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

Marco Eilers
Thibault Dardinier
Peter Müller
def compute(h: int, l: int):
    if h > 0:
        res = 1
    else:
        res = 2
    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

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

```assembly
section .text
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, message
    syscall
    ...
```
Values vs. timing
Shared-Memory Concurrency Ruins Everything

while $i < h$:
    $i += 1$
    shared = 6

while $j < 100$:
    $j += 1$
    shared = 7

return shared
Timing channel + Concurrency = Value Channel
Shared-Memory Concurrency Ruins Everything

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

Secret-dependent timing influences order of modifications of shared data, which influences the final result value.
Secret-dependent timing influences order of modifications of shared data, which influences the final result value.
Existing (Modular) Solutions

```
shared = l

while i < h:
    i += 1
    atomic:
    shared += 6

while j < 100:
    j += 1
    atomic:
    shared += 7

return shared
```

Secret-dependent timing influences order of modifications of shared data, which influences the final result value.
Goal:
Reason about \textit{values} in concurrent programs without reasoning about \textit{timing}
Attacker:
Observes *final results*,
not intermediate state or *timing*
Our Solution

shared = l

while i < h:
    i += 1
    atomic:
    shared += 6

return shared

while j < 100:
    j += 1
    atomic:
    shared += 7

Secret-dependent timing influences order of modifications of shared data, which influences the final result value.
Key Idea:
Order does not influence result if modifications commute
Our Solution

shared = l

while i < h:
    i += 1
    atomic:
        shared += 6

return shared

while j < 100:
    j += 1
    atomic:
        shared += 7

Secret-dependent timing influences order of modifications of shared data, which influences the final result value.
Basic Solution
Basic Solution
Basic Solution
Basic Solution
Basic Solution
Basic Solution
Concurrent Separation Logic

shared = l

while i < h:
    i += 1
atomic:
    shared += 6

while j < 100:
    j += 1
atomic:
    shared += 7

return shared
Concurrent Separation Logic

```python
Concurrent Separation Logic

shared = l

while i < h:
    i += 1

atomic:
    shared += 6

return shared

while j < 100:
    j += 1

atomic:
    shared += 7

A +7

B +6
```

Concurrent Separation Logic

\[
\text{shared} = l \qquad \text{share}
\]

\[
\text{while } i < h: \quad i += 1
\]

\[
\text{while } j < 100: \quad j += 1
\]

\[
\text{atomic:} \quad \text{shared} += 6
\]

\[
\text{atomic:} \quad \text{shared} += 7
\]

\[
\text{return } \text{shared}
\]
Concurrent Separation Logic

```
shared = l
share

while i < h:
    i += 1

atomic:
    shared += 6

while j < 100:
    j += 1

atomic:
    shared += 7

unshare
return shared
```
CommCSL

{low(l)}
shared = l
{low(shared)}

atomic:
shared += 6

atomic:
shared += 7

+6

+7

+7, +6

{low(shared)}
return shared
{low(result)}
We can do better.
We can do better

shared = new List()

while i < h:
    i += 1
atomic:
    shared.add(6)

return sort(shared)

while j < 100:
    j += 1
atomic:
    shared.add(7)

Internal timing differences influence order of modifications of shared data, which influences the final result value.
We can do better

```python
shared = new Map()

while i < h:
i += 1
    atomic:
        shared.put(1,6)

return shared.keySet()

while j < 100:
j += 1
    atomic:
        shared.put(1,7)
```

Internal timing differences influence order of modifications of shared data, which influences the final result value.
Key Idea: Commutativity *modulo* abstraction.
Improved Solution
CommCSL

- Relational concurrent separation logic
- Thread-modular reasoning, mutable heaps
- Support for (abstract) commutativity-based information flow reasoning

- Other features:
  - More complete support for non-symmetric concurrency

- Formalized and proved sound in Isabelle/HOL
  - Challenging soundness argument distinct from existing logics
HyperViper

- Automated, SMT-based verifier
  - Based on Viper verification infrastructure and Z3
  - Relational reasoning using Modular Product Programs

- User provides resource specifications, pre- and postconditions, invariants

- Dynamic thread creation, multiple shared resources,

```plaintext
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))) } }
```
## Evaluation

<table>
<thead>
<tr>
<th>Example</th>
<th>Data structure</th>
<th>Abstraction</th>
<th>LOC</th>
<th>Ann.</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>Integer</td>
<td>None</td>
<td>84</td>
<td>39</td>
<td>2.36</td>
</tr>
<tr>
<td>Accumulator-Add</td>
<td>Integer</td>
<td>None</td>
<td>84</td>
<td>39</td>
<td>2.53</td>
</tr>
<tr>
<td>Bitwise-And</td>
<td>Integer</td>
<td>None</td>
<td>84</td>
<td>39</td>
<td>2.48</td>
</tr>
<tr>
<td>All-High</td>
<td>Integer</td>
<td>Constant</td>
<td>84</td>
<td>39</td>
<td>2.49</td>
</tr>
<tr>
<td>List-Append-Mean</td>
<td>List</td>
<td>Mean</td>
<td>103</td>
<td>55</td>
<td>2.62</td>
</tr>
<tr>
<td>List-Append-Multiset</td>
<td>List</td>
<td>Multiset</td>
<td>90</td>
<td>43</td>
<td>2.65</td>
</tr>
<tr>
<td>List-Append-Length</td>
<td>List</td>
<td>Length</td>
<td>88</td>
<td>42</td>
<td>2.60</td>
</tr>
<tr>
<td>Set-Add-Tree</td>
<td>Set</td>
<td>None</td>
<td>210</td>
<td>107</td>
<td>24.52</td>
</tr>
<tr>
<td>Set-Add-List</td>
<td>Set</td>
<td>None</td>
<td>151</td>
<td>67</td>
<td>3.08</td>
</tr>
<tr>
<td>Map-Keyset</td>
<td>Map</td>
<td>Key set</td>
<td>89</td>
<td>40</td>
<td>2.63</td>
</tr>
<tr>
<td>Map-Disjoint</td>
<td>Map</td>
<td>None</td>
<td>93</td>
<td>45</td>
<td>2.05</td>
</tr>
<tr>
<td>Map-Sum</td>
<td>Map</td>
<td>None</td>
<td>118</td>
<td>52</td>
<td>2.71</td>
</tr>
<tr>
<td>Map-Conditional</td>
<td>Map</td>
<td>None</td>
<td>116</td>
<td>46</td>
<td>2.60</td>
</tr>
<tr>
<td>1-Producer-1-Consumer</td>
<td>Queue</td>
<td>Consumed sequence</td>
<td>120</td>
<td>59</td>
<td>1.97</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Two queues</td>
<td>Consumed sequences</td>
<td>162</td>
<td>73</td>
<td>2.40</td>
</tr>
<tr>
<td>2-Producers-2-Consumers</td>
<td>Queue</td>
<td>Produced multiset</td>
<td>213</td>
<td>111</td>
<td>6.23</td>
</tr>
</tbody>
</table>
Conclusion

● Modular reasoning about value sensitivity for concurrent programs
  ● Independently of timing
  ● Sound on real hardware

● Key idea is to exploit commutativity modulo abstraction

● Proved sound in Isabelle, automated in prototype verifier

● Should be on arXiv in a couple of weeks

● Future work
  ● Fine-grained concurrency
  ● Static analysis etc.