SecWasm: Information Flow Control for WebAssembly

Iulia Bastys Maximilian Algehed Alexander Sjösten Andrei Sabelfeld

CHALMERS

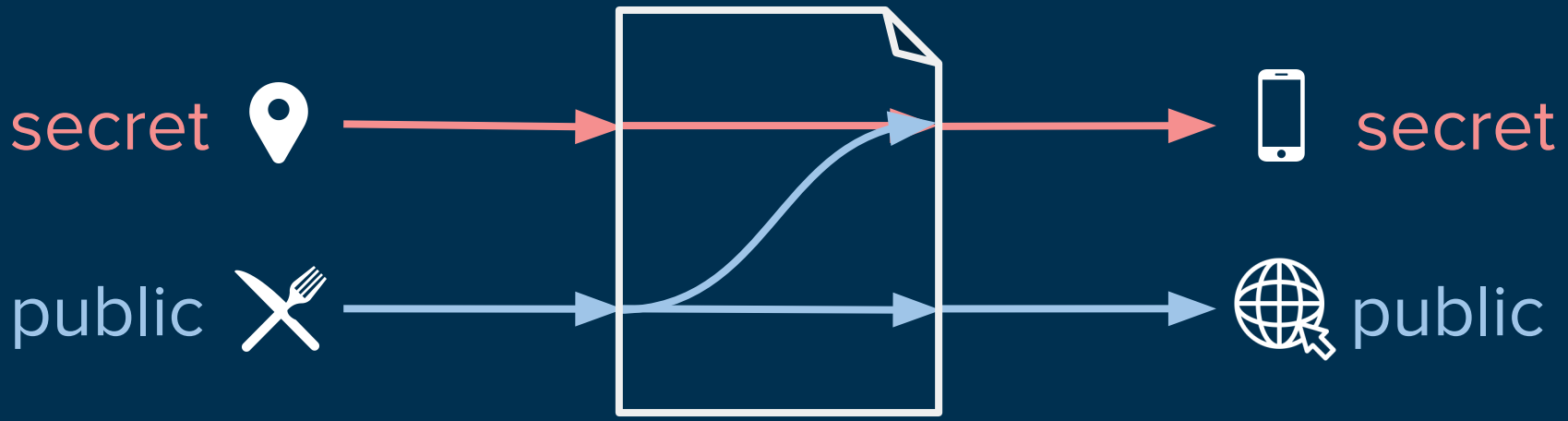
TU WIEN TECHNISCHE UNIVERSITÄT WIEN Vienna University of Technology

SAS | December 6th, 2022

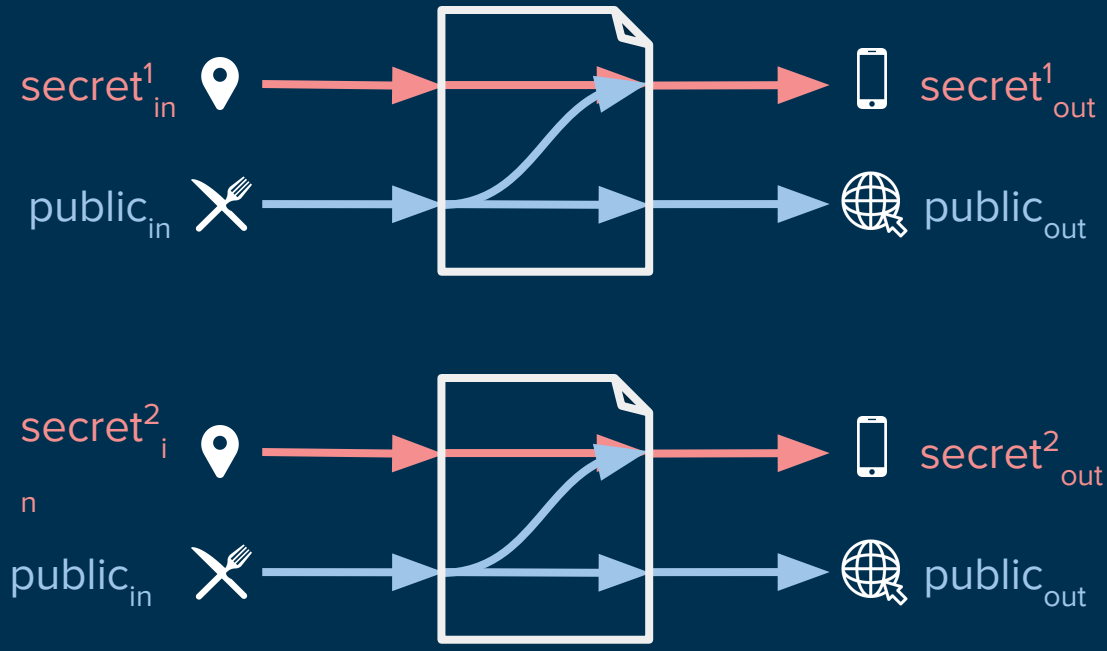
SecWasm: Information Flow Control for WebAssembly

1. IFC
2. Wasm
3. SecWasm

Noninterference



NI - property of traces



- **inputs/outputs**
- **memory locations**
- **…**

- **attacker view**

Tracking flows

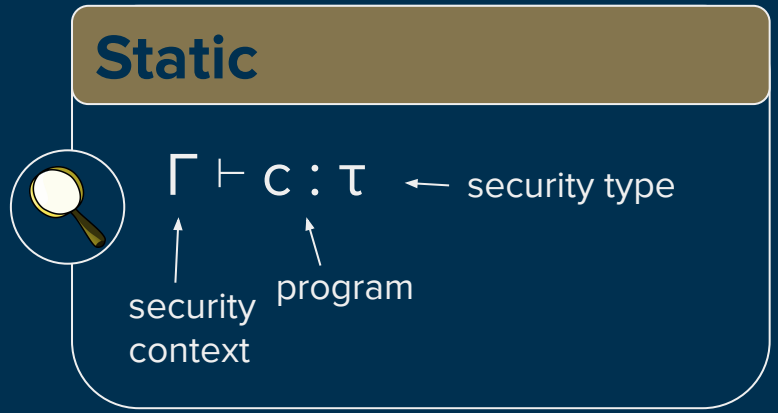
```
x^public := y^secret

out_public(z^secret)
```

Explicit flows

```
if (y^secret) then
    x^public := true
else
    x^public := false
```

Implicit flows

Enforcement mechanisms



Enforcement mechanisms

Static

$\Gamma \vdash c : \tau$

Dynamic

(c, st, S)

program state

security state

Enforcement mechanisms

Static

$\Gamma \vdash c : \tau$

Dynamic

$(c, st, S) \longrightarrow (c', st', S')$

Enforcement mechanisms

Static

$\Gamma \vdash c : \tau$

Dynamic

$(c, st, S) \xrightarrow{e} (c', st', S')$

attacker observation

Enforcement mechanisms

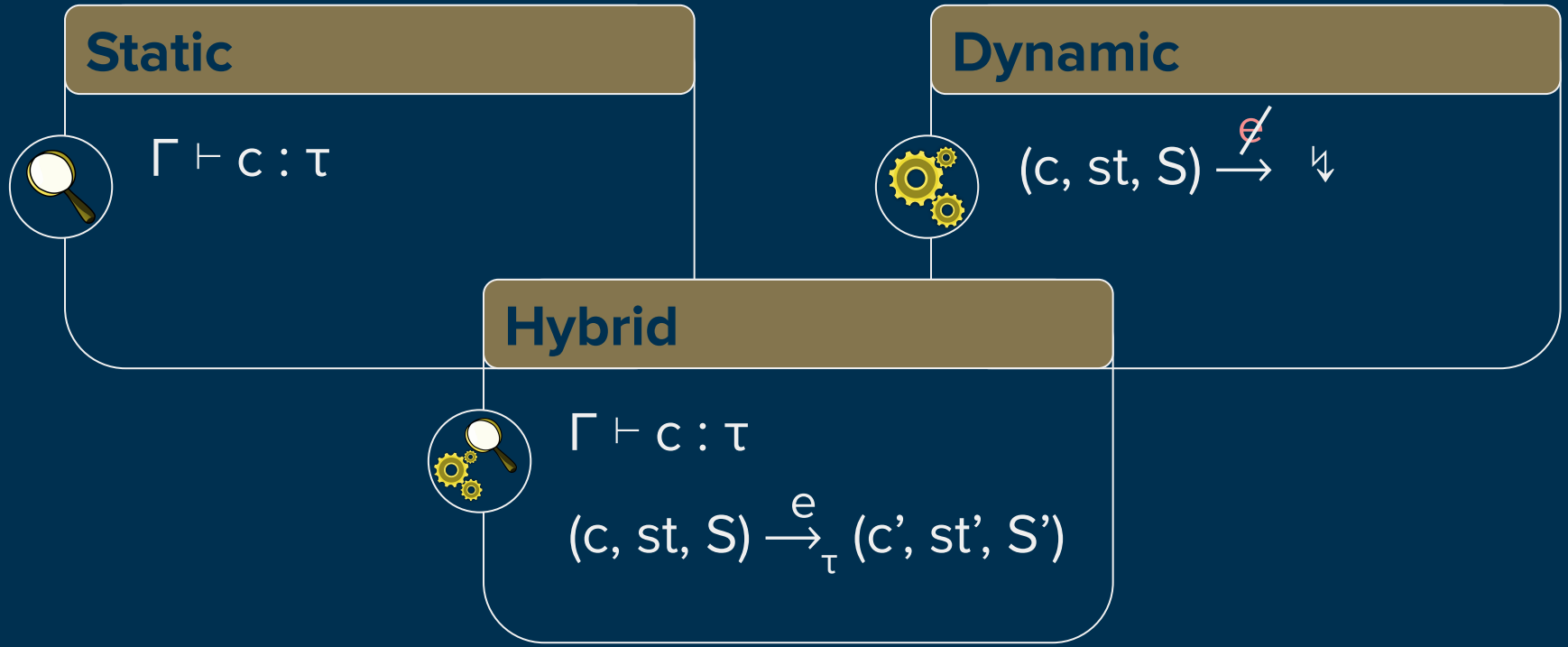
Static

$\Gamma \vdash c : \tau$

Dynamic

$(c, st, S) \not\longrightarrow \lightning$

Enforcement mechanisms



Wasm

- Structured control flow
- Unwinding operand stack
- Unstructured linear memory
- Well-defined type system

$$C \vdash expr : t^n \rightarrow t^m$$

Control flow

ctrl **::= nop | unreachable**
| block *bt expr* **end**
| loop *bt expr* **end**
| if *bt expr* **else** *expr* **end**
| br *i* **| br_if** *i* **| br_table** i^+
| return | call *i*
| call_indirect *ft*

Structured control flow

```
ctrl ::= nop | unreachable
       | block bt expr end
       | loop bt expr end
       | if bt expr else expr end
       | br i | br_if i | br_table i+
       | return | call i
       | call_indirect ft
```

✚ unwinding operand stack

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

← pushes address `ax` of `x` on the stack

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

reads x from memory

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

enters scope of block `$0`
pushes `$0` on the stack

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

takes one argument, returns one value

enters scope of block `$0`
pushes `$0` on the stack

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

enters scope of block $1
pushes $1 on the stack

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

takes one argument, no return value

enters scope of block $1
pushes $1 on the stack

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

← pops top value off the stack
if it is `0`, it pushes back `1`, else it pushes `0`

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

← pops top value off the stack
if it is `0`, it pushes back `1`, else it pushes `0`

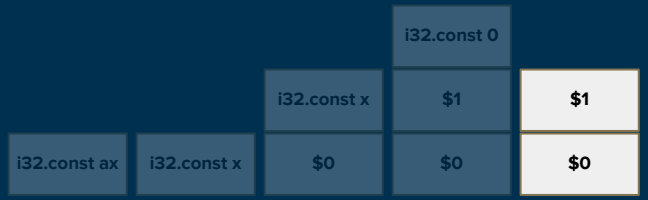
x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

← pops top value off the stack
if it is `0`, it pushes back `1`, else it pushes `0`



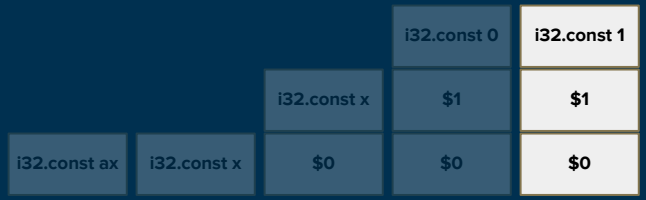
x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

← pops top value off the stack
if it is `0`, it pushes back `1`, else it pushes `0`



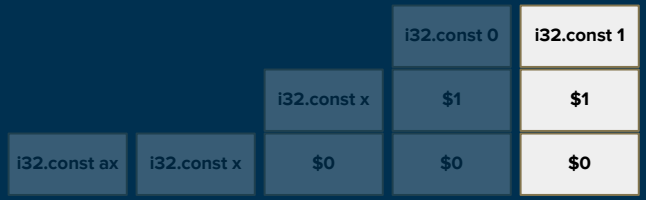
x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

← pops top value off the stack
if it is not `0`, it jumps out `0 + 1` blocks



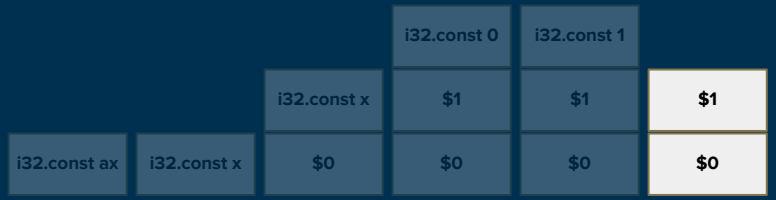
x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

pops top value off the stack
if it is not `0`, it jumps out `0 + 1` blocks



x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

pops top value off the stack
if it is not `0`, it jumps out `0 + 1` blocks

x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

pops top value off the stack
if it is not `0`, it jumps out `0 + 1` blocks

gives control here

x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

pushes value `0` on the stack

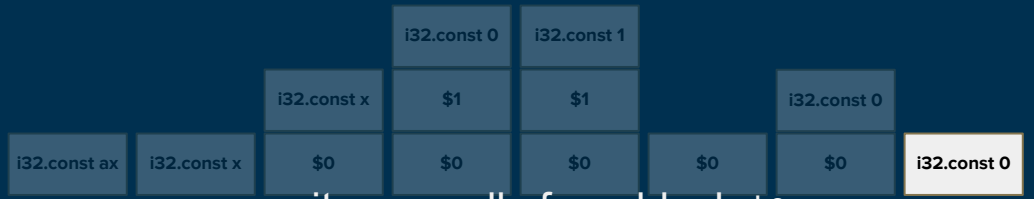
x = 0

Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```

returns one value



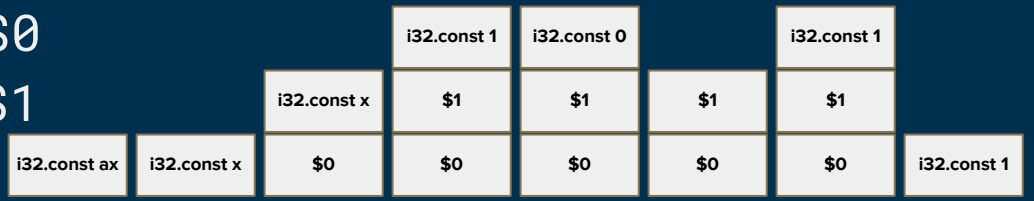
exits normally from block `$0`

x = 0

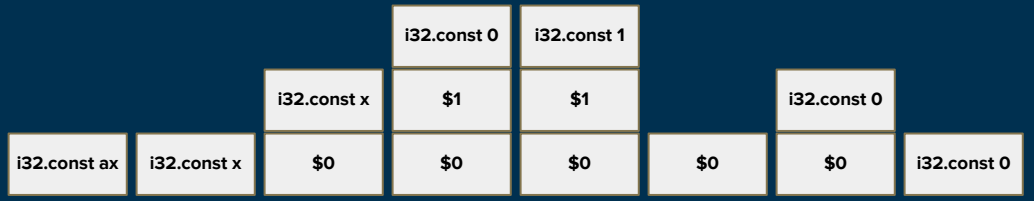
Example

`if (x) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```



x ≠ 0

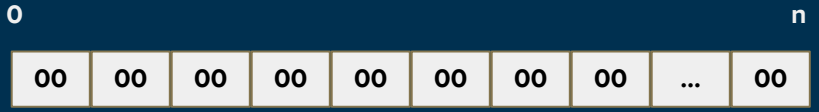


x = 0

Linear memory

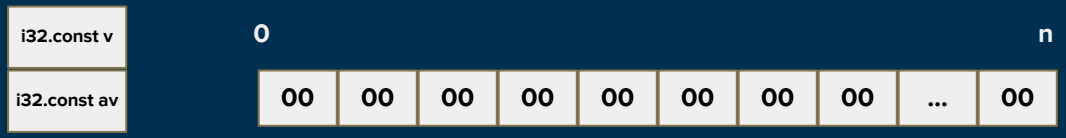
```
mem ::= t.load
      | t.store
      | memory.size
      | memory.grow
```

Linear memory



Linear memory

`i32.store`

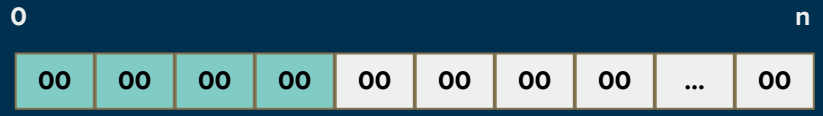


Linear memory

```
i32.const 0
i32.const 10752
i32.store
```

Linear memory

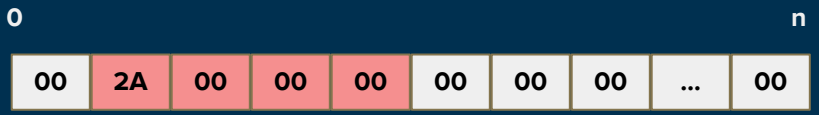
```
i32.const 0
i32.const 10752
i32.store
```



Linear memory

```
i32.const 0
i32.const 10752
i32.store
```

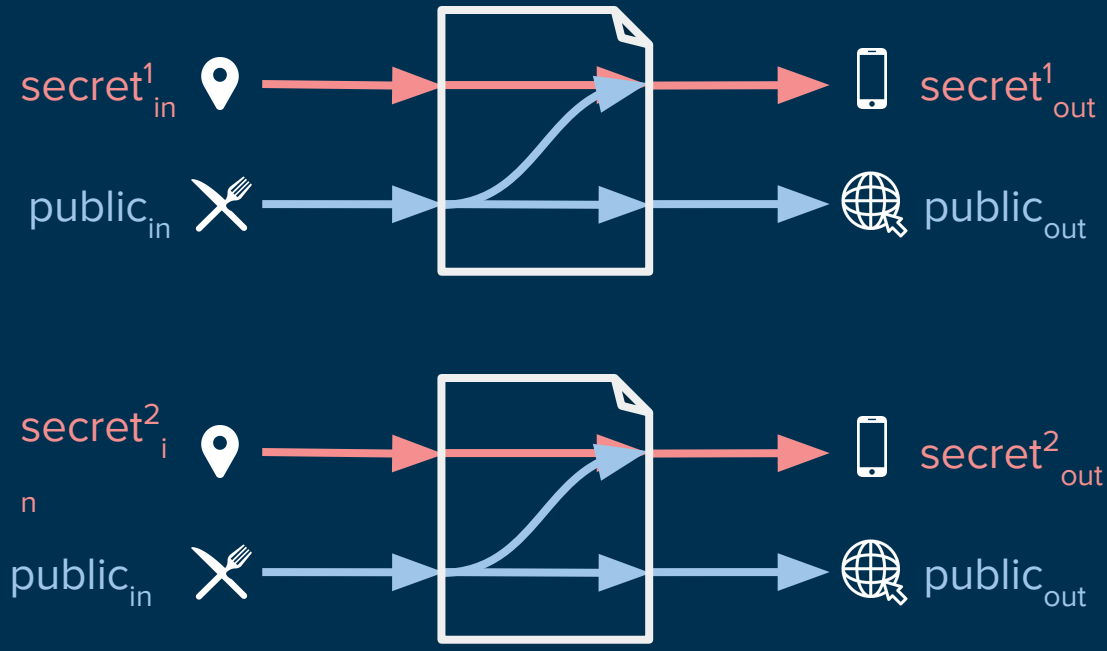

```
i32.const 1
i32.load  (42)
```




WA
+ IFC
?
SEC
WA

Recall

NI - property of traces



- **inputs/outputs**
- **memory locations**
- **…**

- **attacker view**

Attacker capabilities

- Observes information at $\ell \sqsubseteq \mathscr{A}$
- Executes Wasm programs
- Observes final state of global variables
- Does not observe the linear memory
- Does not observe the operand stack

Recall

Wasm

- Structured control flow
- Unwinding operand stack
- Unstructured linear memory
- Well-defined type system

$$C \vdash expr : t^n \rightarrow t^m$$

SecWasm

- Structured control flow
- Unwinding operand stack
- Unstructured **labeled** linear memory
- Well-defined **security** type system

$$\gamma, C \vdash expr \dashv \gamma'$$

SecWasm

- Structured control flow
- Unwinding operand stack
- Unstructured **labeled** linear memory
- Well-defined **security** type system

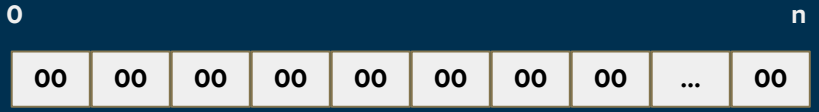
  $\gamma, C \vdash expr \dashv \gamma'$

- **Semantic security checks**

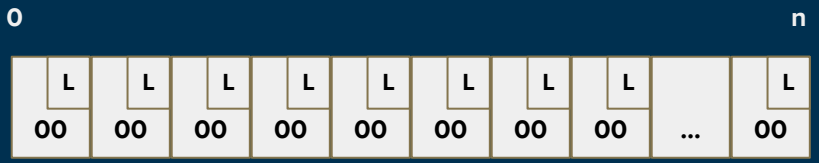
  $\langle\langle \sigma, S, expr \rangle\rangle \Downarrow \langle\langle \sigma', S', \theta \rangle\rangle$

Yes, big-step semantics!

Labeled linear memory

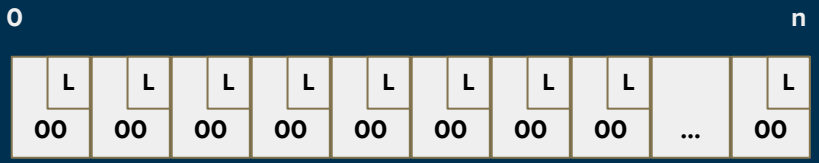


Labeled linear memory



Labeled linear memory

```
mem ::= t.load l
      | t.store l
      | memory.size
      | memory.grow
```



Labeled linear memory

$$
\begin{array}{rl}
\mathit{mem} ::= & t\texttt{.load}\ \ell \\
\mid & t\texttt{.store}\ \ell \\
\mid & \texttt{memory.size} \\
\mid & \texttt{memory.grow}
\end{array}
$$

- Dynamic checks for reads (**load**)

$\texttt{j = i+S.mem.offset}$ $\qquad j+|t|/8 \le \texttt{S.mem.data}$

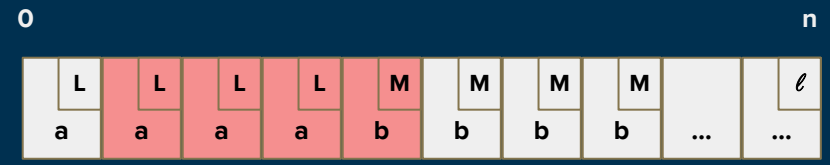
$\texttt{S.mem[j:j+}|t|/8\texttt{]} = (\texttt{b},\ell')^*$ $\qquad \texttt{bytes}_t\texttt{(n)} = \texttt{b}^*$ $\qquad \square\ell' \sqsubseteq \ell$

E-LOAD

$\langle\!\langle i32\texttt{.const i} :: \sigma, S, t\texttt{.load}\ \ell \rangle\!\rangle \Downarrow \langle\!\langle t\texttt{.const n} :: \sigma, S, \mathit{no\text{-}br} \rangle\!\rangle$

Labeled linear memory

```
i32.const 1
i32.load L
```



$L \sqsubseteq L \wedge M \not\sqsubseteq L$

- Dynamic checks for reads (**load**)

$j = i + S.mem.offset \qquad j + |t|/8 \le S.mem.data$

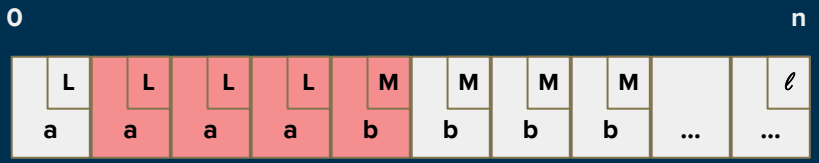
$S.mem[j:j+|t|/8] = (b, \ell')^* \qquad bytes_t(n) = b^* \qquad \square\ell' \sqsubseteq \ell$

E-LOAD

$\langle\langle i32.\mathbf{const}\ i :: \sigma, S, t.\mathbf{load}\ \ell \rangle\rangle \Downarrow \langle\langle t.\mathbf{const}\ n :: \sigma, S, no\text{-}br \rangle\rangle$

Labeled linear memory

```
i32.const 1
i32.load H
```



$L \sqsubseteq H \wedge M \sqsubseteq H$

- Dynamic checks for reads (**load**)

$j = i + S.mem.offset \qquad j + |t|/8 \le S.mem.data$

$S.mem[j:j+|t|/8] = (b, \ell')^* \qquad bytes_t(n) = b^* \qquad \square \ell' \sqsubseteq \ell$

E-LOAD

«*i32*.**const** i :: σ,S,*t*.**load** ℓ» ⇓ «*t*.**const** n :: σ,S,*no-br*»

Labeled linear memory

- Static checks for writes (**store**)

$$\frac{\texttt{C.mem} = \texttt{n} \qquad \texttt{pc} \sqcup \ell_a \sqcup \ell_v \sqsubseteq \ell}{\langle t\texttt{<}\ell_v\texttt{>}\texttt{::}t\texttt{<}\ell_a\texttt{>}\texttt{::st,pc}\rangle \texttt{ :: } \gamma\texttt{,C} \vdash t\texttt{.}\mathbf{store}\ \ell \dashv \langle \texttt{st,pc}\rangle \texttt{ :: } \gamma}\ \textbf{T-STORE}$$

Labeled linear memory

```
i32.const 2
i32.const c
i32.store H
```

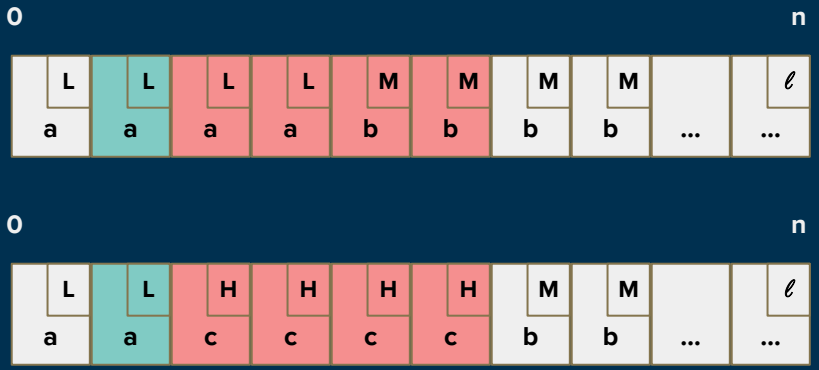
$\mathsf{pc} \sqcup \mathsf{L} \sqcup \mathsf{M} \sqcup \ell_c \sqsubseteq \mathsf{H}$

- Static checks for writes (**store**)

$$\frac{\mathtt{C.mem} = \mathtt{n} \qquad \mathsf{pc} \sqcup \ell_a \sqcup \ell_v \sqsubseteq \ell}{\langle t\langle \ell_v \rangle :: t\langle \ell_a \rangle :: \mathtt{st}, \mathtt{pc} \rangle :: \gamma, \mathtt{C} \vdash t.\mathbf{store}\ \ell \dashv \langle \mathtt{st}, \mathtt{pc} \rangle :: \gamma}\ \textbf{T-STORE}$$

Labeled linear memory

```
i32.const 2
i32.const c
i32.store H
```

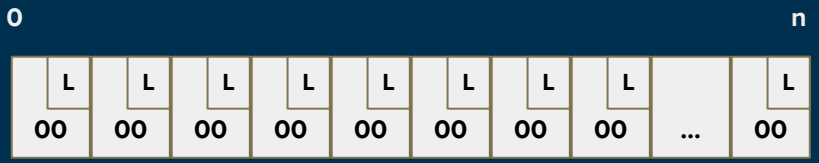


- Flow-sensitive memory labeling

$$\frac{\texttt{j = i+S.mem.offset} \qquad \texttt{j+}|t|/8 \le \texttt{S.mem.data} \qquad \texttt{bytes}_t\texttt{(n) = b*} \qquad \texttt{S' = S.mem[j:j+}|t|/8 \mapsto \texttt{(b,}\ell\texttt{)*]}}{\texttt{«}i32\texttt{.const n :: }i32\texttt{.const i :: }\sigma\texttt{,S,}t\texttt{.store }\ell\texttt{»} \Downarrow \texttt{«}\sigma\texttt{,S',}no\text{-}br\texttt{»}}\ \textbf{E-STORE}$$

Labeled linear memory

```
mem ::= t.load l
      | t.store l
      | memory.size
      | memory.grow
```



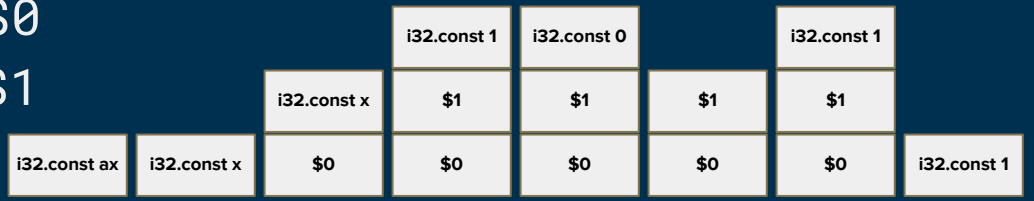
- Dynamic checks for reads (**load**)
- Static checks for writes (**store**)
- Flow-sensitive memory labeling

Recall

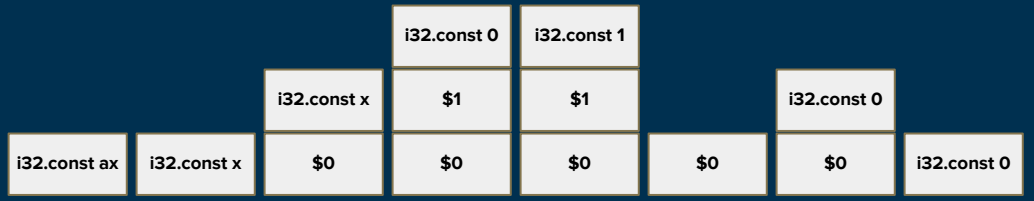
`if (x) { return 0; } else { return 1; }`

Example

```
1  i32.const ax
2  i32.load
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```



x ≠ 0

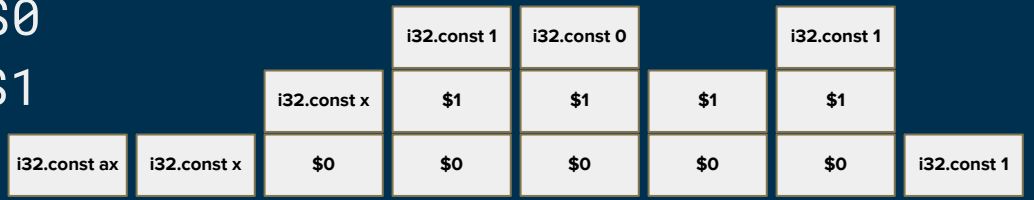


x = 0

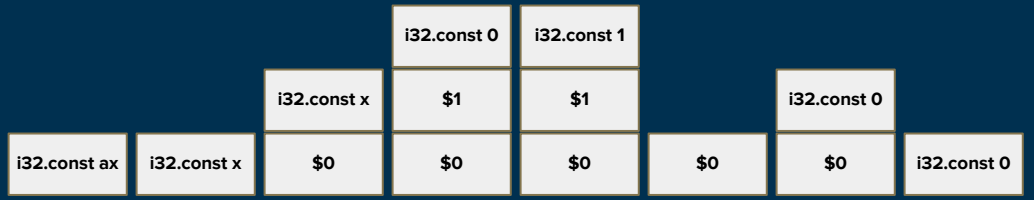
Example

`if (x`[H]`) { return 0; } else { return 1; }`

```
1  i32.const ax
2  i32.load H
3  block (i32 → i32) $0
4    block (i32 → ε) $1
5      i32.eqz
6      br_if 0
7      i32.const 1
8      br 1
9    end
10   i32.const 0
11 end
```



x ≠ 0



x = 0

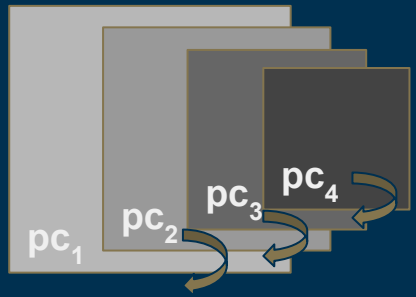
Tracking flows

- stack of security labels *st*

  one for every element on the operand stack
- well-formedness: *st* ⊢ σ

Explicit flows

- stack of pcs, one for every block

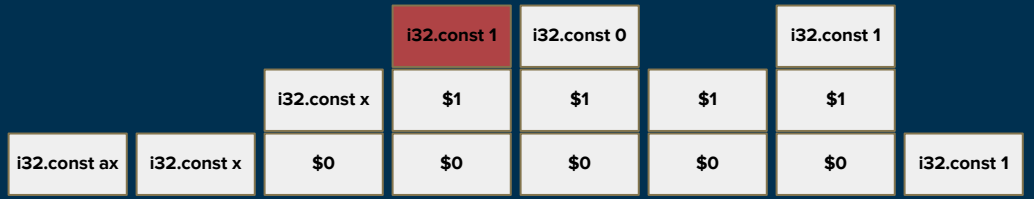


- combined in `γ ::= ⟨st,pc⟩ :: γ'`

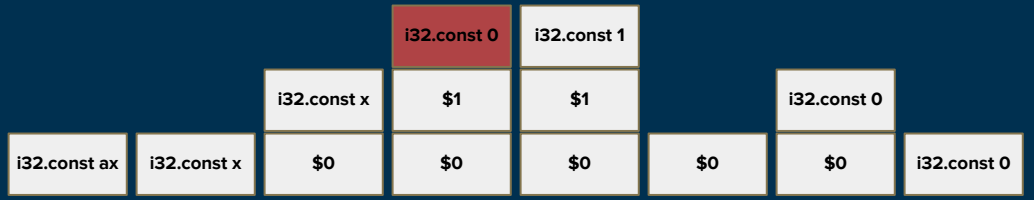
Implicit flows

Tracking implicit flows

```
1  i32.const ax
2  i32.load H
3  block (i32 ⟶ i32) $0
4    block (i32 ⟶ ε) $1
5        i32.eqz
6        br_if 0
7        i32.const 1
8        br 1
9      end
10     i32.const 0
11 end
```



x ≠ 0



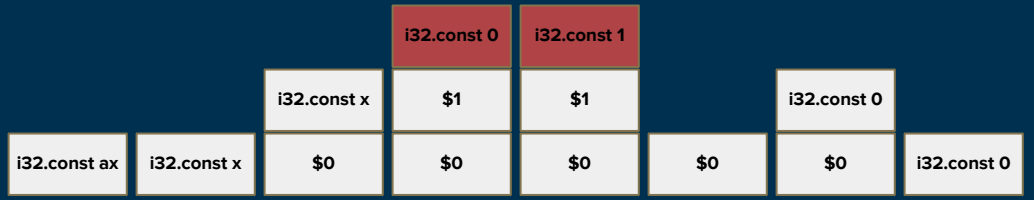
x = 0

```
if (xH) { return 0; } else { return 1; }
```

Tracking implicit flows

```
1  i32.const ax
2  i32.load H
3  block (i32 ⟶ i32) $0
4    block (i32 ⟶ ε) $1
5        i32.eqz
6        br_if 0
7        i32.const 1
8        br 1
9      end
10     i32.const 0
11 end
```

x ≠ 0



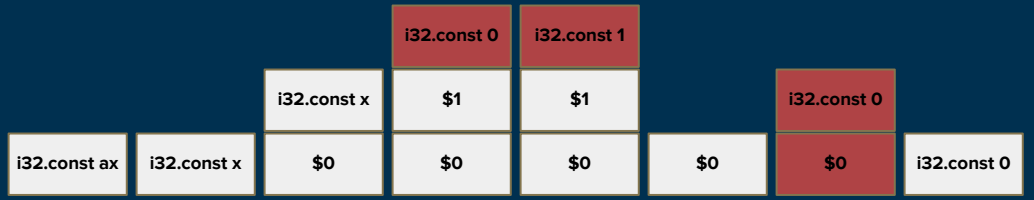
x = 0

```
if (xH) { return 0; } else { return 1; }
```

Tracking implicit flows

```
1  i32.const ax
2  i32.load H
3  block (i32 ⟶ i32) $0
4    block (i32 ⟶ ε) $1
5        i32.eqz
6        br_if 0
7        i32.const 1
8        br 1
9     end
10    i32.const 0
11 end
```

x ≠ 0



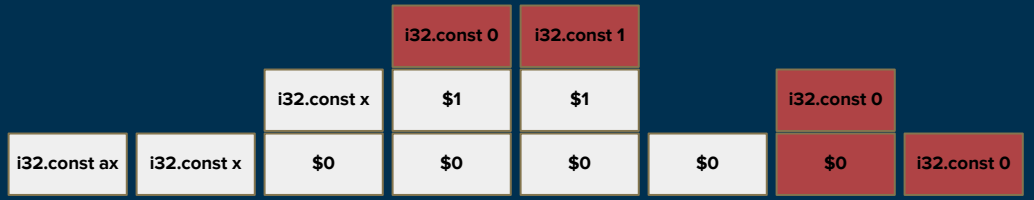
x = 0

```
if (xH) { return 0; } else { return 1; }
```

Tracking implicit flows

```
1  i32.const ax
2  i32.load H
3  block (i32 ⟶ i32) $0
4    block (i32 ⟶ ε) $1
5        i32.eqz
6        br_if 0
7        i32.const 1
8        br 1
9     end
10    i32.const 0
11 end
```

x ≠ 0



x = 0

```
if (xH) { return 0; } else { return 1; }
```

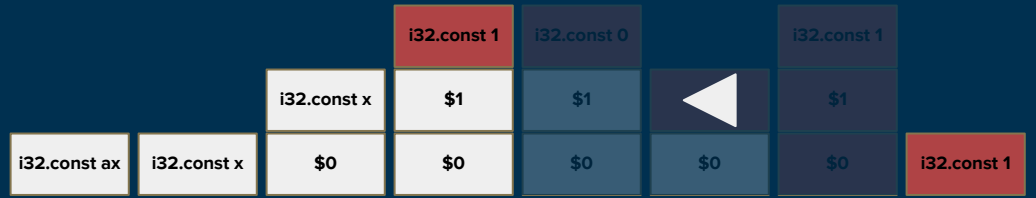
Recall

Attacker model

- Observes information at $\ell \sqsubseteq \mathscr{A}$
- Executes Wasm programs
- Observes final state of global variables
- Does not observe the linear memory
- Does not observe the operand stack

Confinement and Noninterference

```
1  i32.const ax
2  i32.load H
3  block (i32 ⟶ i32) $0
4    block (i32 ⟶ ε) $1
5        i32.eqz
6        br_if 0
7        i32.const 1
8        br 1
9     end
10    i32.const 0
11 end
```



x ≠ 0

x = 0

```
if (xH) { return 0; } else { return 1; }
```

Confinement and Noninterference

Lemma 1 (Confinement). *For any typing context C, store S_0, operand stack σ_0, stack-of-stacks γ_0, and expression expr, such that $C \vdash S_0$, $C \vdash \sigma_0$, and $\gamma_0 \Vdash \sigma_0$, if $\langle\!\langle \sigma_0, S_0, expr \rangle\!\rangle \Downarrow \langle\!\langle \sigma_1, S_1, \theta \rangle\!\rangle$, $\langle st_0, pc \rangle :: \gamma_0, C \vdash expr \dashv \gamma_1$, and $\gamma[0].\mathsf{snd} \notin \mathcal{A}$, then the following statements hold:*

1) $\gamma_0 \Vdash \sigma_0 \blacktriangleleft^C_{\mathcal{A}} \Delta(C, \gamma_1, \theta) \Vdash \sigma_1$,
2) $S_0 \blacktriangleleft^C_{\mathcal{A}} S_1$, *and*
3) $\gamma_1[0 : \mathsf{nat}(\mathsf{pred}(\theta))].\mathsf{snd} \notin \mathcal{A}$.

Theorem 1 (Noninterference). *If*

1) $\gamma, C \vdash expr \dashv \gamma'$,
2) $C \vdash S_0$ *and* $C \vdash S_1$,
3) $C \vdash \sigma_0$ *and* $C \vdash \sigma_1$,
4) $\gamma \Vdash \sigma_0 \sim^C_{\mathcal{A}} \gamma \Vdash \sigma_1$,
5) $\langle\!\langle \sigma_0, S_0, expr \rangle\!\rangle \Downarrow \langle\!\langle \sigma'_0, S'_0, \theta_0 \rangle\!\rangle$ *and* $\langle\!\langle \sigma_1, S_1, expr \rangle\!\rangle \Downarrow \langle\!\langle \sigma'_1, S'_1, \theta_1 \rangle\!\rangle$, *and*
6) $S_0 \sim^C_{\mathcal{A}} S_1$,

then $S'_0 \sim^C_{\mathcal{A}} S'_1$ *and* $WS_{\gamma', C}(\langle \sigma'_0, \theta_0 \rangle, \langle \sigma'_1, \theta_1 \rangle)$.

Conclusion

- SecWasm: hybrid IFC enforcement for Wasm 
- Fine-grained flow-sensitive memory labeling
- Security type system
- Some dynamic checks
- Termination-insensitive noninterference