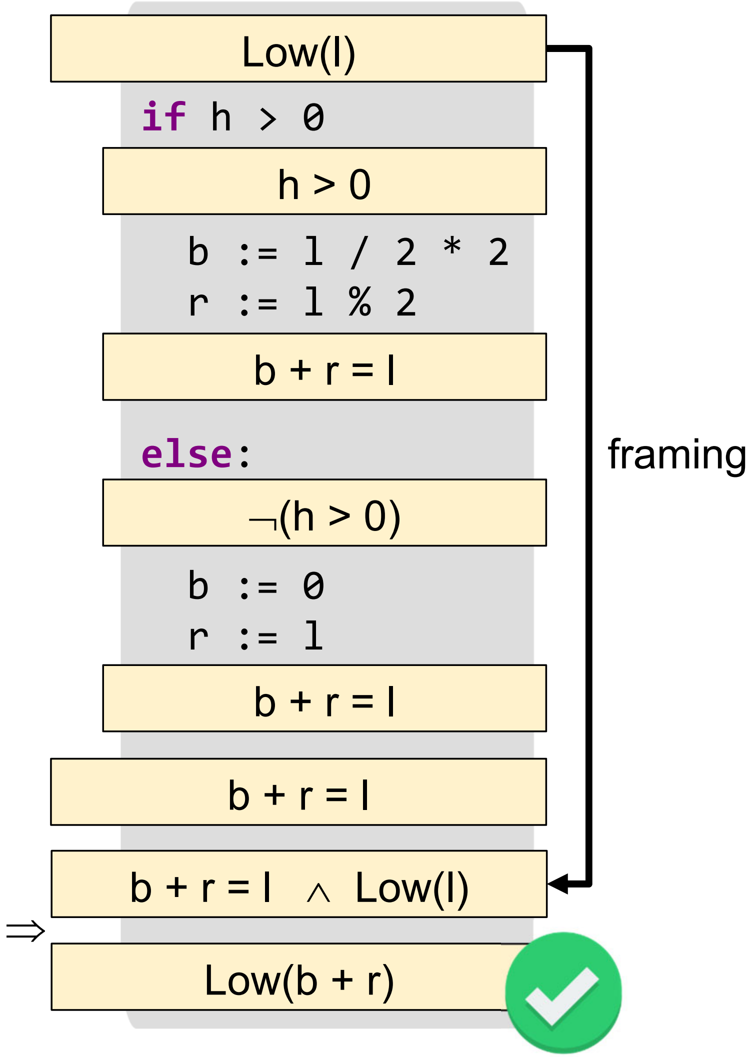
e evaluates to the same value in both states

- Judgment of the logic relates two program executions

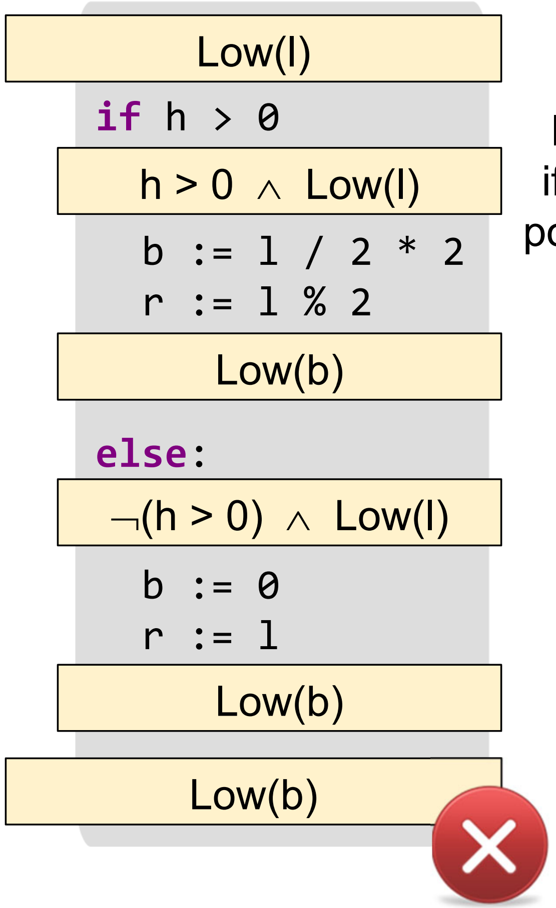
⊢ { P } c { Q }

Two executions of c from a pair of initial states that together satisfy P will not abort and, if both terminate, the resulting states will satisfy Q

Relational Reasoning: Example



```
Low(l)
if h > 0
    h > 0
    b := l / 2 * 2
    r := l % 2
    b + r = l
else:
    ¬(h > 0)
    b := 0
    r := 1
    b + r = l
b + r = l
b + r = l ∧ Low(l)
⇒
Low(b + r)
```



```
Low(l)
if h > 0
    h > 0 ∧ Low(l)
    b := l / 2 * 2
    r := l % 2
    Low(b)
else:
    ¬(h > 0) ∧ Low(l)
    b := 0
    r := 1
    Low(b)
Low(b)
```

If-rule requires that if-condition is low or postcondition is unary

Data Abstraction in Separation Logic

```
class List {
  elem: Int
  next: List

  void appendBack(e: Int)
    requires list(this, s)
    ensures  list(this, s∘[e])
  { … }
}
```

Separation logic specifies functional behavior in terms of an abstraction of the concrete data structure

```
list(ptr: List, s: Seq) ≡
  ptr.elem ↦ e * ptr.next ↦ n *
  (n = null ⇒ s = [ ]) *
  (n ≠ null ⇒ s[0] = e * list(n, s[1..]))
```

- We reason about commutative actions on the level of these abstractions
- A resource is the abstraction of a shared data structure

```
resource Sequence:
  type          Seq
  invariant     list(x, v)
  actions:
    append(v, e) ≡ v ∘ [e]
```

Proof Obligation 1: Same Initial Value in Both Executions

- Our verification technique
  - Checks that shared data is low before concurrent accesses
  - Guarantees that shared data is low after concurrent accesses

- These points in the execution are indicated by a share block-statement

$$\frac{\vdash \{\ P\ \}\ \ c\ \ \{\ Q\ \}}{\vdash \{\ I(x,v)\ *\ Low(v)\ *\ P\ \}\ \ \mathbf{share}\ x\ \mathbf{in}\ c\ \ \{\ \exists v' \bullet I(x,v')\ *\ Low(v')\ *\ Q\ \}}$$

Prove shared data structure has same initial value

Assume shared data structure has same final value

For simplicity, we assume that
there is only one resource,
which is implicit in the rule

Proof Obligation 2: Same Updates in Both Executions

- The shared data structure may be updated only through atomic statements

Every atomic statement performs one action

$$\frac{\vdash \{\ P * I(x,v)\ \}\ \ c\ \ \{\ Q * I(x,f(v,e))\ \}}{\vdash \{\ P * acs^{r}(args)\ \}\ \ \textbf{atomic}\ c\ \ \{\ Q * acs^{r}(args \cup^{\#} \{e\})\ \}}$$

- Without loss of generality, we assume that our resource has exactly one action f (multiple actions can be simulated via an additional parameter)

- We collect for every execution the argument tuples of the actions it performs
  - As a multiset of argument tuples
  - This multiset is stored in a separation logic resource acs (with fraction r)

Proof Obligation 2: Same Updates in Both Executions

- Actual check is performed when the resource is un-shared

Initially no actions were performed

Multiset of performed actions is the same in both executions

$$\frac{\vdash \{\ P * \mathtt{acs}^1(\varnothing^{\#})\ \}\quad c\quad \{\ Q * \mathtt{acs}^1(\mathtt{args}) * \mathtt{Low}(\mathtt{args})\ \}}{\vdash \{\ \mathtt{I}(x,v) * \mathtt{Low}(v) * P\ \}\quad \mathbf{share}\ x\ \mathbf{in}\ c\quad \{\ \exists v' \bullet \mathtt{I}(x,v') * \mathtt{Low}(v') * Q\ \}}$$

- This ”delayed” check avoids the need to closely align the two program executions

Proof Obligation 3: The Updates Commute

- Commutativity is checked for each resource declaration

```
resource R:
  type         T
  invariant    I(p, v)
  actions:
    f(v, e) ≡ …
```

```
∀e, e’ • f(f(v, e), e’) = f(f(v, e’), e)
```

Recall that we consider
only a single action

- Checking commutativity of the (abstract) action is much simpler than of concrete implementations

```
resource Sequence:
  type         Seq
  invariant    list(x, v)
  actions:
    append(v, e) ≡ v ∘ [e]
```

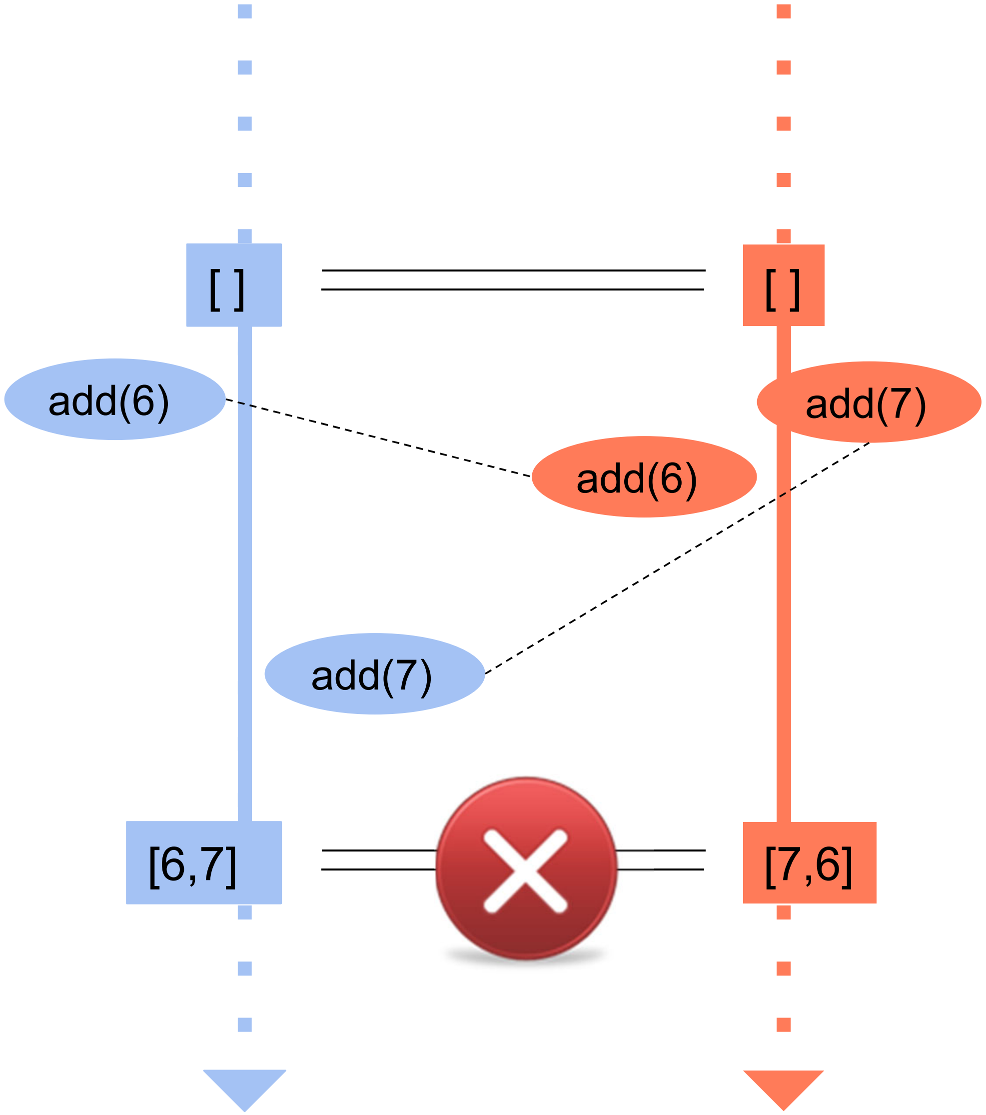
Limitations

Secure

```
      shared = new List()

i = 0              ||  j = 0
while i < h:       ||  while j < 100:
  i += 1           ||    j += 1
atomic:            ||  atomic:
  shared.add(6)    ||    shared.add(7)

      return sort(shared)
```

- ✓ **Prove**: *shared* has the same initial value in both executions
- ✓ **Prove**: the two executions perform the “same” updates
- ✗ **Prove**: the updates commute

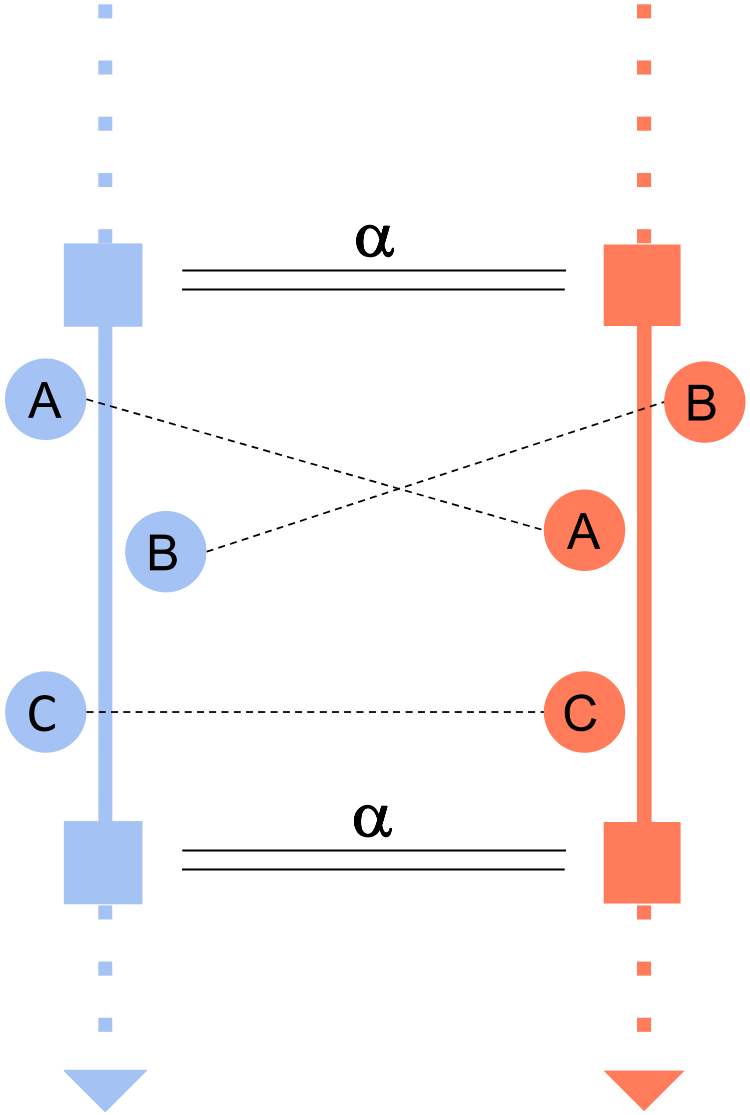
Assume: *shared* has the same final value in both executions



Key idea:
Commutativity modulo abstraction

Commutativity Modulo Abstraction

```
            shared = …
atomic:            ||  atomic:
   shared.A()      ||     shared.B()
atomic:            ||
   shared.C()      ||
                …
```



(0) **Define**: abstraction α of shared data structure

(1) **Prove**: *shared* has the same initial **abstract** value

(2) **Prove**: the two executions perform the “same” updates **modulo abstraction**

(3) **Prove**: the updates commute **modulo abstraction**

Assume: *shared* has the same final **abstract** value in both executions

Commutativity Modulo Abstraction

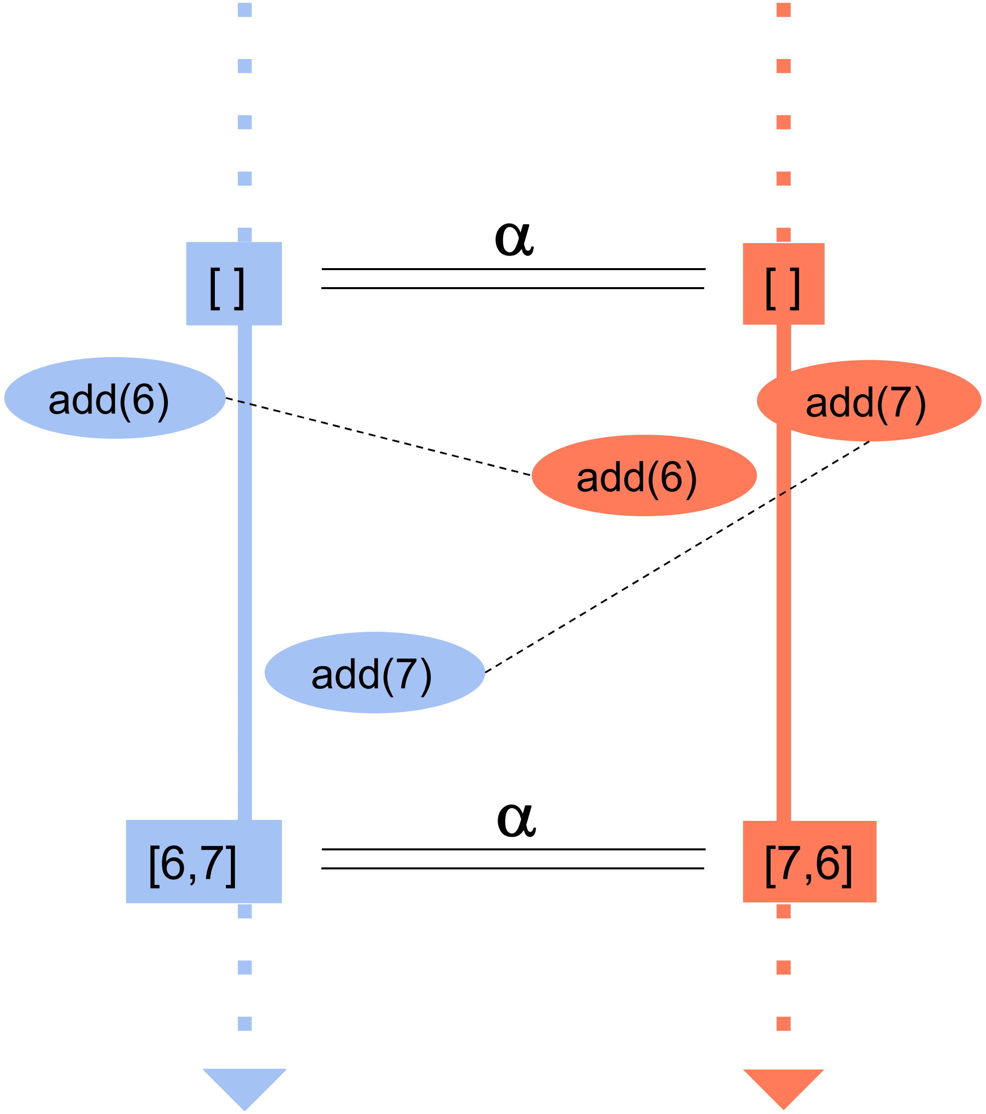
Secure

```
        shared = new List()

i = 0               ||  j = 0
while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
atomic:             ||  atomic:
  shared.add(6)     ||    shared.add(7)

        return sort(shared)
```

(0) **Define**: abstraction α of shared data structure: multiset of integers

- ✓ **Prove**: *shared* has the same initial **abstract** value
- ✓ **Prove**: the two executions perform the “same” updates **modulo abstraction**
- ✓ **Prove**: the updates commute **modulo abstraction**

Assume: *shared* has the same final **abstract** value in both executions



Abstract Commutativity

- Abstraction α is chosen depending on what information about a shared data structure needs to be leaked
- It is part of the resource declaration

```
resource Sequence:
  type        Seq
  invariant   list(x, v)
  abstraction multiset(v)
  actions:
    append(v, e) ≡ v ∘ [e]
```

- Other use cases might abstract a list to its length, sum of elements, mean of elements, etc.

```
           shared = new List()

while i < h:     || while j < 100:
  i += 1         ||   j += 1
atomic:          || atomic:
  shared.add(6)  ||   shared.add(7)

           return sort(shared)
```

✓

Abstract Commutativity: Examples

```
             shared = new Map()

while i < h:         || while j < 100:
  i += 1             ||   j += 1
atomic:              || atomic:
  shared.put(1,8)    || shared.put(1,h)
       return shared.keySet()
```

✓

```
resource Map:
  type        K⇀V
  invariant   map(x, v)
  abstraction dom(v)
  actions:
    put(v, key, val) ≡ v[key↦val]
```

```
             shared = new Map()

if h > 0:           || if h <= 0:
 atomic:            ||  atomic:
  shared.put(1,8)   ||   shared.put(1,h)

         return shared.keySet()
```

✓

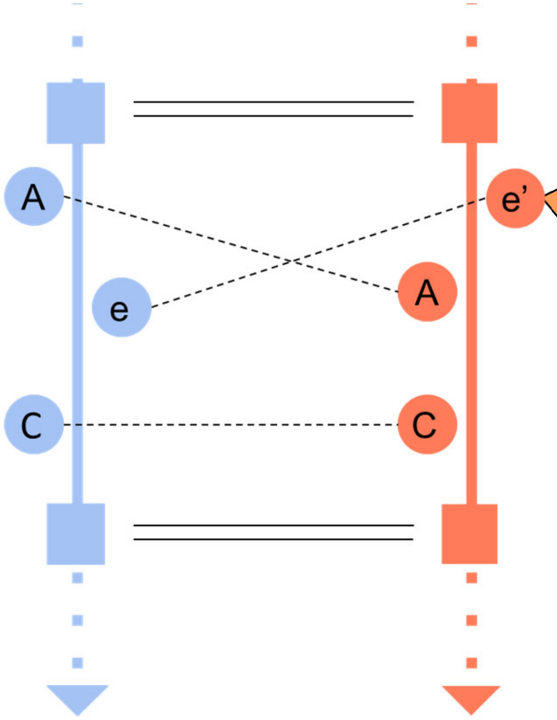
- By the end of the parallel branch, both executions performed exactly one put operation, with key 1
- They performed the same updates modulo abstraction
- The “delayed” check succeeds

Adjusted Proof Obligations

Proof obligation 2:
Same updates in both executions

$$\frac{\vdash \{\ P * acs^1(\emptyset^\#)\ \}\ \ c\ \ \{\ Q * acs^1(args) * Low(args)\ \}}{\vdash \{\ I(x,v) * Low(v) * P\ \}\ \ \textbf{share}\ x\ \textbf{in}\ c\ \ \{\ \exists v' \bullet I(x,v') * Low(v') * Q\ \}}$$

Proof obligation 1:
Same initial abstract value
in both executions:
$Low(\alpha(v))$



Match pairs of actions arguments
e,e’ such that:
$\forall v,v' \bullet \alpha(v) = \alpha(v') \Rightarrow$
$\alpha(f(v,e)) = \alpha(f(v',e'))$

Proof Obligation 3: The updates commute
modulo abstraction

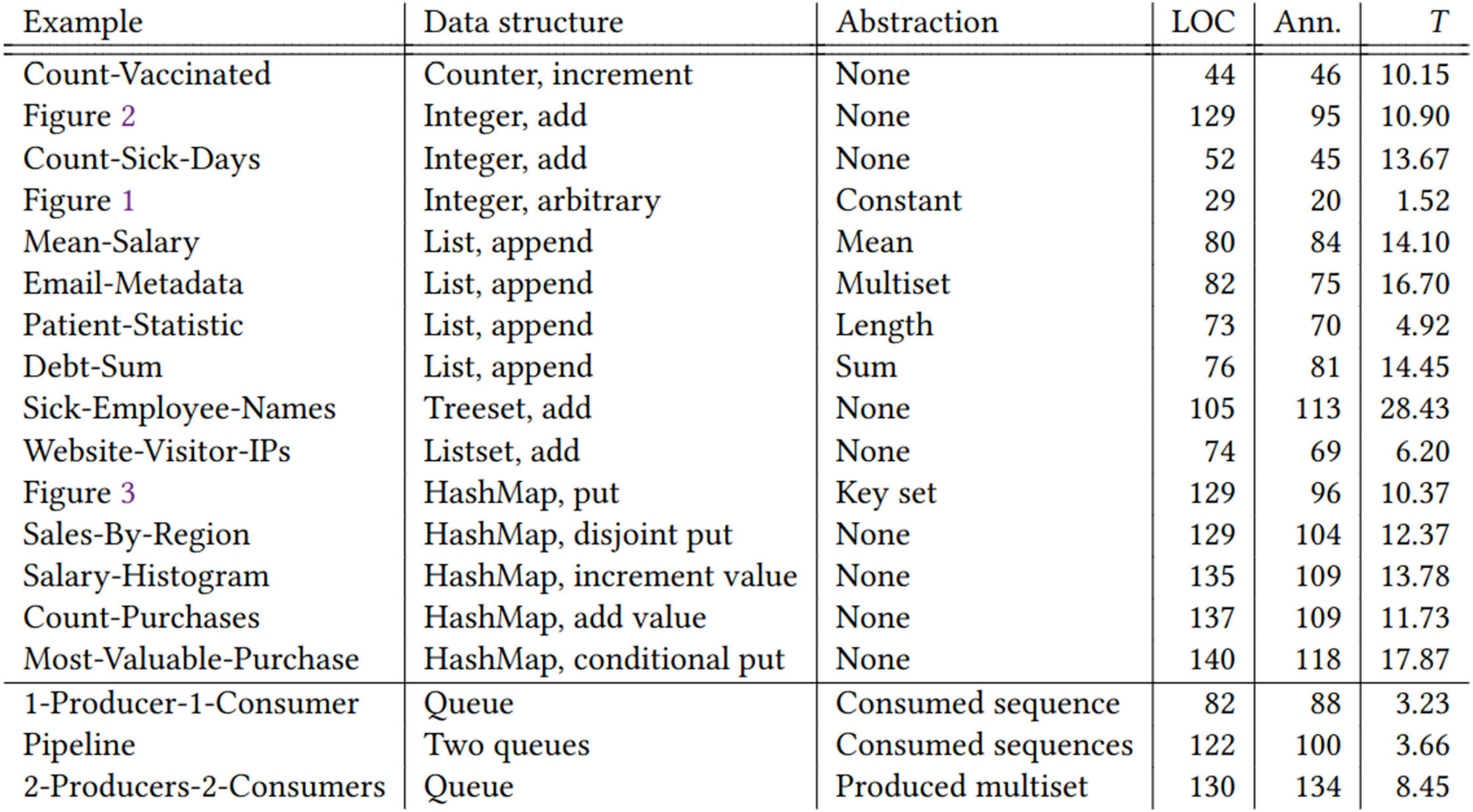
$\forall v,v',e,e' \bullet \alpha(v) = \alpha(v') \Rightarrow$
$\alpha(f(f(v,e),e')) = \alpha(f(f(v',e'),e))$

Implementation: HyperViper

- Automated, SMT-based verifier
    - Based on Viper verification infrastructure
    - Relational reasoning using Modular Product Programs
- Supports dynamic thread creation, multiple shared resources, observable events, etc.

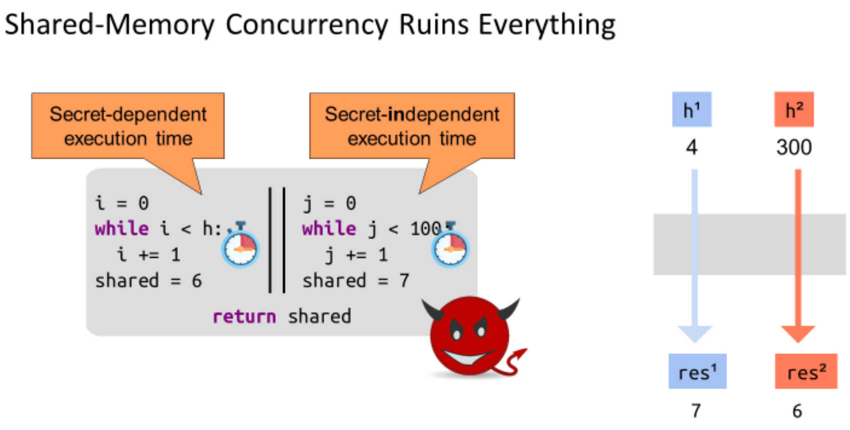
```
lockType IntLock {
 type Int
 invariant(l, v) = [l.lockInt |-> ?cp && [cp.val |-> v]]
 alpha(v): Int = 0  // we abstract to a constant, so everything commutes
 actions = [(SetValue, Int, duplicable)]
 action SetValue(v, arg)
   requires true
 { arg }
 noLabels = 2
}

...

method worker(l: Lock, lbl: Int)
 requires lowEvent && sguard[IntLock,SetValue](l, Set(lbl))
 requires sguardArgs[IntLock,SetValue](l, Set(lbl)) == Multiset[Int]()
 ensures sguard[IntLock,SetValue](l, Set(lbl))
 ensures allPre[IntLock, SetValue](sguardArgs[IntLock,SetValue](l, Set(lbl)))
{

   var v: Int
   v := lbl
   with[IntLock] l performing SetValue(v) at lbl {
       l.lockInt.val := v
   }

}

method print(i: Int)
 requires lowEvent && low(i)
```

Evaluation



Example	Data structure	Abstraction	LOC	Ann.	T
Count-Vaccinated	Counter, increment	None	44	46	10.15
Figure 2	Integer, add	None	129	95	10.90
Count-Sick-Days	Integer, add	None	52	45	13.67
Figure 1	Integer, arbitrary	Constant	29	20	1.52
Mean-Salary	List, append	Mean	80	84	14.10
Email-Metadata	List, append	Multiset	82	75	16.70
Patient-Statistic	List, append	Length	73	70	4.92
Debt-Sum	List, append	Sum	76	81	14.45
Sick-Employee-Names	Treeset, add	None	105	113	28.43
Website-Visitor-IPs	Listset, add	None	74	69	6.20
Figure 3	HashMap, put	Key set	129	96	10.37
Sales-By-Region	HashMap, disjoint put	None	129	104	12.37
Salary-Histogram	HashMap, increment value	None	135	109	13.78
Count-Purchases	HashMap, add value	None	137	109	11.73
Most-Valuable-Purchase	HashMap, conditional put	None	140	118	17.87
1-Producer-1-Consumer	Queue	Consumed sequence	82	88	3.23
Pipeline	Two queues	Consumed sequences	122	100	3.66
2-Producers-2-Consumers	Queue	Produced multiset	130	134	8.45

Conclusion

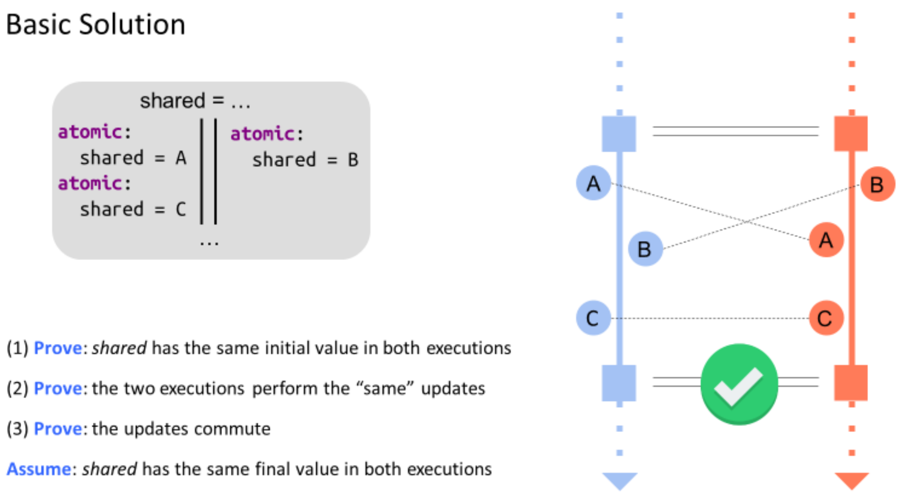


Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-independent execution time

```
i = 0                ||  j = 0
while i < h:         ||  while j < 100:
  i += 1             ||    j += 1
shared = 6           ||  shared = 7
              return shared
```



Basic Solution

```
        shared = ...
atomic:           ||  atomic:
  shared = A      ||    shared = B
atomic:           ||
  shared = C      ||
          ...
```

(1) **Prove**: *shared* has the same initial value in both executions

(2) **Prove**: the two executions perform the “same” updates

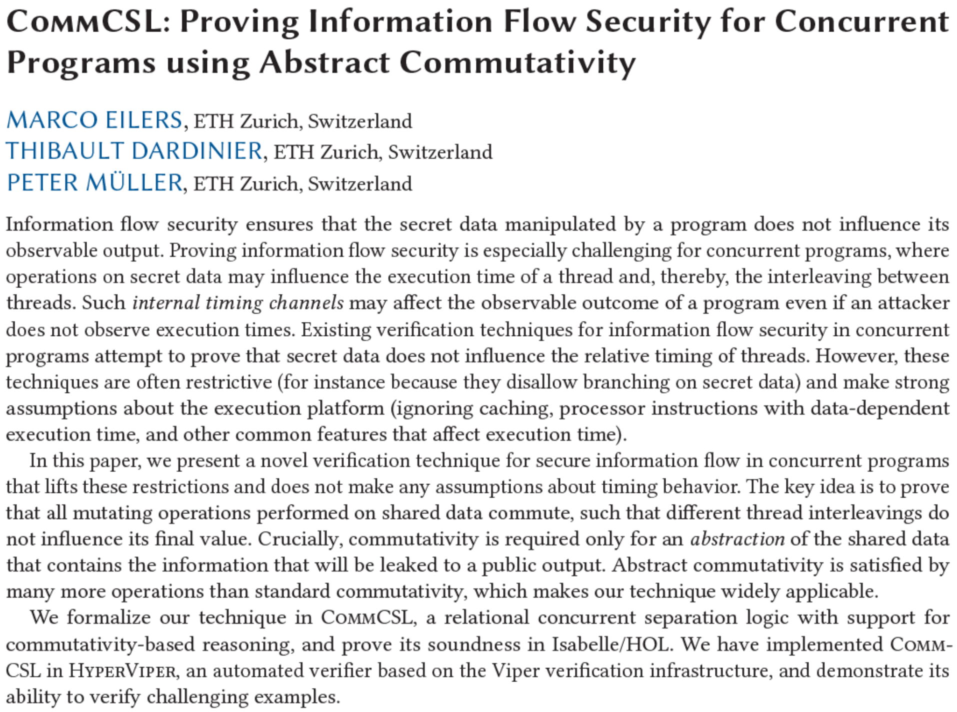
(3) **Prove**: the updates commute

Assume: *shared* has the same final value in both executions

- CommCSL is a relational concurrent separation logic with support for (abstract) commutativity-based information flow reasoning
- Modular reasoning about value sensitivity for concurrent programs
  - Independently of timing, sound on real hardware

More Details in the PLDI 2023 Paper

- Unique actions for asymmetric concurrency
  - Weaker commutativity requirement
- Formalization and soundness proof in Isabelle/HOL



CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity

MARCO EILERS, ETH Zurich, Switzerland
THIBAULT DARDINIER, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland

Information flow security ensures that the secret data manipulated by a program does not influence its observable output. Proving information flow security is especially challenging for concurrent programs, where operations on secret data may influence the execution time of a thread and, thereby, the interleaving between threads. Such *internal timing channels* may affect the observable outcome of a program even if an attacker does not observe execution times. Existing verification techniques for information flow security in concurrent programs attempt to prove that secret data does not influence the relative timing of threads. However, these techniques are often restrictive (for instance because they disallow branching on secret data) and make strong assumptions about the execution platform (ignoring caching, processor instructions with data-dependent execution time, and other common features that affect execution time).

In this paper, we present a novel verification technique for secure information flow in concurrent programs that lifts these restrictions and does not make any assumptions about timing behavior. The key idea is to prove that all mutating operations performed on shared data commute, such that different thread interleavings do not influence its final value. Crucially, commutativity is required only for an *abstraction* of the shared data that contains the information that will be leaked to a public output. Abstract commutativity is satisfied by many more operations than standard commutativity, which makes our technique widely applicable.

We formalize our technique in CommCSL, a relational concurrent separation logic with support for commutativity-based reasoning, and prove its soundness in Isabelle/HOL. We have implemented CommCSL in HyperViper, an automated verifier based on the Viper verification infrastructure, and demonstrate its ability to verify challenging examples.