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

Marco Eilers
Thibault Dardinier
Peter Müller

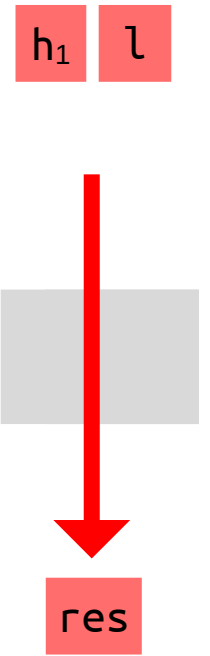
ETH zürich

Value Leaks: Noninterference

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

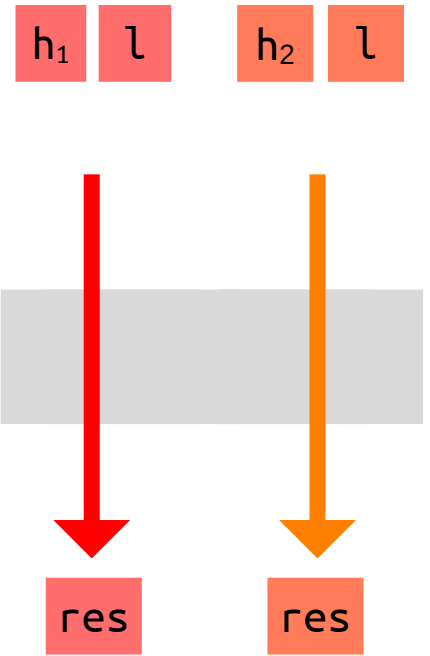
Value Leaks: Noninterference

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



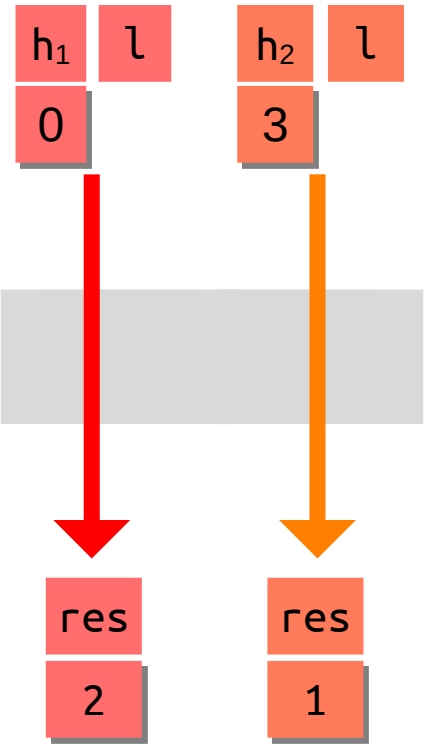
Value Leaks: Noninterference

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



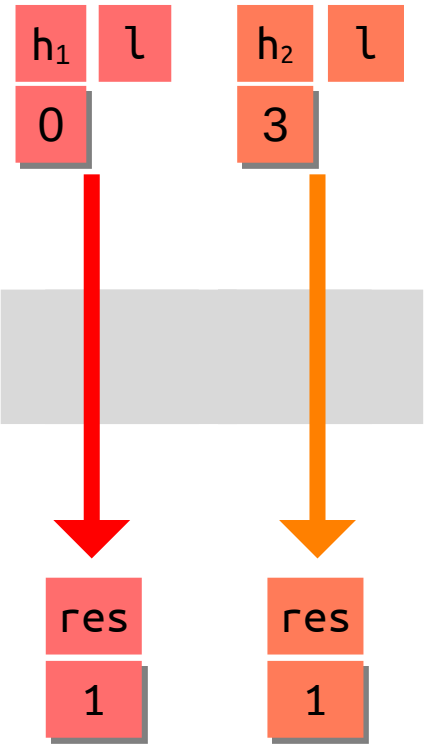
Value Leaks: Noninterference

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



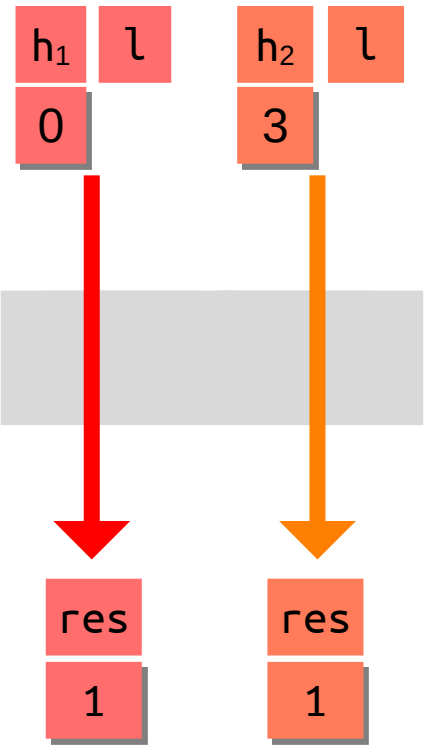
Value Leaks: Noninterference

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



Value Leaks: Noninterference

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

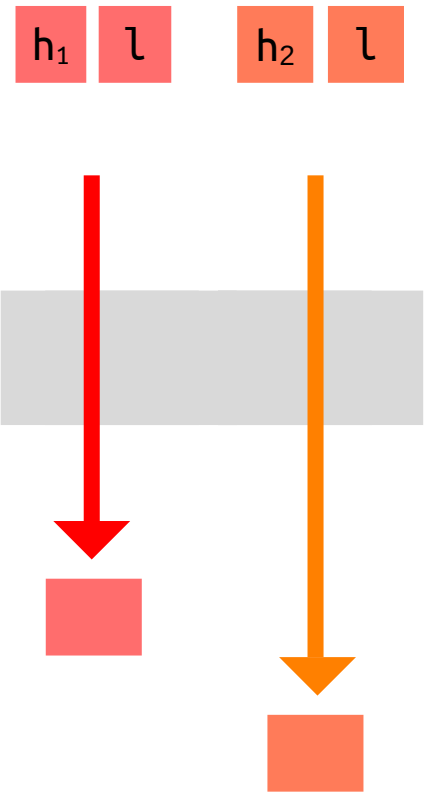


Timing side channels

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

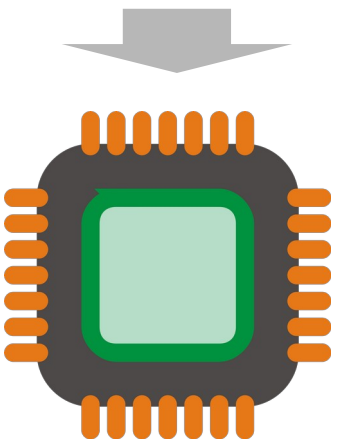
Timing side channels

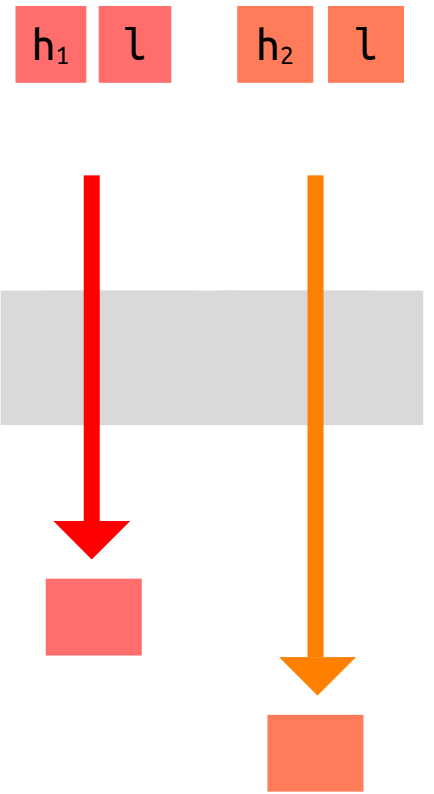
```
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```



Timing side channels

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





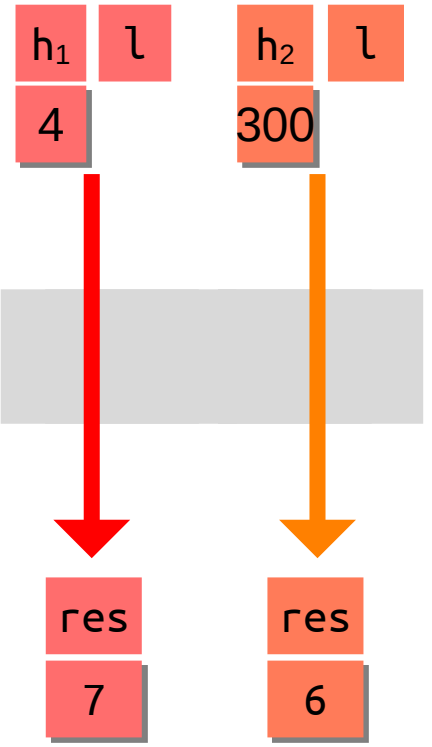
Concurrency
turns
timing channels
into
value channels

Shared-Memory Concurrency

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

          return shared
```

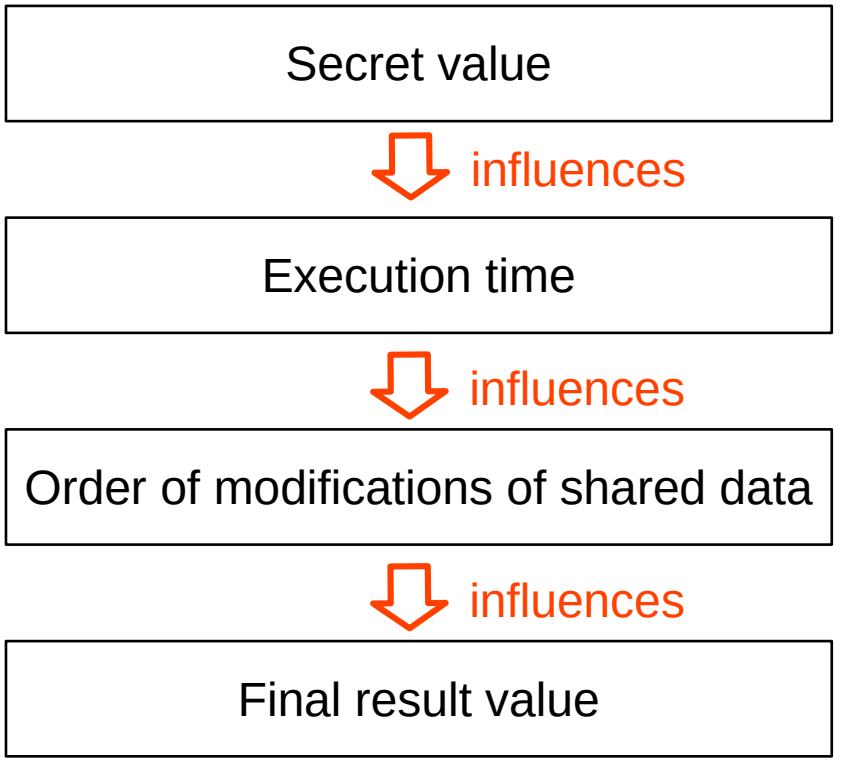
Shared-Memory Concurrency

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

          return shared
```



Shared-Memory Concurrency

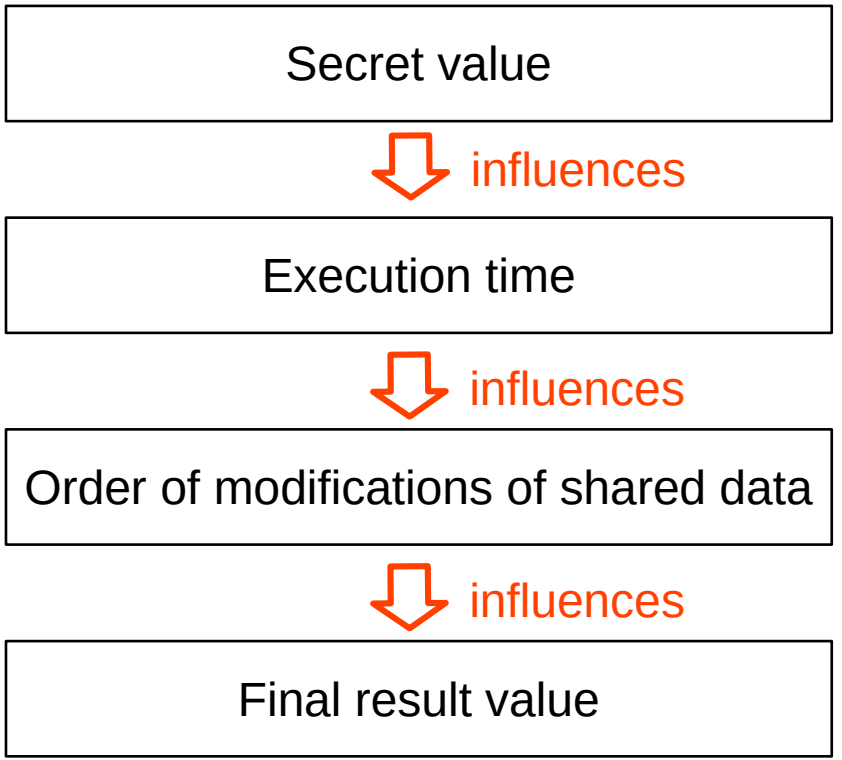
```
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
shared = 6            ||  shared = 7

          return shared
```



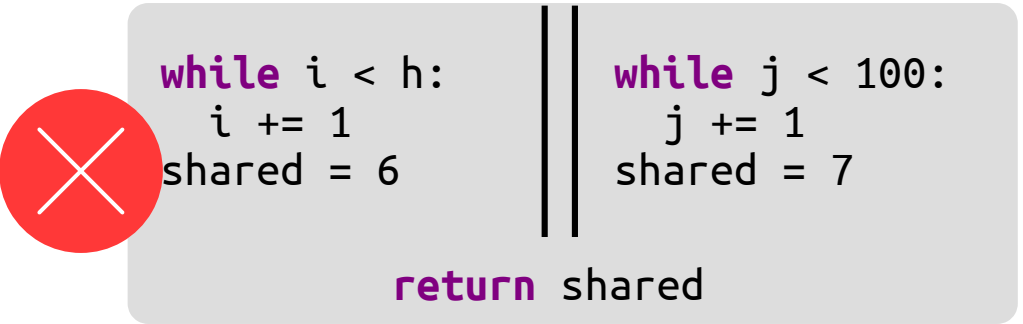
Attacker:
Observes only ***final results***,
not intermediate state or **timing**

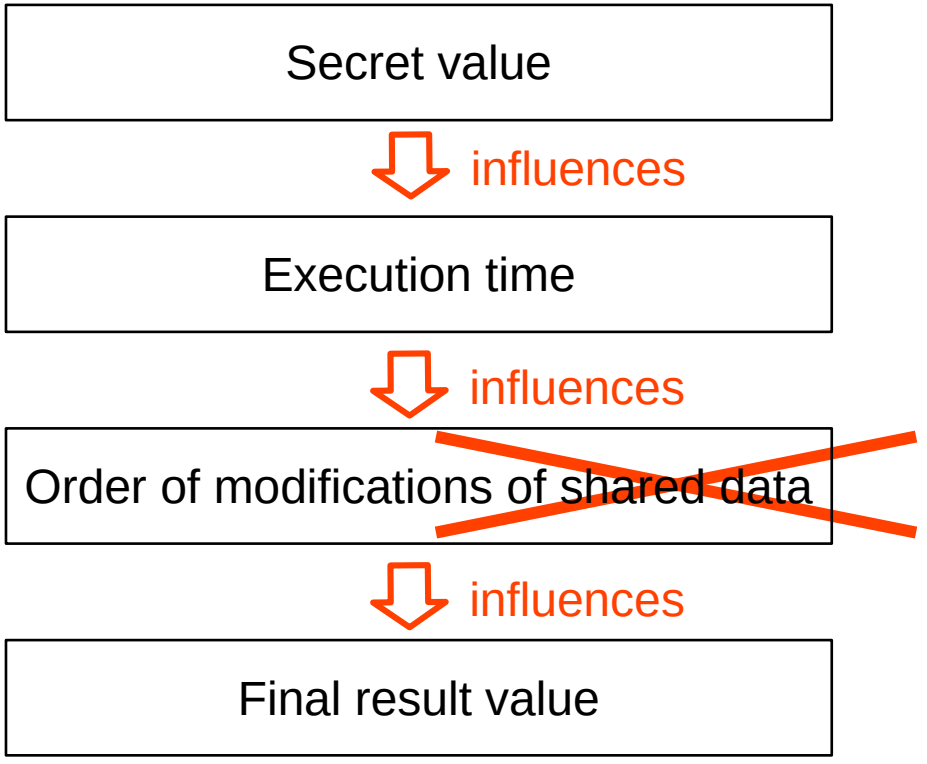
Existing (Modular) Solutions

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

          return shared
```

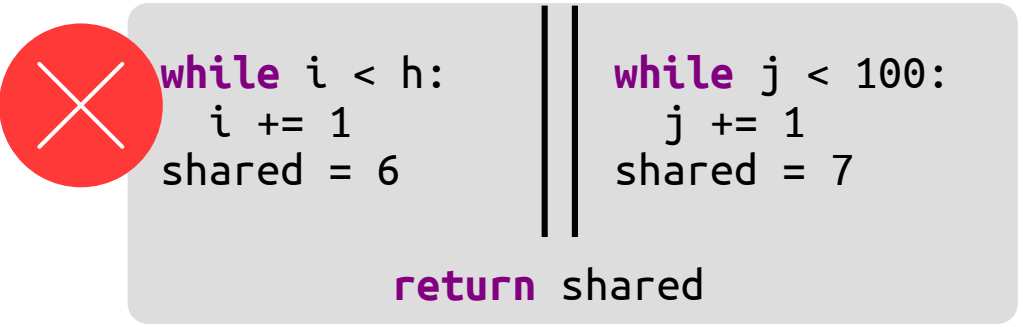


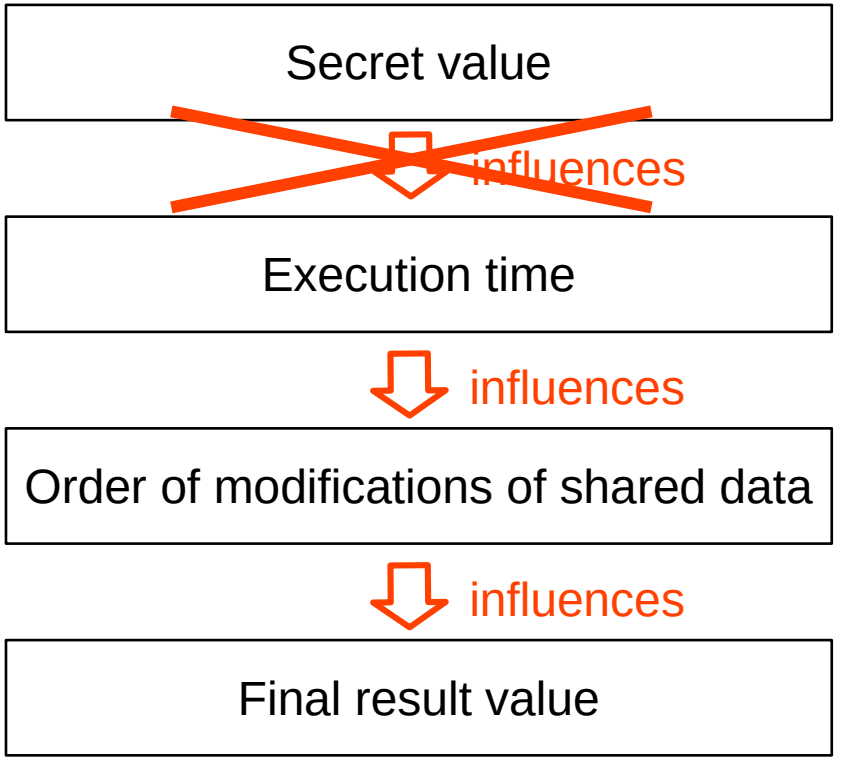
Existing (Modular) Solutions





Existing (Modular) Solutions

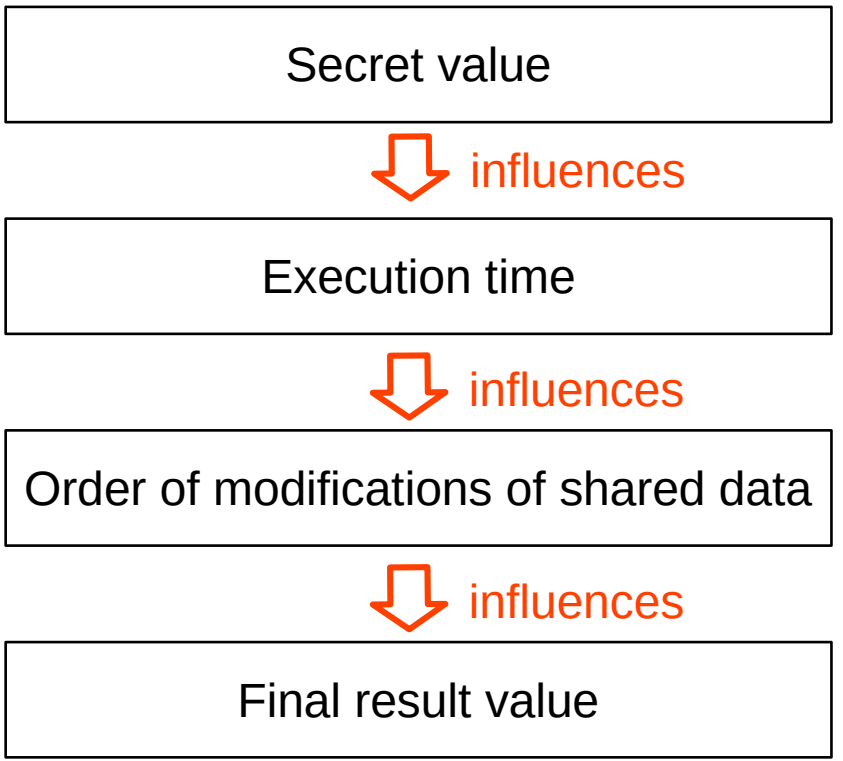




Goal:
Modularly reason about ***values***
in concurrent programs
without reasoning about ***timing***

Our Solution

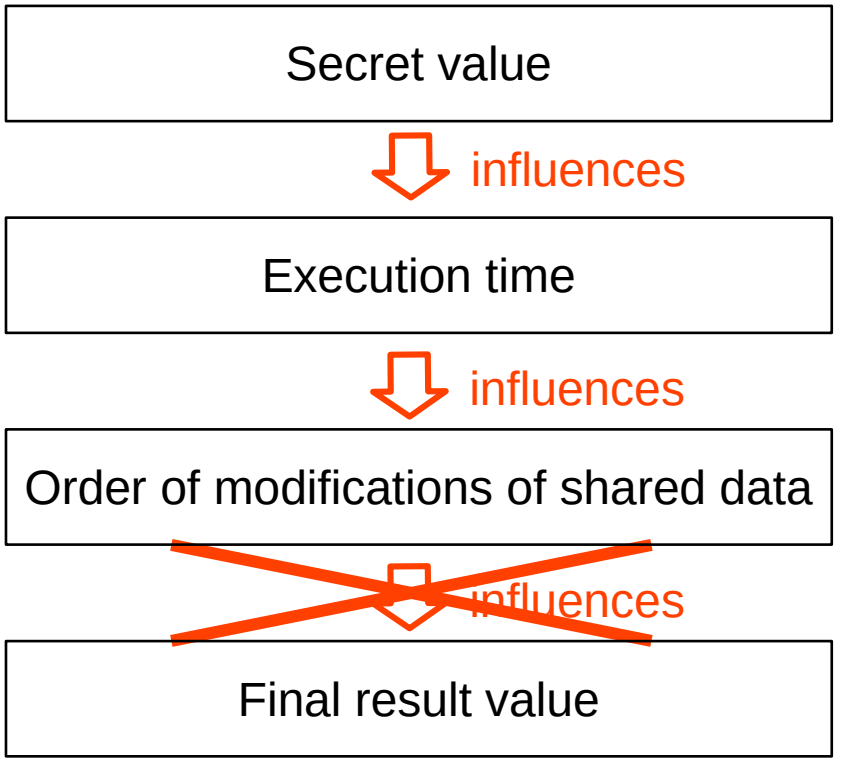
```
          shared = l
while i < h:      ||  while j < 100:
  i += 1          ||    j += 1
atomic:           ||  atomic:
  shared += 6     ||    shared += 7
          return shared
```



Key Idea:
Order does not influence result if modifications ***commute***

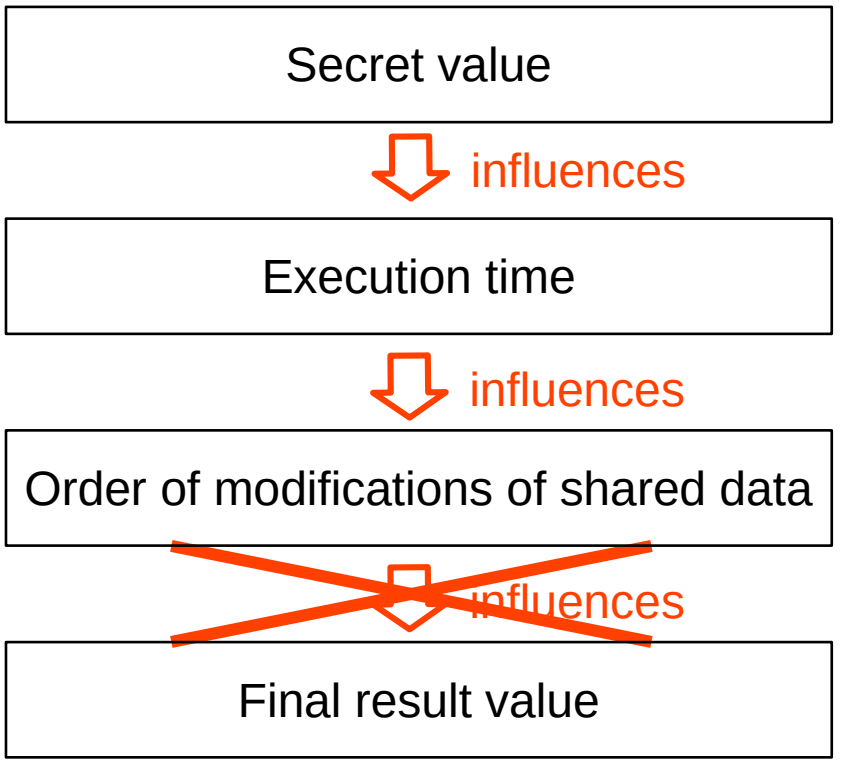
Our Solution

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

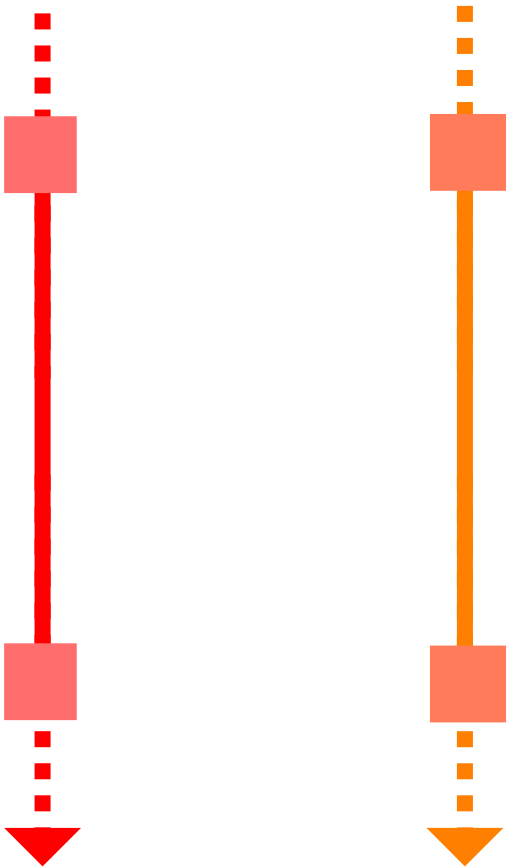


Our Solution

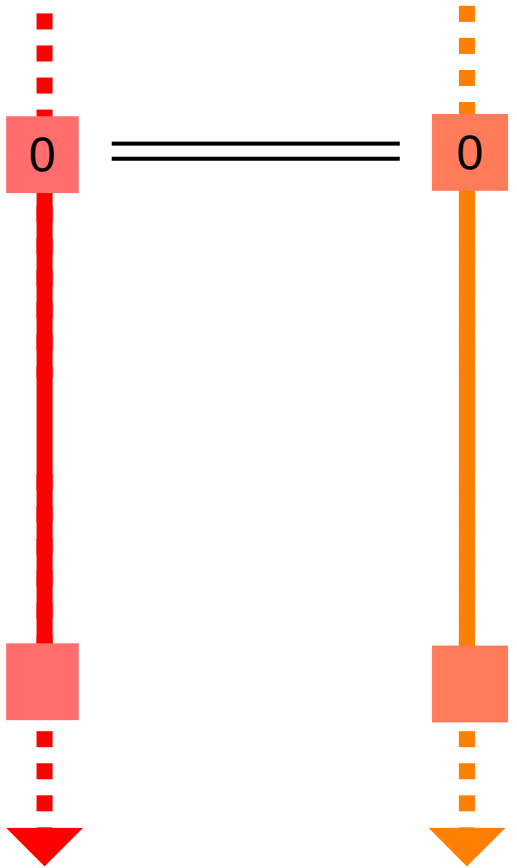
```
          shared = l
while i < h:      ||  while j < 100:
  i += 1          ||    j += 1
atomic:           ||  atomic:
  shared += 6     ||    shared += 7
          return shared
```



Basic Solution

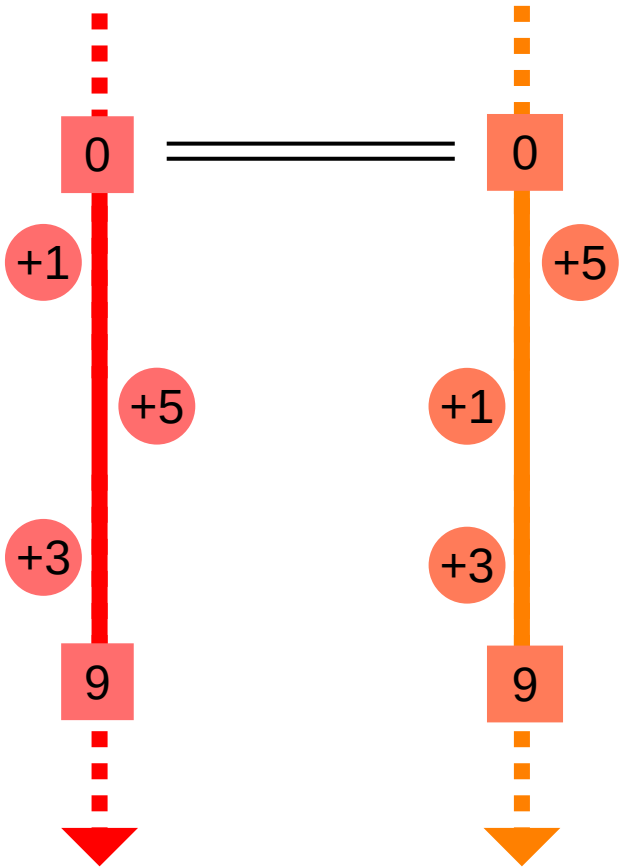


Basic Solution



Prove: Same initial values

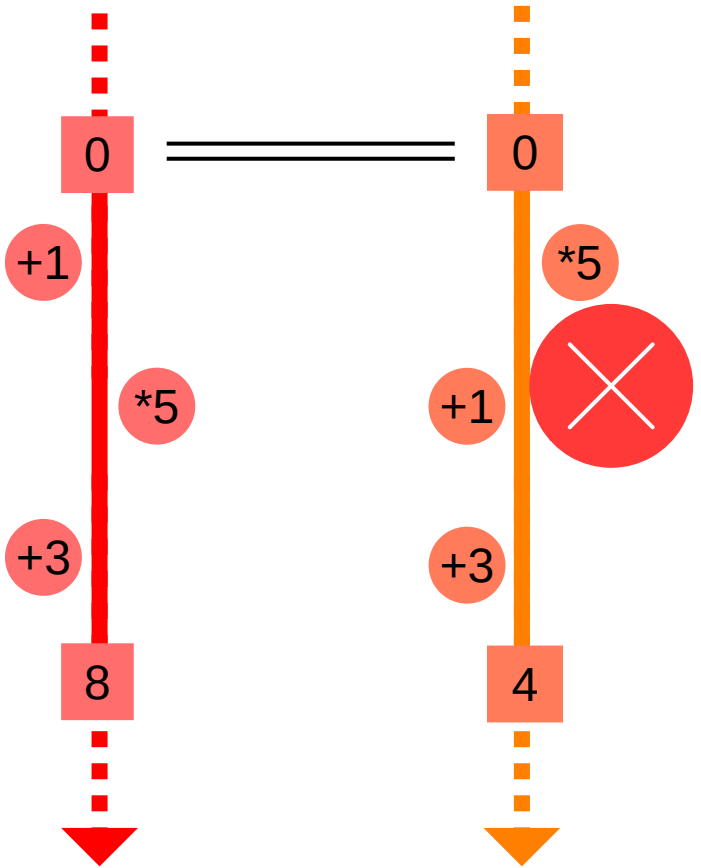
Basic Solution



Prove: Same initial values

Prove: Updates commute

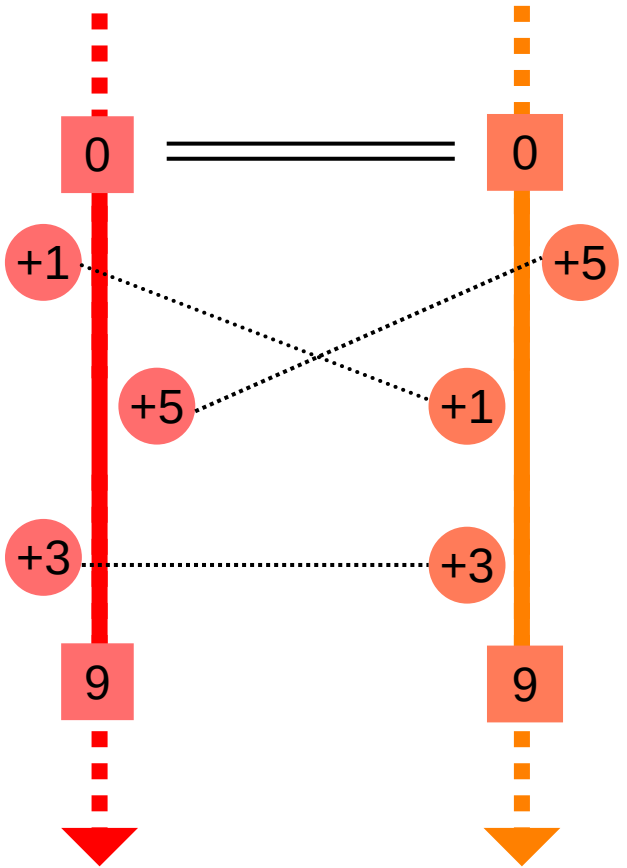
Basic Solution



Prove: Same initial values

Prove: Updates commute

Basic Solution

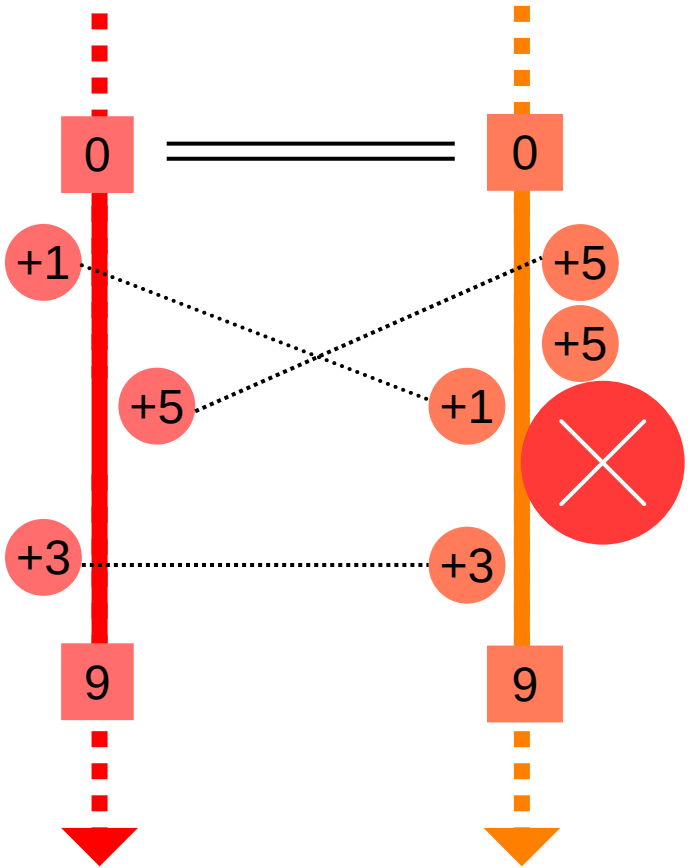


Prove: Same initial values

Prove: Updates commute

Prove: Same updates

Basic Solution

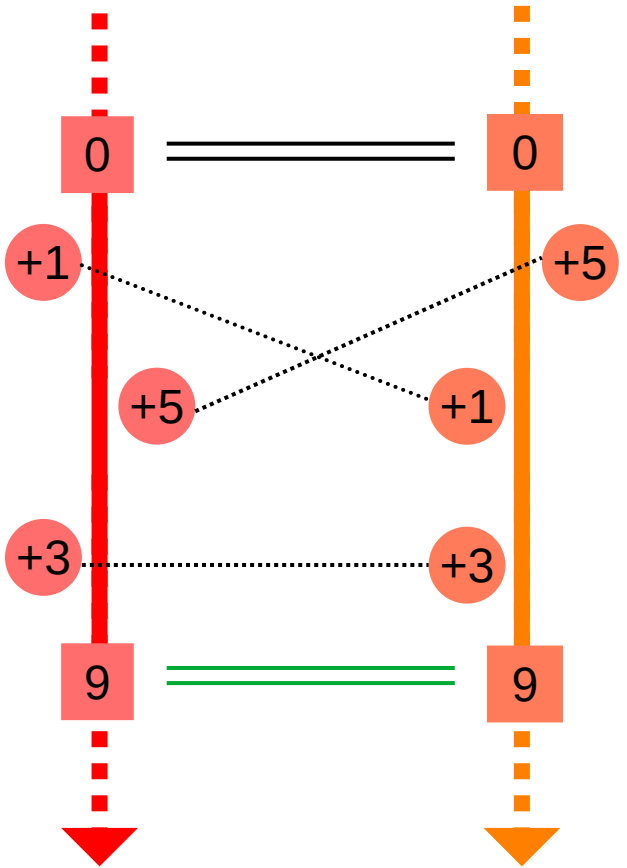


Prove: Same initial values

Prove: Updates commute

Prove: Same updates

Basic Solution



Prove: Same initial values

Prove: Updates commute

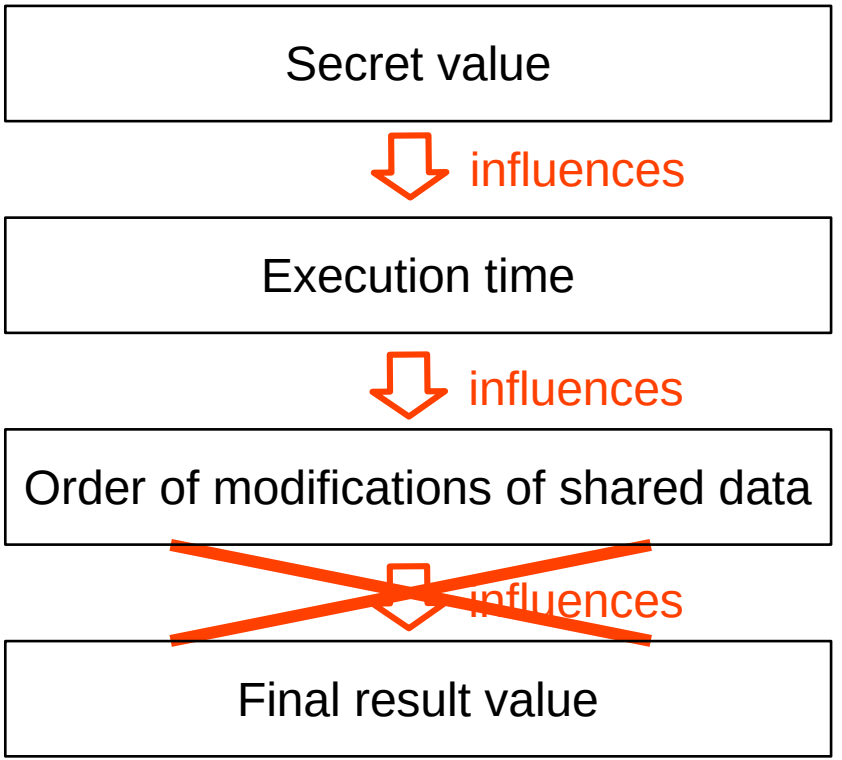
Prove: Same updates

Assume: Same final values

We can do even better.

We can do even better

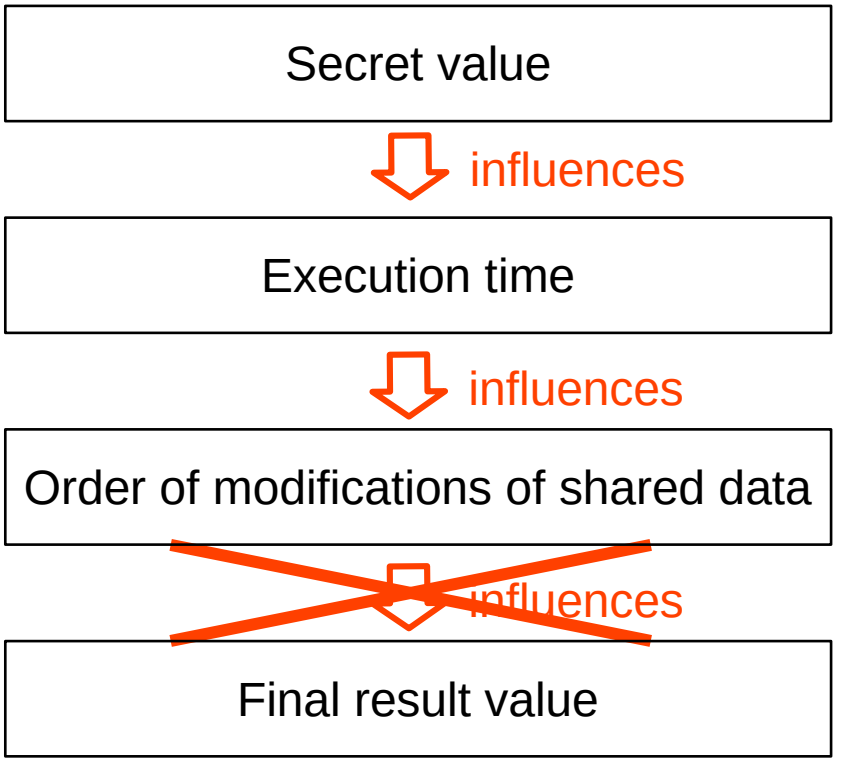
```
          shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
          return sort(shared)
```



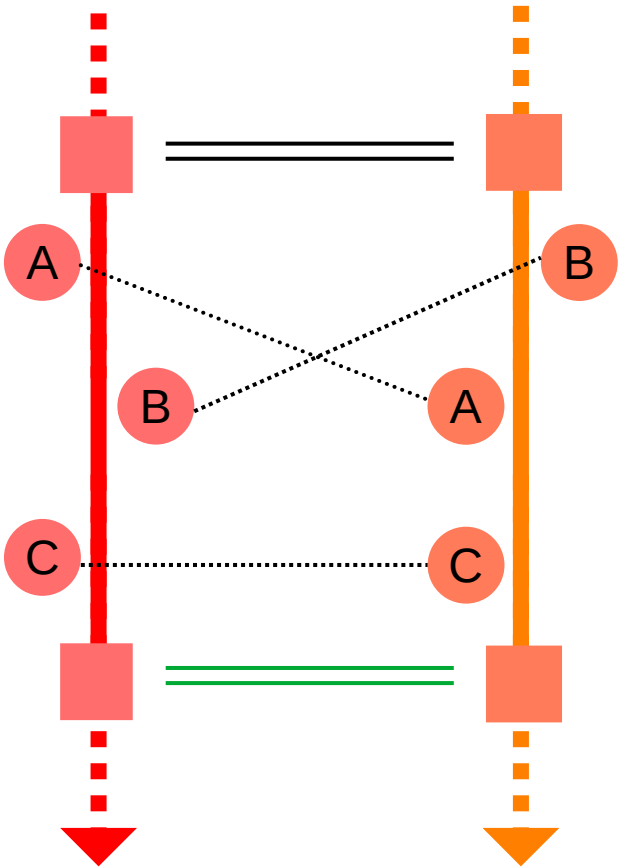
Key Idea:
Commutativity ***modulo abstraction.***

We can do even better

```
          shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
          return sort(shared)
```



Improved Solution



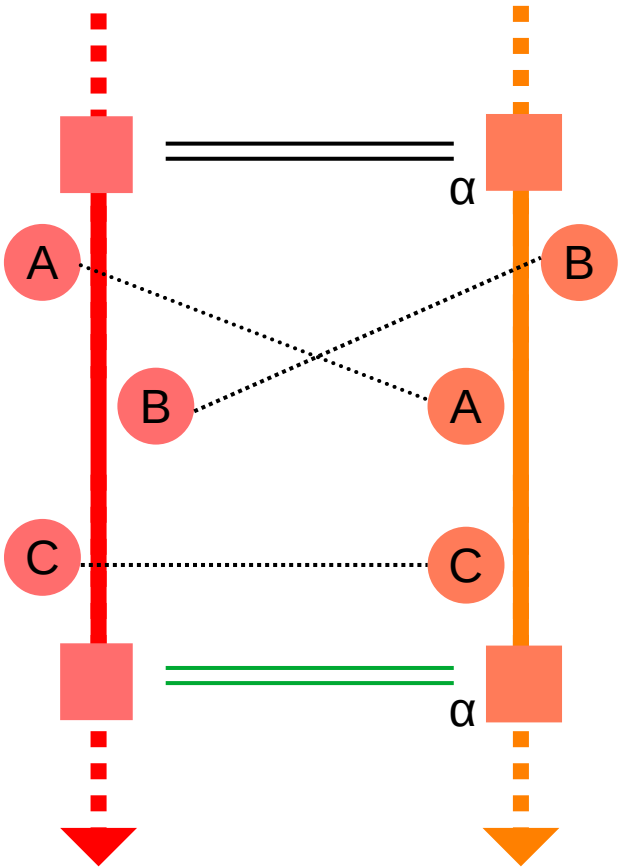
Prove: Same initial values

Prove: Updates commute

Prove: Same updates

Assume: Same final values

Improved Solution

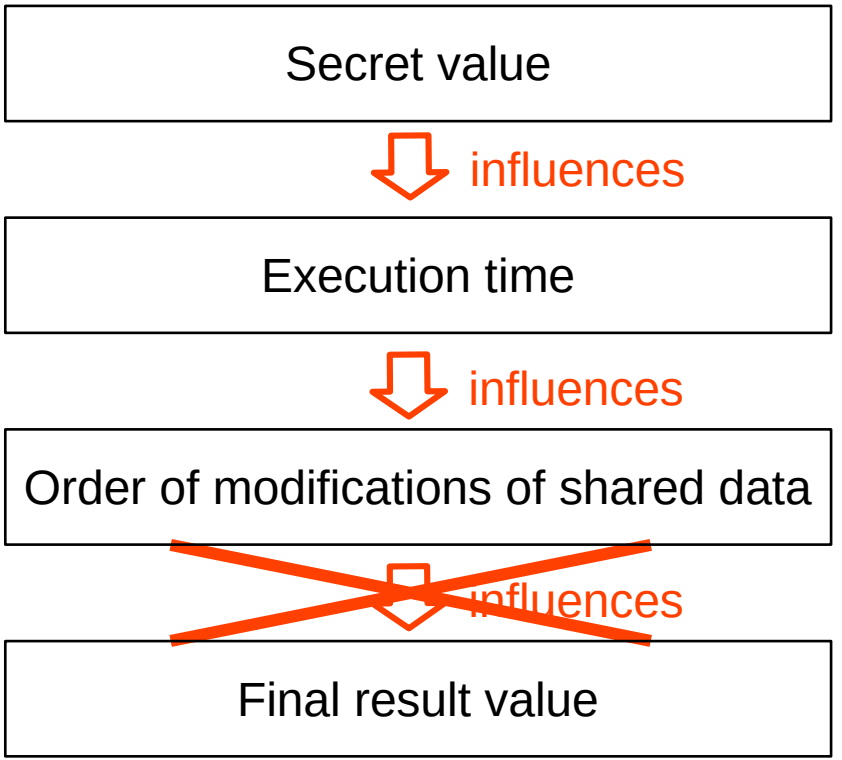


Prove: Same initial values *modulo abstraction*
Prove: Updates commute *modulo abstraction*
Prove: Same updates

Assume: Same final values *modulo abstraction*

We can do even better

```
          shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
          return sort(shared)
```



CommCSL

- Relational concurrent separation logic
- Thread-modular reasoning, mutable heaps
- Support for (abstract) commutativity-based information flow reasoning
- Formalized and proved sound in Isabelle/HOL

$$\frac{\Gamma_\bot = \Gamma \Rightarrow x \notin fv(\Gamma)}{\Gamma_\bot \vdash \{P[e/x]\}x{:=}e\{P\}}\ (\textsc{Assign}) \qquad \frac{x \notin fv(e) \quad \Gamma_\bot = \Gamma \Rightarrow x \notin fv(\Gamma)}{\Gamma_\bot \vdash \{\mathsf{emp}\}x{:=}\mathtt{alloc}(e)\{x \mapsto^1 e\}}\ (\textsc{New})$$

$$\frac{x \notin fv(e_1, e_2) \quad \Gamma_\bot = \Gamma \Rightarrow x \notin fv(\Gamma)}{\Gamma_\bot \vdash \{e_1 \mapsto^r e_2\}x{:=}[e]\{e_1 \mapsto^r e_2 * x = e_2\}}\ (\textsc{Read}) \qquad \frac{}{\Gamma_\bot \vdash \{e_1 \mapsto^1 \_\}[e_1]{:=}e_2\{e_1 \mapsto^1 e_2\}}\ (\textsc{Write})$$

$$\frac{\Gamma_\bot \vdash \{P \wedge b\}c_1\{Q\} \quad \Gamma_\bot \vdash \{P \wedge \neg b\}c_2\{Q\}}{\Gamma_\bot \vdash \{P \wedge \mathrm{Low}(b)\}\mathtt{if}\ (b)\ \mathtt{then}\ \{c_1\}\ \mathtt{else}\ \{c_2\}\{Q\}}\ (\textsc{If1})$$

$$\frac{\Gamma_\bot \vdash \{P \wedge b\}c_1\{Q\} \quad \Gamma_\bot \vdash \{P \wedge \neg b\}c_2\{Q\} \quad \mathit{unary}\ Q}{\Gamma_\bot \vdash \{P\}\mathtt{if}\ (b)\ \mathtt{then}\ \{c_1\}\ \mathtt{else}\ \{c_2\}\{Q\}}\ (\textsc{If2})$$

$$\frac{\Gamma_\bot \vdash \{P \wedge b\}c_I\{P \wedge \mathrm{Low}(b)\}}{\Gamma_\bot \vdash \{P \wedge \mathrm{Low}(b)\}\mathtt{while}\ (b)\ \mathtt{do}\ \{c_I\}\{P \wedge \neg b\}}\ (\textsc{While1}) \qquad \frac{\Gamma_\bot \vdash \{P \wedge b\}c_I\{P\} \quad \mathit{unary}\ P}{\Gamma_\bot \vdash \{P\}\mathtt{while}\ (b)\ \mathtt{do}\ \{c_I\}\{P \wedge \neg b\}}\ (\textsc{While2})$$

$$\frac{\Gamma_\bot \vdash \{P\}c_1\{R\} \quad \Gamma_\bot \vdash \{R\}c_2\{Q\}}{\Gamma_\bot \vdash \{P\}c_1;c_2\{Q\}}\ (\textsc{Seq}) \qquad \frac{}{\Gamma_\bot \vdash \{P\}\mathtt{skip}\{P\}}\ (\textsc{Skip})$$

$$\frac{\Gamma_\bot \vdash \{P_1\}c_1\{Q_1\} \quad \Gamma_\bot \vdash \{P_2\}c_2\{Q_2\} \quad fv(P_1,c_1,Q_1) \cap mod(c_2) = \emptyset \quad fv(P_2,c_2,Q_2) \cap mod(c_1) = \emptyset \quad \Gamma_\bot = \Gamma \Rightarrow fv(\Gamma) \cap mod(c_1,c_2) = \emptyset \quad P_1 \text{ is precise or } P_2 \text{ is precise}}{\Gamma_\bot \vdash \{P_1 * P_2\}c_1 \| c_2\{Q_1 * Q_2\}}\ (\textsc{Par})$$

$$\frac{P \Rightarrow P' \quad \Gamma_\bot \vdash \{P'\}c\{Q'\} \quad Q' \Rightarrow Q}{\Gamma_\bot \vdash \{P\}c\{Q\}}\ (\textsc{Cons}) \qquad \frac{fv(R) \cap mod(c) = \emptyset \quad \Gamma_\bot \vdash \{P\}c\{Q\} \quad P \text{ is precise or } R \text{ is precise}}{\Gamma_\bot \vdash \{P * R\}c\{Q * R\}}\ (\textsc{Frame})$$

$$\frac{x \notin fv(c) \quad \Gamma_\bot = \Gamma \Rightarrow x \notin fv(\Gamma) \quad \Gamma_\bot \vdash \{P\}c\{Q\}}{\Gamma_\bot \vdash \{\exists x.\, P\}c\{\exists x.\, Q\}}\ (\textsc{Exists})$$

$$\frac{\Gamma = \langle \alpha, f_{a_s}, f_{a_u}, I(x)\rangle \quad \Gamma \text{ is valid} \quad I(x) \text{ is unary and precise} \quad \Gamma \vdash \{P * \mathit{sguard}(1,\emptyset^\#) * \mathit{uguard}([\,])\}c\{Q * \mathit{sguard}(1,x_s) * \mathit{PRE}_s(x_s) * \mathit{uguard}(x_u) * \mathit{PRE}_u(x_u)\}}{\bot \vdash \{I(x) * \mathrm{Low}(\alpha(x)) * P\}c\{\exists x'.\, I(x') * \mathrm{Low}(\alpha(x')) * Q\}}\ (\textsc{Share})$$

$$\frac{\Gamma = \langle \alpha, f_{a_s}, f_{a_u}, I(x)\rangle \quad I(x_v) \text{ is unary and precise} \quad x_v \notin fv(P,Q) \quad x_s, x_a, x_v \notin mod(c) \quad \mathit{noguard}(P) \quad \mathit{noguard}(Q) \quad \bot \vdash \{P * I(x_v)\}c\{Q * I(f_{a_s}(x_v, x_a))\}}{\Gamma \vdash \{P * \mathit{sguard}(r, x_s)\}\mathtt{atomic}\ c\{Q * \mathit{sguard}(r, x_s \cup^\# \{x_a\}^\#)\}}\ (\textsc{AtomicShr})$$

$$\frac{\Gamma = \langle \alpha, f_{a_s}, f_{a_u}, I(x)\rangle \quad I(x_v) \text{ is unary and precise} \quad x_v \notin fv(P,Q) \quad x_s, x_a, x_v \notin mod(c) \quad \mathit{noguard}(P) \quad \mathit{noguard}(Q) \quad \bot \vdash \{P * I(x_v)\}c\{Q * I(f_{a_u}(x_v, x_a))\}}{\Gamma \vdash \{P * \mathit{uguard}(x_s)\}\mathtt{atomic}\ c\{Q * \mathit{uguard}(x_s +\!\!+ [x_a])\}}\ (\textsc{AtomicUnq})$$

HyperViper

- Automated, SMT-based verifier
  - Based on Viper and Z3
- User provides specifications, shared resource information
- Implements superset of CommCSL

```
lockType MapLock {
  type MyMap[Int, Int]
  invariant(l, v) = [l.lockMap |-> ?mp && isMap(mp)
                     && v == mapValue(mp)]
  alpha(v): Set[Int] = keys(v)

  actions = [(Put, Pair[Int, Int], duplicable)]

  action Put(v, arg)
    requires low(fst(arg))
  { (put(v, fst(arg), snd(arg))) }
}

method worker(inputs: Seq[Ref], l: Lock, lbl: Int)
  requires low(|inputs|)
  requires sguard[ListLock,Append](l, Set(lbl))
  requires ...
  ensures sguard[ListLock,Append](l, Set(lbl)) && ...
{
  ...
}
```

Open source: https://github.com/viperproject/hyperviper

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

- Modular reasoning about value sensitivity for concurrent programs
  - Independently of timing
  - Sound on real hardware
- Key idea is commutativity modulo abstraction
- Formalized in CommCSL
- Proved sound in Isabelle
- Automated in HyperViper