Non-Interference for a Typed Assembly Language

Ricardo Medel[1]

Joint work with **Adriana Compagnoni[1] and Eduardo Bonelli[2]**

1. Dept. of Computer Science, Stevens Institute of Technology, Hoboken, NJ.

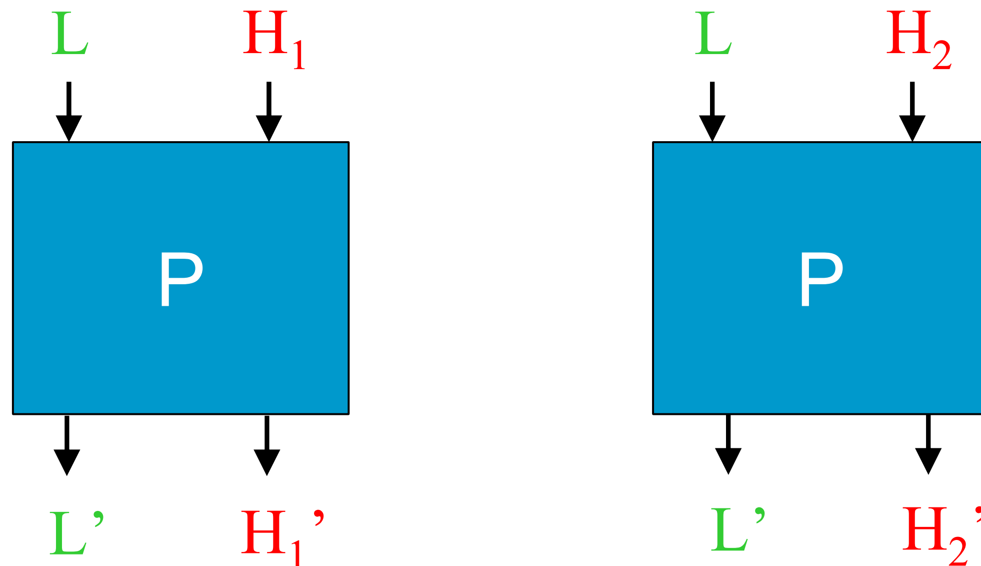
2. LIFIA, Universidad Nacional de La Plata, Buenos Aires (Argentina)

Motivations: Confidentiality

- Secret data should not flow to public channels
- Access control mechanisms do not control the propagation of data
- Enforce confidentiality policies on an end-to-end basis

Motivations: Non-interference

- Low security output should not be affected by high security input [Goguen & Meseguer 82]



Motivations: Assembly Language

High-level source code

Compilation

Assembly code (or other exec. code)

Information flow analysis (non-interference) → Ok

Type preservation

Information flow analysis (non-interference) → ?

Formalizing Confidentiality Policies

- **Indicate which information is public, secret, etc.:** *decorate computational objects with security labels {L,H}*
- **Specify information flow policy:** *information may flow from low security to high security locations*

Information Flow Policy

- Information flow policy as lattice on security labels [Bell & LaPadula 73, Denning 76]:

$l_1 \leq l_2 \Rightarrow$ ***information can flow from level l_1 to level l_2***

$$L \leq H$$

Easy to express merging/splitting requirements.

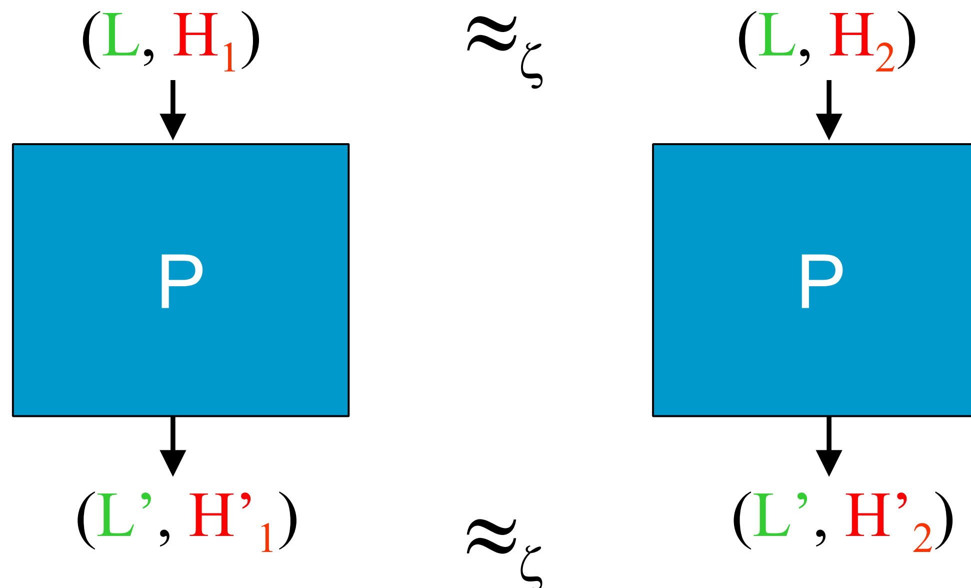
$$\mathtt{x}^{lx}\mathtt{:=y}^{ly}\mathtt{+z}^{lz}\mathtt{;} \qquad ly \cup lz \leq lx$$

Non-interference

- ζ-indistinguishability

$$v^{l_1} \approx_\zeta w^{l_2} \quad \text{iff} \quad l_1, l_2 \leq \zeta \Rightarrow v = w$$

- ζ-indist. input produces ζ-indist. output



Examples of Illegal Flows

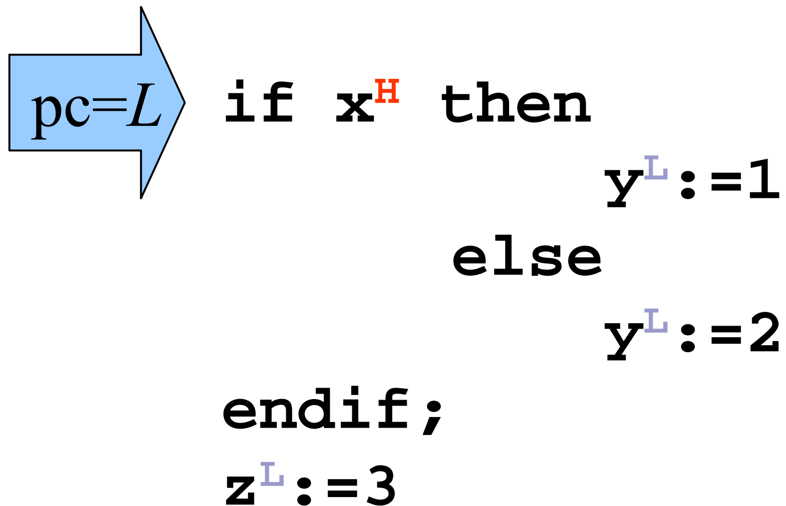
```
y^L:=x^H
```

explicit

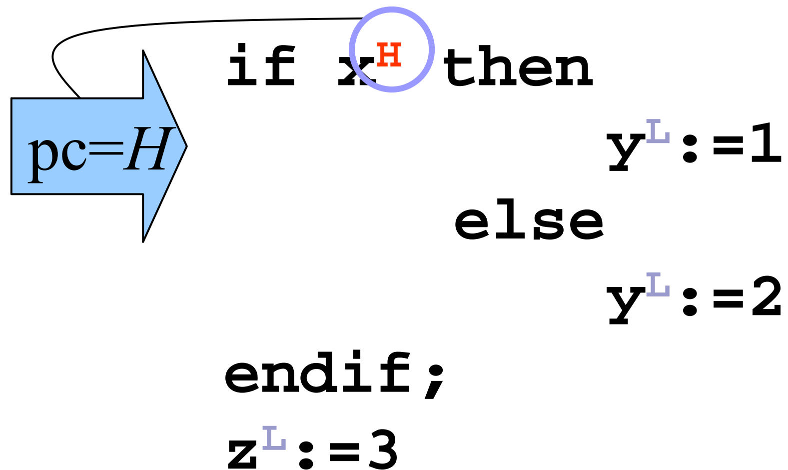
```
if x^H then
        y^L:=1
    else
        y^L:=2
endif
```

implicit

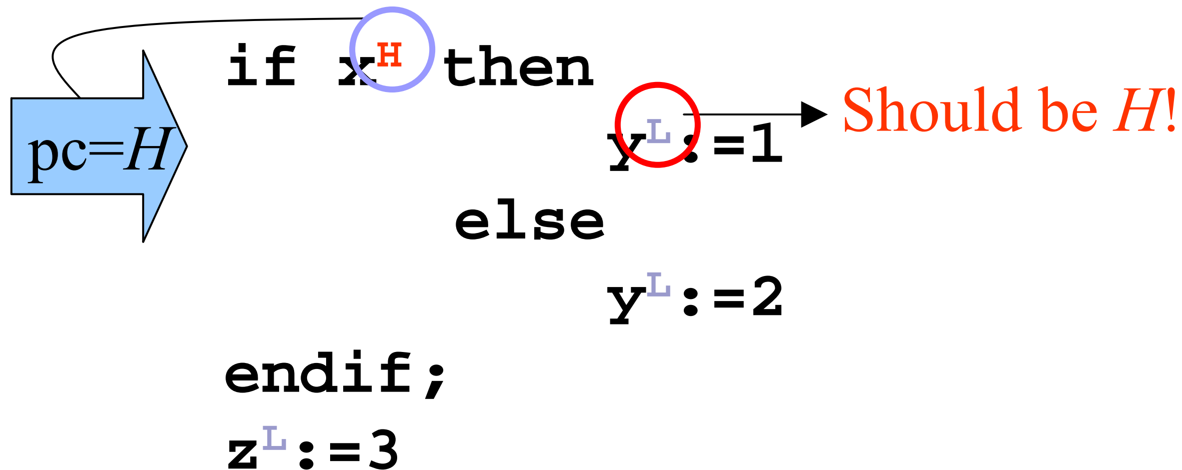
Detection of Implicit Illegal Flow



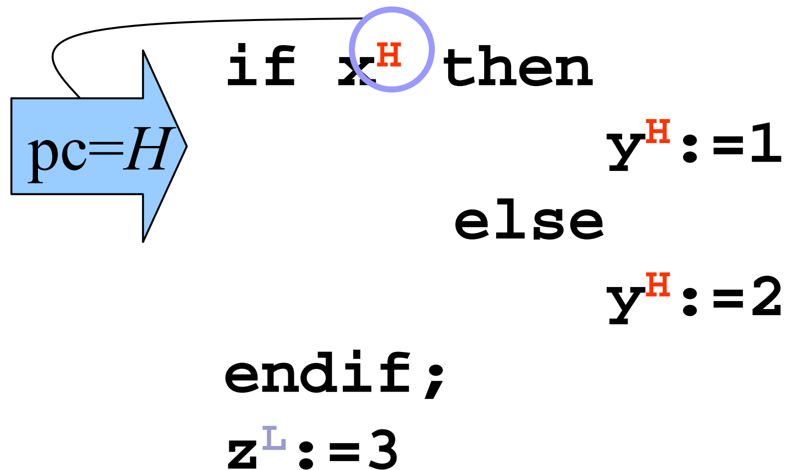
Detection of Implicit Illegal Flow



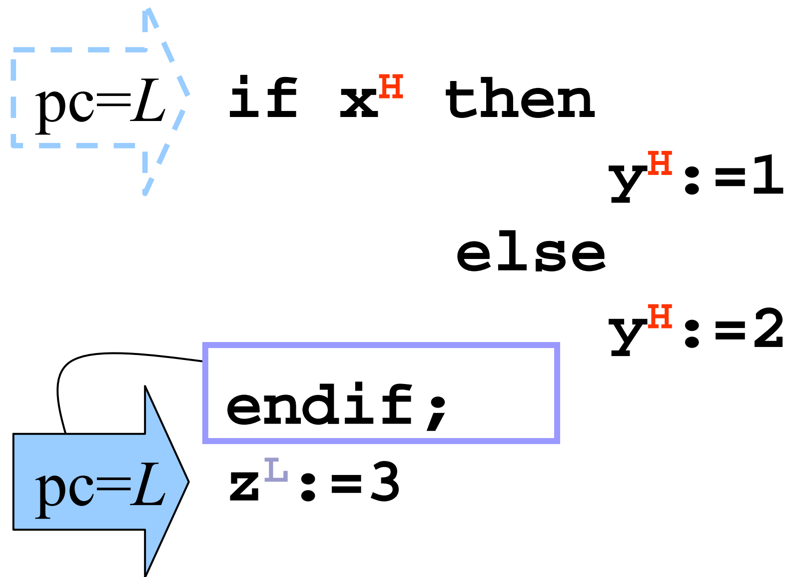
Detection of Implicit Illegal Flow



Detection of Implicit Illegal Flow



Detection of Implicit Illegal Flow



Restore the level **pc** had before entering **`if-then-else`**

Difficulties with Assembly Language

- High-level control flow constructs not available
  - **Simulate the block structure in low-level programs**
  - **Code labels represent the junction points**
  - **A stack of code labels represents the nested block structures**

Example revisited

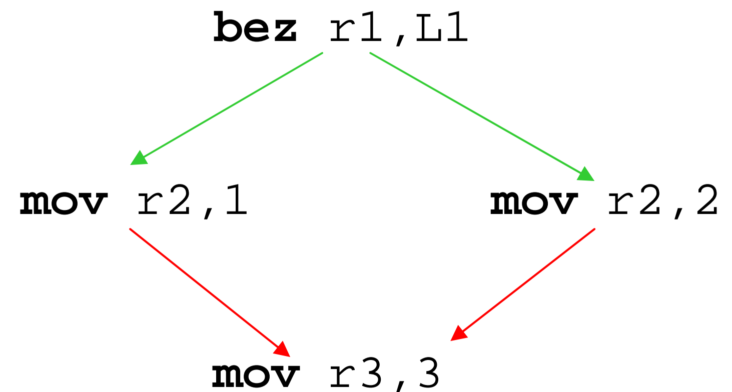
- Standard Translation into Assembly Language

```
LSTART: bez r1,L1;        % if x
        mov r2,1;         % then y:=1
        jmp L2;
    L1: mov r2,2;         % else y:=2
    L2: mov r3,3;         % z:=3
        halt
```

Example revisited (cont'd)

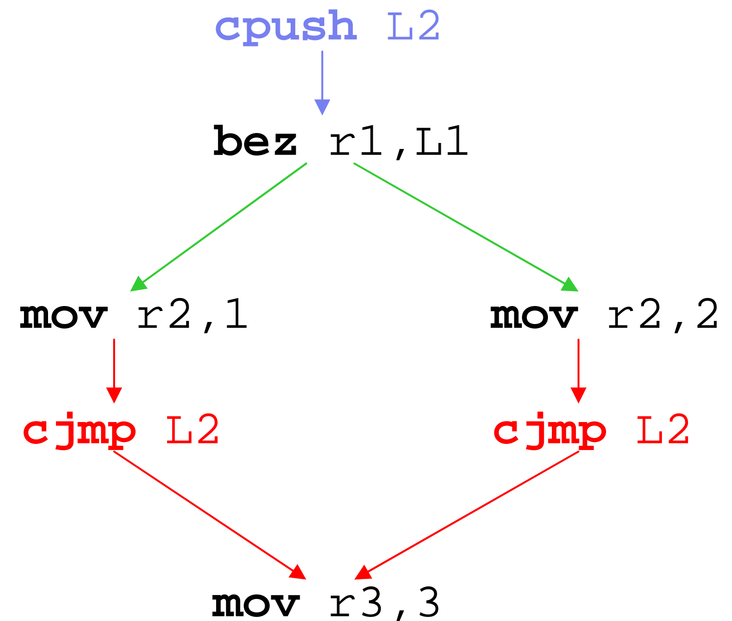
- Standard Translation into Assembly Language

```
L_START:  bez r1,L1;
          mov r2,1;
          jmp L2;
     L1:  mov r2,2;
     L2:  mov r3,3;
          halt
```



SIF Version of Example

```
L_START: cpush L2;
         bez r1,L1;
         mov r2,1;
         cjmp L2;
L1:      mov r2,2;
         cjmp L2;
L2:      mov r3,3;
         halt
```



Syntax of SIF

$$Program\ \ P ::= \texttt{eof} \mid L\texttt{:}P \mid i\texttt{;}P$$

$$Instructions\ \ i ::= \texttt{halt} \mid \texttt{jmp}\ L \mid \texttt{bnz}\ \ r, L \mid \texttt{bez}\ \ r, L$$

$$\mid \texttt{arith}\ r \leftarrow r \otimes r \mid \texttt{arithi}\ r \leftarrow r \otimes n$$

$$\mid \texttt{load}\ r_d \leftarrow r_s[c] \mid \texttt{store}\ r_d[n] \leftarrow r_s$$

$$\mid \texttt{cpush}\ L \mid \texttt{cjmp}\ \ L$$

Types in SIF

Security types	$\sigma ::= \omega^{l}$
Word types	$\omega ::= \mathsf{int} \mid [\tau]$
Heap location types	$\tau ::= \sigma_1 \times \ldots \times \sigma_n \mid \mathsf{code}$

Registers Context	$\Gamma = \{r_0{:}\sigma_0, \ldots, r_n{:}\sigma_n, \mathrm{pc}{:}[\mathsf{code}]^{l}\}$
Junction Points Stack	$\Lambda ::= \varepsilon \mid L \cdot \Lambda$

Types in SIF (cont'd)

Contexts $\Gamma \mid \Lambda$

Signature Σ : *Code labels* $\rightarrow$ *Contexts*

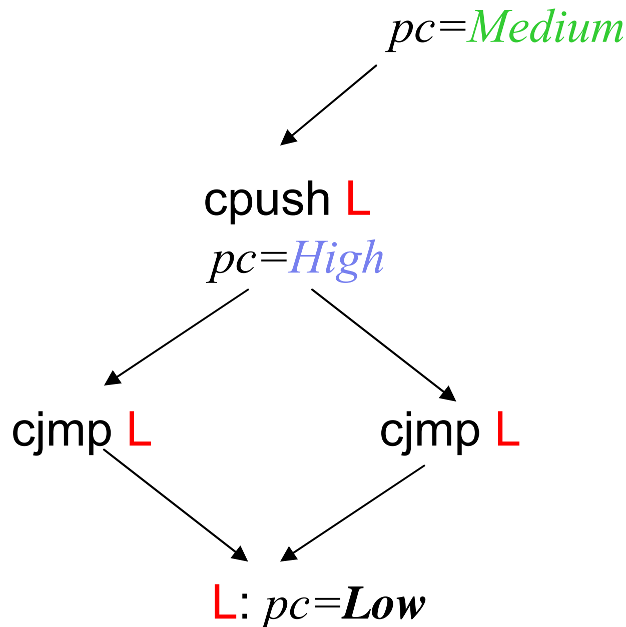
$\mathtt{L}_{\mathtt{START}}$: $\{r_1{:}int^H, r_2{:}int^H, \mathbf{pc}{:}\ L\} \mid \varepsilon$

```
        cpush L2;
        bez r1,L1;
        mov r2,1
```

A program P is well-typed if $\Sigma(\mathtt{L}_{\mathtt{start}}) \vdash_{\Sigma} P$

Sample typing rules: cpush

$$\frac{l \leq \Sigma(L)\,(\mathrm{pc}) \qquad \Gamma, \mathrm{pc}{:}\, l \mid L \cdot \Lambda \vdash_{\Sigma} P}{\Gamma, \mathrm{pc}{:}\, l \mid \Lambda \vdash_{\Sigma} \texttt{cpush}\ L\,\texttt{;}\ P}$$




```
if z^M=0 then
         if x^H=0 then
                   …
                   else
                   …
         endif
pc=Low →  y^L:=1
endif
```

Sample typing rules: `cjmp`

$$\frac{\Sigma(L) = \Gamma' \mid \Lambda \qquad \Gamma'_{/pc} \subseteq \Gamma_{/pc} \qquad \mathbf{Ctxt}(P) \vdash_\Sigma P}{\Gamma \mid L \cdot \Lambda \vdash_\Sigma \texttt{cjmp}\ L\texttt{;}P}$$

$\mathbf{Ctxt}(L\texttt{;}P) = \Sigma(L)$

$\mathbf{Ctxt}(\texttt{eof}) = \{\} \mid \varepsilon$

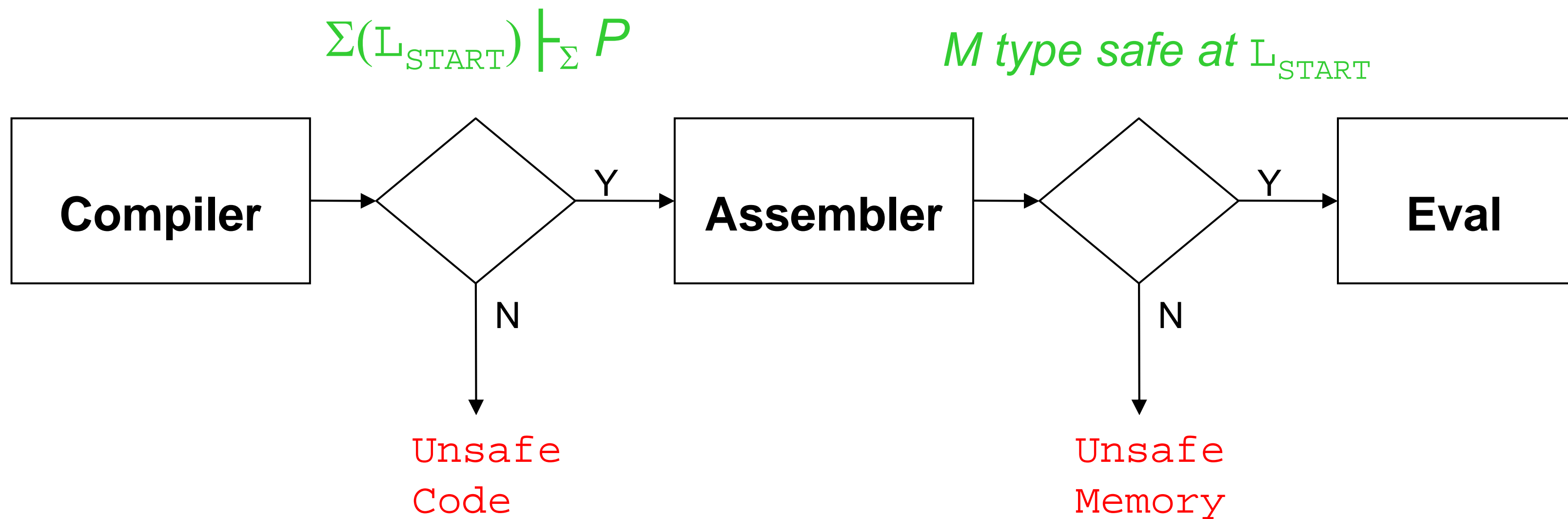
Machine Configurations

Machine configuration: $M = (H, R)$

M is type safe at u in a well-typed program $P = p_1; \ldots p_u; \ldots; p_n$ with heap type Ψ if M satisfies the typing assumptions Γ_u for the instruction p_u

A well-typed program P will be non-interferent if executed in a type-safe machine configuration

Verification Schema



- No typed heap required at type-checking

Non-Interference

ζ-indistinguishability of machine configurations:

$$M_1 : \Gamma_1, \Lambda_1, \Psi_1 \;\approx_{P,\zeta}\; M_2 : \Gamma_2, \Lambda_2, \Psi_2$$

Non-Interference Theorem:

$$M_1 : \Gamma_{start}, \Lambda_{start}, \Psi \;\approx_{P,\zeta}\; M_2 : \Gamma_{start}, \Lambda_{start}, \Psi$$

$$\downarrow^* \qquad\qquad \downarrow^*$$

$$M'_1 : \Gamma_v, \Lambda_v, \Psi_v \;\approx_{P,\zeta}\; M'_2 : \Gamma_w, \Lambda_w, \Psi_w$$

with M_1, M_2 *type safe at* L_{START}*, and* M'_1, M'_2 *in final state.*

Future Work

- Include an **execution stack** in SIF.
- **Reuse** of registers.
- **Compilation** function from a imperative language, and proof of type preservation.
- **Complexity** of typechecking.

Related work

- High Level Languages
  - Smith, Volpano & Irvine [SVI96, SV98]
  - Myers [ML97, My99]
  - Heintze & Riecke [HR98, ABHR99]
  - Sabelfeld & Sands [SS99, SS00]
  - Pottier & Conchon [PC00]
  - Banerjee & Naumann [BN05]
- Low Level Languages
  - Myers & Zdancewic [ZM01,ZM02]
  - Barthe et al [BBR04]
  - Crary et al [CKP05]

Thank you!

Non-Interference (formal)

Non-Interference Theorem:

Given a well-typed program $P=p_1;\ldots p_u;\ldots;p_n$ and machines M_1, M_2 type safe at 1 in P with Ψ and

$$M_1{:}\Gamma_1, \Lambda_1, \Psi \approx_{P,\zeta} M_2{:}\Gamma_1, \Lambda_1, \Psi$$

If $M_1 \rightarrow^* M'_1$ and $M_2 \rightarrow^* M'_2$, with p_v (resp. p_w) the current instruction in M'_1 (resp. M'_2), and with both M'_1, M'_2 in final state, then

$$M'_1{:}\Gamma_v, \Lambda_v, \Psi_v \approx_{P,\zeta} M'_2{:}\Gamma_w, \Lambda_w, \Psi_w$$

Other channels

- Timing (including Non-Termination)
- Resource Exhaustion
- Power consumption

Other proposed solution

- A list of program points where is safe to lower the ***pc*** replaces the signature Σ
  - Depends on the trustworthiness of the list
  - Our type rules verify the well-formedness of Σ

Other proposed solution (cont'd)

Suppose that **pc**=*medium* at position 1.

```
1.     L:   bez r1,L1;
2.          mov r2,1;
3.          jmp L2;
4.    L1:   mov r2,2;
5.    L2:   mov r3,3;
6.          halt
```

List = ⟨(5,low)⟩

The **pc** is lowered beyond the original level. Medium-security information flows to a low-security register (`r3`).

Other proposed solution (cont'd)

```
L:      cpush L2;
        bez r1,L1;
        mov r2,1;
        cjmp L2;
L1:     mov r2,2;
        cjmp L2;
L2:     mov r3,3;
        halt
```

$$\frac{\textit{medium} \leq \Sigma(L2)\ (\mathrm{pc}) \qquad \Gamma,\ \mathrm{pc}{:}\textit{medium} \mid L2 \cdot \Lambda \vdash_{\Sigma} P}{\Gamma,\ \mathrm{pc}{:}\textit{medium} \mid \Lambda \vdash_{\Sigma} \mathbf{cpush}\ \ L2\,\mathbf{;}\ P}$$