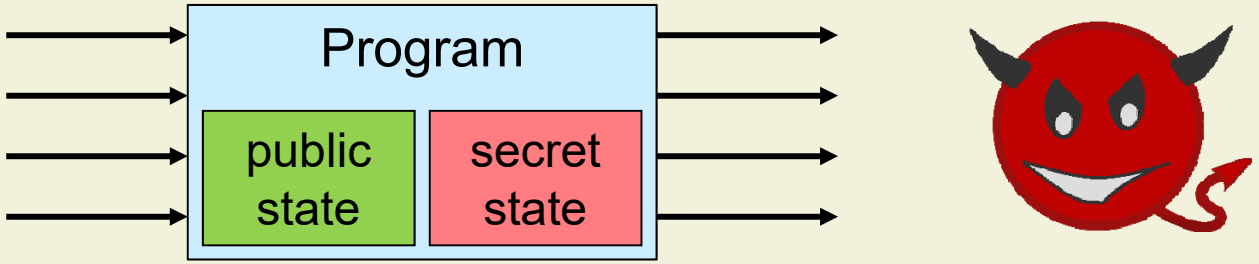
Modular Verification of Secure Information Flow

Peter Müller
ETH Zurich

Joint work with
Marco Eilers and Sam Hitz

Secure Information Flow

- Programs maintain secret state such as crypto keys



- High-level goal:
  Verify that attackers are not able to learn secrets by interacting with the implementation

Traditional Non-Interference

- Classify all variables as either high (secret) or low (public)
- Assume attacker can observe values of low variables
- A statement s is information-flow secure if:

$$\forall \sigma_1, \sigma_2 \bullet \sigma_1 \equiv_{low} \sigma_2 \wedge \langle s, \sigma_1 \rangle \rightarrow \sigma'_1 \wedge \langle s, \sigma_2 \rangle \rightarrow \sigma'_2 \Rightarrow \sigma'_1 \equiv_{low} \sigma'_2$$

Self-Composition

- Reduce non-interference to a trace property
- Strengths
  - Supports off-the-shelf verifiers
- Weaknesses
  - Relational specifications
  - Non-modular

```
while( l < 10 )
  invariant low( l )
{ l := l + 1 }
```

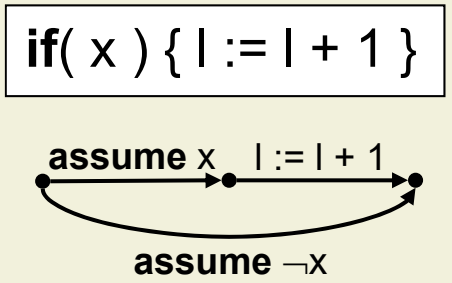
```
assume l == l'
while( l < 10 ) { l := l + 1 }
while( l' < 10 ) { l' := l' + 1 }
assert l == l'
```

```
int foo( x )
  ensures   low( x ) ⇒
            low( result )
```

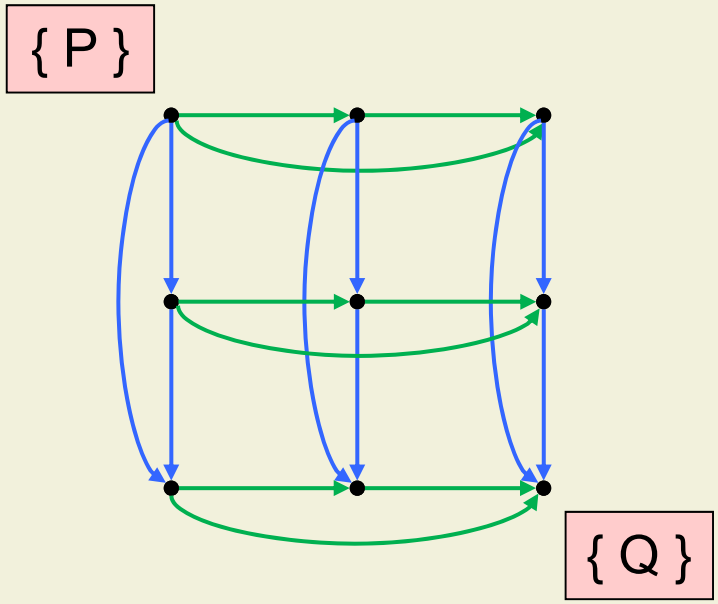
```
assume l == l'
a := foo( l )
a' := foo( l' )
assert a == a'
```

Product Programs

- Execute two versions of the program interleaved rather than in sequence



- Strengths
  - Supports off-the-shelf verifiers
- Weaknesses
  - Relational specifications
  - Non-modular

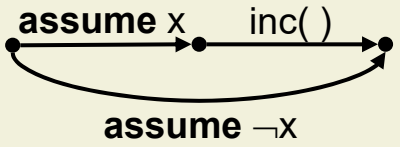


Relational Specifications

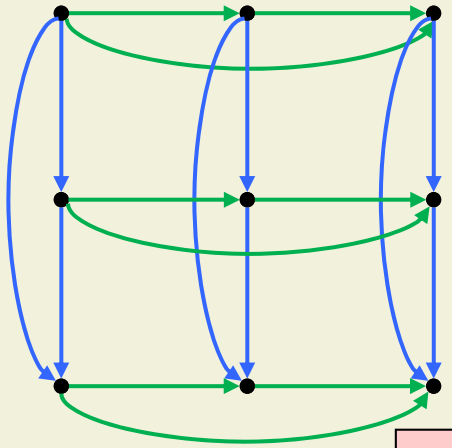
- Method specifications

```
inc( )
  requires low( g )
  ensures low( g )
{ g := g + 1 }
```

if(x) { inc( ) }



{ x = x' ∧ g = g' }



{ g = g' }

Modular Product Programs

- Introduce boolean ghost variables p, p' to track which execution is active

- Make all operations conditional on p,p'

- p,p' are initially true

```
inc( )
{ g := g + 1 }
```

```
inc( p, p' ) {
  if( p ) { g := g + 1 }
  if( p' ) { g' := g' + 1 }
}
```

Modular Product Programs: Definition

[[s]] (p, p')

```
if( p ) { s }
if( p' ) { s' }
```

[[**if**(b) { s_0 } **else** { s_1 }]] (p, p')

$p_0 := p \wedge b$
$p_0' := p' \wedge b'$
$p_1 := p \wedge \neg b$
$p_1' := p' \wedge \neg b'$
[[s_0]] (p_0, p_0')
[[s_1]] (p_1, p_1')

[[**while**(b) **invariant** J { s }]] (p, p')

while($p \wedge b \vee p' \wedge b'$)
  invariant $(p \Rightarrow J) \wedge (p' \Rightarrow J')$
{
  $p_0 := p \wedge b$
  $p_0' := p' \wedge b'$
  [[s]] (p_0, p_0')
}

Interpreting Relational Specifications

- Relational specifications apply only if both program executions proceed synchronously

```
if( x ) { inc( ) }
```

$p_0 := p \wedge x$
$p_0' := p' \wedge x'$
$inc(p_0, p_0')$

```
inc( )
  requires low( g )
  ensures low( g )
{ g := g + 1 }
```

```
inc( p, p' )
  requires p∧p' ⇒ g=g'
  ensures p∧p' ⇒ g=g'
{
  if( p ) { g := g + 1 }
  if( p' ) { g' := g' + 1 }
}
```

Observable Output

```
print( x )
  requires low( x )
  requires lowEvent
```

```
while( y < 10 )
  invariant lowEvent
  invariant low( y )
{ print( 5 ); y := y + 1 }
```

```
print( p, p', x, x' )
  requires p∧p' ⇒ x=x'
  requires p=p'
```

```
while( p ∧ y<10 ∨ p' ∧ y'<10 )
  invariant p=p'
  invariant p∧p' ⇒ y=y'
{
  p0 := p ∧ y < 10
  p0' := p' ∧ y' < 10
  print( p0, p0', 5, 5 ); …
}
```

Declassification

- Many programs intentionally leak some secret information

```
bool equals( a, b )
  ensures   low( a ) ∧ low( b ) ⇒
            low( result )
```

```
b := equals( passwd, input )
declassify( b )
if( b ) { … }
else  { print( error ) }
```

```
assume p∧p' ⇒ b=b'
```

Termination

- Attackers can observe termination

```
while( y ≠ 0 )
{ y := y - 1 }
```

- Specify for each loop a precise termination condition

```
while( y ≠ 0 )
  terminates 0 ≤ y
{ y := y - 1 }
```

- Verify for each loop that the two executions either both terminate or both run forever

Termination: Encoding

```
while( y ≠ 0 )
  terminates 0 ≤ y
  decreases rank
{ y := y – 1 }
```

```
if( p )   { t := ( 0 ≤ y ) }
if( p' )  { t' := ( 0 ≤ y' ) }
assert p ∧ ¬t  ⇔  p' ∧ ¬t'
while( p∧y ≠ 0  ∨  p'∧y' ≠ 0 )
  invariant p ⇒ ( ¬t ⇒ y ≠ 0 )
  invariant p' ⇒ ( ¬t' ⇒ y' ≠ 0 )
{
  p0 := p ∧ y ≠ 0
  p0' := p' ∧ y' ≠ 0
  if( p0 ) { y := y – 1 }
  if( p0' ) { y' := y' – 1 }
  assert p0 ∧ t ⇒ rank decreased
  assert p0' ∧ t' ⇒ rank' decreased
}
```

Summary

- Modular verification of secure information flow
  - Modular product programs
  - Relational specifications
- Implemented in Viper
  - Based on powerful program logic
  - Heap data structures, race-free concurrency
- Generalization to k-safety properties