A type system for complexity flow analysis

Jean-Yves Marion
LICS, Juin 2011

## Summary

1. Complexity and Information flow
(a) Information flow induced by predicative recurrence (ICC)
(b) Types for secure flow analysis
2. A type system for complexity analysis of while-programs
3. A characterization of Ptime by While programs

A while programming language

$$
E \in \text { Expressions }::=X|d| o p\left(E_{1}, \ldots, E_{n}\right)
$$

$C, C^{\prime} \in$ Commands $::=X:=E\left|C ; C^{\prime}\right|$ while $(E)\{C\}$
| if $E$ then $C$ else $C^{\prime}$

A while programming language

$$
E \in \text { Expressions }::=X|d| o p\left(E_{1}, \ldots, E_{n}\right)
$$

$C, C^{\prime} \in$ Commands $::=X:=E\left|C ; C^{\prime}\right|$ while $(E)\{C\}$ if $E$ then $C$ else $C^{\prime}$

How to define a type system to control the computational complexity?

## Related results on compexity and imperative programming langagues

- Matrices flow calculus (Ben-Amram , Jones, Kristiansen, Moyen, Niggl, Wunderlich)
- Sup-interpretation (Marion, Péchoux) and OO-programming style
- Applied Linear logic (Hofmann \& al) applied to multi-threading (Amadio, Madet)
- Symbolic Ressource analysis
- Java bytecode (Albert \& al)
- Speed (Gulwani \& al)
- See also WCET analysis community


## Main result

1. A type system for an imperative programming language such that
2. Terminating and typed programs are computable in polynomial time
3. Conversely, each polynomial time function is definable by a typed program.

## Main result

1. A type system for an imperative programming language such that
2. Terminating and typed programs are computable in polynomial time
3. Conversely, each polynomial time function is definable by a typed program.

Two rationales
$\Rightarrow$ Information flow induced by ramified recursion
$\Rightarrow$ Type systems for secure information flow analysis

## Ramified recursion and complexity

The set of functions defined by ramified primitive recursion is exactly the set of polynomial time functions (PTIME).

Bellantoni \& Cook and Leivant (1992)

## Ramified recursion and complexity

## Ramified recursion and complexity

$\Rightarrow$ Tiers
$\underset{\sim}{\mathbb{N}}(k)$
$\uparrow{ }_{\substack{2}}^{\mathbb{N}}(1)$
$\mathbb{N}(0)$

## Ramified recursion and complexity

$\begin{array}{ccr}=\text { Tiers } & \mathbb{N}(k) & g: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \\ & \uparrow & h: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \rightarrow \mathbb{N}(0) \\ & \mathbb{N}(1) & \\ & \uparrow{ }^{\mathbb{N}}(0) & \end{array}$

Ramified recursion and complexity
$\Rightarrow$ Tiers


$$
\begin{aligned}
& g: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \\
& h: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \rightarrow \mathbb{N}(0)
\end{aligned}
$$

$\Rightarrow$ Primitive recursion

$$
\begin{aligned}
f(0, y) & =g(y) \\
f(x+1, y) & =h(x, f(x, y))
\end{aligned}
$$

## Ramified recursion and complexity

$\begin{array}{cc}- \text { Tiers } & \mathbb{N}(k) \\ & \uparrow \\ & \mathbb{N}(1) \\ & \uparrow \\ & \mathbb{N}(0)\end{array}$
$f: \mathbb{N}(1) \rightarrow \mathbb{N}(k) \rightarrow \mathbb{N}(0)$

$$
\begin{array}{r}
g: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \\
h: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \rightarrow \mathbb{N}(0)
\end{array}
$$

$\Rightarrow$ Primitive recursion

$$
\begin{aligned}
f(0, y) & =g(y) \\
f(x+1, y) & =h(x, f(x, y))
\end{aligned}
$$

Ramification

## Ramified recursion and complexity

$\begin{array}{cc}- \text { Tiers } & \mathbb{N}(k) \\ & \uparrow \\ & \mathbb{N}(1) \\ & \uparrow \\ & \mathbb{N}(0)\end{array}$
$f: \mathbb{N}(1) \rightarrow \mathbb{N}(k) \rightarrow \mathbb{N}(0)$

$g: \mathbb{N}(k) \rightarrow \mathbb{N}(0)$
$h: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \rightarrow \mathbb{N}(0)$
$\Rightarrow$ Primitive recursion

$$
\begin{aligned}
f(0, y) & =g(y) \\
f(x+1, y) & =h(x, f(x, y))
\end{aligned}
$$

Ramification

- Downward flow from 1 to $\mathbf{0}$


## Ramified recursion and complexity

$\begin{array}{cc}- \text { Tiers } & \mathbb{N}(k) \\ & \uparrow \\ & \mathbb{N}(1) \\ & \uparrow \\ & \mathbb{N}(0)\end{array}$
$f: \mathbb{N}(1) \rightarrow \mathbb{N}(k) \rightarrow \mathbb{N}(0)$

$\Rightarrow$ Downward flow from 1 to $\mathbf{0}$
$g: \mathbb{N}(k) \rightarrow \mathbb{N}(0)$
$h: \mathbb{N}(k) \rightarrow \mathbb{N}(0) \rightarrow \mathbb{N}(0)$
$\Rightarrow$ Primitive recursion

$$
\begin{aligned}
& f(0, y)=g(y) \\
& f(x+1, y)=h(x, f(x, y)) \\
& \text { Ramification }
\end{aligned}
$$

.
$\Rightarrow$ But No upward flow from 0 to 1

## Ramified recursion and complexity

The set of functions defined by ramified primitive recursion is exactly the set of polynomial time functions. Bellantoni \& Cook and Leivant

Ramified recursion enforces a restriction on data flow


Downward data flow

## A few sources

- Set theory (Russel)
- On (non)-constructions of too fast (to be real) functions (Nelson, Simmons)
- In second order logic with restriction on comprehension axiom (Leivant)
- Ramified systems (Bellantoni-Cook, Leivant, Marion,...)
- Light linear logics (Girard, Lafont, Baillot...)
- Typed lambda-calculus (Hofmann, Baillot, Dal Lago, Ronchi Della Rocca, ...)

A data flow analysis from an ICC point of view

A data flow analysis from an ICC point of view
Implicit flow from $x$ to $y$

```
int copy(int x, int y)
{
y=0;
while (x)
        {
        x:=x-1;
        y:=y+1;
    }
return y;
}
```

A data flow analysis from an ICC point of view
Implicit flow from $x$ to $y$
Types/tier
$\Gamma(x)$
$\Gamma(y)$
return $y$;
\}

A data flow analysis from an ICC point of view
Implicit flow from x to y

## Types/tier

$$
\Gamma(x)=\tau
$$

\{

$$
y=0 ;
$$

$$
\begin{aligned}
& \text { while } \quad(x) \\
& \left\{\begin{array}{l}
x:=x-1 \\
y:=y+1 \\
\}
\end{array}\right.
\end{aligned}
$$

return y ;
\}

A data flow analysis from an ICC point of view
Implicit flow from x to y

## Types/tier

$$
\Gamma(x)=\tau
$$

$$
\text { while }(x)
$$

$$
\Gamma(y)=\rho
$$



This is the intuition...

## Integrity security policy

- Security lattice

1 O Confidential
$0 \bigcirc$ Public

Types are security levels: $\Gamma(x), \Gamma(y)$

## Integrity security policy

- Security lattice

1 O Confidential
Types are security levels:

$$
\Gamma(x), \Gamma(y)
$$

$0 \bigcirc$ Public

- Biba's integrity policy (1977) :

Write down

$$
x \text { can write } y \text { if } \Gamma(y) \leq \Gamma(x)
$$

Read up

$$
x \text { can read } y \text { if } \Gamma(x) \leq \Gamma(y)
$$

A sound type system for information flow analysis

A security type system enforces an information flow policy (integrity/confidentialitiy)

- Volpano Smith and Irvine (1997)
- Survey of Sabelfeld and Myers

A sound type system for information flow analysis

A security type system enforces an information flow policy (integrity/confidentialitiy)

- Volpano Smith and Irvine (1997)
- Survey of Sabelfeld and Myers
- Type systems garantee non-interference:

Low level values does not alter high level values (integrity)

A sound type system for information flow analysis

A security type system enforces an information flow policy (integrity/confidentialitiy)

- Volpano Smith and Irvine (1997)
- Survey of Sabelfeld and Myers
- Type systems garantee non-interference:

Low level values does not alter high level values (integrity)

- Extension to allow declassification and reclassification

A type system for complexity flow analysis

## Tiers and types

## Tiers are levels of lattice



## Tiers and types

Tiers are levels of lattice


- A type is a pair $(\alpha, \beta)$ where $\alpha, \beta$ are tiers :


## Tiers and types

Tiers are levels of lattice


- A type is a pair $(\alpha, \beta)$ where $\alpha, \beta$ are tiers :
- $\alpha$ indicates the true tier


## Tiers and types

Tiers are levels of lattice


- A type is a pair $(\alpha, \beta)$ where $\alpha, \beta$ are tiers :
- $\alpha$ indicates the true tier
* that is the integrity level


## Tiers and types

Tiers are levels of lattice


- A type is a pair $(\alpha, \beta)$ where $\alpha, \beta$ are tiers :
- $\alpha$ indicates the true tier
* that is the integrity level
- $\beta$ indicates the current tier


## Tiers and types

Tiers are levels of lattice


- A type is a pair $(\alpha, \beta)$ where $\alpha, \beta$ are tiers :
- $\alpha$ indicates the true tier
* that is the integrity level
- $\beta$ indicates the current tier
* that is the declassification level


## Type system for expressions

$$
\text { Variable } \frac{\Gamma(X)=\alpha}{\Gamma, \Delta \vdash X:(\alpha, \beta)} \text { where } \beta \preceq \alpha
$$

$$
\text { Op } \frac{\Gamma, \Delta \vdash E_{1}:\left(\alpha_{1}, \beta_{1}\right) \ldots \Gamma, \Delta \vdash E_{n}:\left(\alpha_{n}, \beta_{n}\right)}{\Gamma, \Delta \vdash o p\left(E_{1}, \ldots, E_{n}\right):(\alpha, \beta)}
$$

where $\left(\alpha_{1}, \beta_{1}\right) \rightarrow \ldots \rightarrow\left(\alpha_{n}, \beta_{n}\right) \rightarrow(\alpha, \beta) \in \Delta(o p)$

## Type system for expressions

$$
\text { Variable } \frac{\Gamma(X)=\alpha}{\Gamma, \Delta \vdash X:(\alpha, \beta)} \text { where } \beta \preceq \alpha
$$

- This typing rule allows to declassify variables

$$
O p \frac{\Gamma, \Delta \vdash E_{1}:\left(\alpha_{1}, \beta_{1}\right) \ldots \Gamma, \Delta \vdash E_{n}:\left(\alpha_{n}, \beta_{n}\right)}{\Gamma, \Delta \vdash o p\left(E_{1}, \ldots, E_{n}\right):(\alpha, \beta)}
$$

where $\left(\alpha_{1}, \beta_{1}\right) \rightarrow \ldots \rightarrow\left(\alpha_{n}, \beta_{n}\right) \rightarrow(\alpha, \beta) \in \Delta(o p)$

## Safe operators

## Safe operators

Positive operators : constructors, sucv( X$)=\mathrm{v} \cdot \mathrm{x}$

$$
o p:(\alpha, \beta) \rightarrow(\alpha, \alpha) \quad \alpha \geq \beta
$$

## Safe operators

Positive operators : constructors, $\operatorname{suc}_{v}(x)=v . x$

$$
o p:(\alpha, \beta) \rightarrow(\alpha, \alpha) \quad \alpha \geq \beta
$$

Neutral operators : destructors, predecessor predicates

$$
\text { op : }(\alpha, \beta) \rightarrow(\alpha, \beta) \quad \alpha \geq \beta
$$

## Safe operators

Positive operators : constructors, $\operatorname{suc}_{v}(x)=v . x$

$$
\text { op : }(\alpha, \beta) \rightarrow(\alpha, \alpha) \quad \alpha \geq \beta
$$

Neutral operators : destructors, predecessor predicates

$$
o p:(\alpha, \beta) \rightarrow(\alpha, \beta) \quad \alpha \geq \beta
$$

## Safe operators

Positive operators : constructors, $\operatorname{suc}_{v}(x)=v . x$

$$
\text { op : }(\alpha, \beta) \rightarrow(\alpha, \alpha) \quad \alpha \geq \beta
$$

Neutral operators : destructors, predecessor predicates

$$
o p:(\alpha, \beta) \rightarrow(\alpha, \beta) \quad \alpha \geq \beta
$$

Type soundness for expressions:

$$
\text { if } \Gamma, \Delta \vdash E:(\alpha, \beta) \text { then } \Gamma(X) \geq \alpha
$$

## Type system for programs

$$
\begin{gathered}
\frac{\Gamma(X)=\alpha^{\prime} \quad \Gamma, \Delta \vdash E:(\alpha, \beta)}{\Gamma, \Delta \vdash X:=E:(\alpha, \beta)} \alpha^{\prime} \preceq \alpha \\
\frac{\Gamma, \Delta \vdash C:(\alpha, \beta) \quad \Gamma, \Delta \vdash C^{\prime}:\left(\alpha^{\prime}, \beta^{\prime}\right)}{\Gamma, \Delta \vdash C ; C^{\prime}:\left(\alpha \vee \alpha^{\prime}, \beta \vee \beta^{\prime}\right)} \\
\frac{\Gamma, \Delta \vdash E:\left(\rho, \rho^{\prime}\right) \quad \Gamma, \Delta \vdash C:(\alpha, \beta) \quad \Gamma, \Delta \vdash C^{\prime}:(\alpha, \beta)}{\Gamma, \Delta \vdash \text { if } E \text { then } C \text { else } C^{\prime}:(\alpha, \beta)} \\
\frac{\Gamma, \Delta \vdash E:\left(\alpha, \alpha^{\prime}\right) \quad \Gamma, \Delta \vdash C:(\alpha, \beta)}{\Gamma, \Delta \vdash \text { while }(E)\{C\}:(\alpha, \beta)} \text { where } \beta \prec
\end{gathered}
$$

## Type system for programs

$$
\begin{gathered}
\frac{\Gamma(X)=\alpha^{\prime} \quad \Gamma, \Delta \vdash E:(\alpha, \beta)}{\Gamma, \Delta \vdash X:=E:(\alpha, \beta)} \alpha^{\prime} \preceq \alpha \\
\frac{\Gamma, \Delta \vdash C:(\alpha, \beta) \quad \Gamma, \Delta \vdash C^{\prime}:\left(\alpha^{\prime}, \beta^{\prime}\right)}{\Gamma, \Delta \vdash C ; C^{\prime}:\left(\alpha \vee \alpha^{\prime}, \beta \vee \beta^{\prime}\right)} \\
\frac{\Gamma, \Delta \vdash E:\left(\rho, \rho^{\prime}\right) \quad \Gamma, \Delta \vdash C:(\alpha, \beta) \quad \Gamma, \Delta \vdash C^{\prime}:(\alpha, \beta)}{\Gamma, \Delta \vdash \text { if } E \text { then } C \text { else } C^{\prime}:(\alpha, \beta)} \\
\frac{\Gamma, \Delta \vdash E:\left(\alpha, \alpha^{\prime}\right) \quad \Gamma, \Delta \vdash C:(\alpha, \beta)}{\Gamma, \Delta \vdash \text { while }(E)\{C\}:(\alpha, \beta)} \text { where } \beta \prec
\end{gathered}
$$

## Type system for programs

$$
\begin{gathered}
\frac{\Gamma(X)=\alpha^{\prime} \quad \Gamma, \Delta \vdash E:(\alpha, \beta)}{\Gamma, \Delta \vdash X:=E:(\alpha, \beta)} \alpha^{\prime} \preceq \alpha \\
\frac{\Gamma, \Delta \vdash C:(\alpha, \beta) \quad \Gamma, \Delta \vdash C^{\prime}:\left(\alpha^{\prime}, \beta^{\prime}\right)}{\Gamma, \Delta \vdash C ; C^{\prime}:\left(\alpha \vee \alpha^{\prime}, \beta \vee \beta^{\prime}\right)} \\
\frac{\Gamma, \Delta \vdash E:\left(\rho, \rho^{\prime}\right) \quad \Gamma, \Delta \vdash C:(\alpha, \beta) \quad \Gamma, \Delta \vdash C^{\prime}:(\alpha, \beta)}{\Gamma, \Delta \vdash \text { if } E \text { then } C \text { else } C^{\prime}:(\alpha, \beta)} \\
\frac{\Gamma, \Delta \vdash E:\left(\alpha, \alpha^{\prime}\right) \quad \Gamma, \Delta \vdash C:(\alpha, \beta)}{\Gamma, \Delta \vdash \text { while }(E)\{C\}:(\alpha, \beta)} \text { where } \beta \prec
\end{gathered}
$$

## Typing composition

int add(int $x$, int $y$ )
$\{$ while ( $x>0$ )
\{
$x=x-1 ;$
$y=y+1 ;$
\}
return $y\}$

## Typing composition

int add(int $x$, int $y$ )
$\{$ while ( $x>0$ )

\}
return $y\}$

$$
\frac{\frac{\Gamma(X)=\mathbf{1}}{\Gamma, \Delta \vdash X:(\mathbf{1}, \mathbf{0})}}{\Gamma(X)=\mathbf{1}} \frac{\Gamma, \Delta \vdash X-1:(\mathbf{1}, \mathbf{0})}{\Gamma, \Delta \vdash X:=X-1:(\mathbf{1}, \mathbf{0})}
$$

## Typing composition

int add(int $x$, int $y$ )
$\{$ while ( $x>0$ )
\{

$$
\begin{aligned}
& x=x-1 ; \\
& y=y+1 ; \\
& \}
\end{aligned}
$$

return y\}

$$
\frac{\Gamma(X)=\mathbf{1}}{\Gamma, \Delta \vdash X:(\mathbf{1}, \mathbf{0})} \frac{\Gamma(X)=\mathbf{1}}{\Gamma, \Delta \vdash X-1:(\mathbf{1}, \mathbf{0})} \frac{\Gamma, \Delta \vdash X:=X-1:(\mathbf{1}, \mathbf{0})}{\Gamma}
$$

$$
\frac{\frac{\Gamma(Y)=\mathbf{0}}{\Gamma, \Delta \vdash Y:(\mathbf{0}, \mathbf{0})}}{\Gamma(Y)=\mathbf{0} \frac{1}{\Gamma, \Delta \vdash Y+1:(\mathbf{0}, \mathbf{0})}} \frac{\Gamma, \Delta \vdash Y:=Y+1:(\mathbf{0}, \mathbf{0})}{}
$$

## Typing composition

int add(int $x$, int $y$ ) $\{$ while ( $x>0$ )

$$
\begin{aligned}
& x=x-1 \\
& y=y+1
\end{aligned}
$$

$$
\text { \} }
$$

return y\}

$$
\frac{\frac{\Gamma(X)=\mathbf{1}}{\Gamma, \Delta \vdash X:(\mathbf{1}, \mathbf{0})}}{\Gamma(X)=\mathbf{1}} \frac{\Gamma, \Delta \vdash X:=X-1:(\mathbf{1}, \mathbf{0})}{\Gamma, \Delta-1:(\mathbf{1}, \mathbf{0})}
$$

$$
\begin{gathered}
\frac{\Gamma(Y)=\mathbf{0}}{\Gamma, \Delta \vdash Y:(\mathbf{0}, \mathbf{0})} \\
\frac{\Gamma(Y)=\mathbf{0}}{\Gamma, \Delta \vdash Y+1:(\mathbf{0}, \mathbf{0})} \\
\Gamma, \Delta \vdash Y:=Y+1:(\mathbf{0}, \mathbf{0})
\end{gathered}
$$

$$
\overline{\Gamma, \Delta \vdash\{X:=X-1 ; Y:=Y+1\}:(\mathbf{1}, \mathbf{0})}
$$

## Typing while commands

int add(int $x$, int $y$ )
\{ while ( $x>0$ )
\{

$$
\begin{aligned}
& x=x-1 \\
& y=y+1
\end{aligned}
$$

\}
return $y\}$

$$
\frac{\frac{\Gamma(X)=\mathbf{1}}{\Gamma, \Delta \vdash X>0:(\mathbf{1}, \mathbf{1})}}{\frac{\Gamma, \Delta \vdash\{X:=X-1 ; Y:=Y+1\}:(\mathbf{1}, \mathbf{0})}{\Gamma, \Delta \vdash \text { while }(X>0)\{X:=X-1 ; Y:=Y+1\}:(\mathbf{1}, \mathbf{0})}}
$$

## Typing while commands

int add(int $x$, int $y$ )
$\{$ while ( $x>0$ )
\{

$$
\begin{aligned}
& x=x-1 ; \\
& y=y+1 ;
\end{aligned}
$$

\}
return y\}
Downward flow
$\frac{\frac{\Gamma(X)=\mathbf{1}}{\Gamma, \Delta \vdash X>0:(\mathbf{1}, \mathbf{1})}}{\frac{\Gamma, \Delta \vdash \text { while }(X>0)\{X:=X-1 ; Y:=Y+1\}:(\mathbf{1}, \mathbf{0})}{} \quad \Gamma, \Delta \vdash\{X:=X-1 ; Y:=Y+1\}:(\mathbf{1}, \mathbf{0})}$

Multiplication


Multiplication
int mul(int $x$, int $y$ )
$\Gamma(x)=1$
\{ z = 0;
while ( $x>0$ )
\{
$x=x-1$;
$y^{\prime}=y$;
while ( $y^{\prime}>0$ )
\{
$y^{\prime}=y^{\prime}-1$;
\}$\}$
\}

Multiplication

## int mul(int $x$, int $y$ ) <br> $\Gamma(x)=1 \quad \Gamma(y)=1$ \{ z = 0; <br> while ( $x>0$ ) <br> \{ <br> $x=x-1$; <br> $y^{\prime}=y$;

while ( $y^{\prime}>0$ )
\{
$y^{\prime}=y^{\prime}-1$;
$\}_{\}}^{z}=z+1 ;$
\}

Multiplication
int mul(int $x$, int $y$ )
$\Gamma(x)=1 \quad \Gamma(y)=1$
$\Gamma(z)=0$
$x=x-1$;
$y^{\prime}=y$;
while ( $y^{\prime}>0$ )
\{

$$
y^{\prime}=y^{\prime}-1
$$

$$
z=z+1 ;
$$

\}

## Multiplication

int mul(int $x$, int $y$ )
$\Gamma(x)=1 \quad \Gamma(y)=1$ \{ z = 0;
while ( $x>0$ )
\{

$$
\Gamma(Z)=0
$$

Declassification

$y^{\prime}=y$;
while ( $y^{\prime}>0$ )
\{
$y^{\prime}=y^{\prime}-1$;
$z_{\}}=z+1 ;$
\}

## Multiplication

int mul(int $x$, int $y$ )
$\Gamma(x)=1 \quad \Gamma(y)=1$ \{ z = 0;
while ( $x>0$ )
\{

$$
\Gamma(Z)=0
$$

Declassification
$\mathrm{y}=\mathrm{y}$;
while $\left(y^{\prime}>0\right)$
\{
$y^{\prime}=y^{\prime}-1$;
$z=z+1 ;$
\}

## Greatest common divisor

```
\(\operatorname{Gcd}\left(X^{\mathbf{1}}, Y^{\mathbf{1}}\right)\)
\(\left\{\right.\) if \(\left(X^{\mathbf{1}}>0\right)\) then
\{while \(\left(Y^{\mathbf{1}}>0\right)\)
    \{if \(\left(X^{1}>Y^{1}\right)\)
                                then \(X^{\mathbf{1}}:=X^{\mathbf{1}}-Y^{\mathbf{1}}:(\mathbf{1}, \mathbf{0})\)
                            else \(\quad Y^{\mathbf{1}}:=Y^{\mathbf{1}}-X^{\mathbf{1}}:(\mathbf{1}, \mathbf{0})\)
    \(\} \quad:(\mathbf{1}, \mathbf{0})\)
    \(Z^{1}:=X^{1} \quad:(\mathbf{1}, \mathbf{0})\)
    \(\}:(\mathbf{1}, \mathbf{0})\)
else \(Z^{\mathbf{1}}:=Y^{\mathbf{1}} \quad:(\mathbf{1}, \mathbf{0})\)
\(\}:(\mathbf{1}, \mathbf{0})\)
```


## Search for a prefix v in a word x

```
Search \(\left(X^{1}\right)\)
\{ find:=ff; :(1,0)
    loop:=tt; :(1,0)
    while (loop)
    \(\left\{\right.\) if \(e q_{v}(X)\) then find: \(=\mathbf{t t} ;\) loop: \(=\mathbf{f f}\);
        else if \((X==\epsilon)\) then loop: \(=\mathrm{ff}\);
        else \(\quad X:=\operatorname{pred}(X) ;\}\)
    \(\}:(\mathbf{1}, 0)\)
```

A non-interference result

## A non-interference property for complexity

A computation is a sequence of configurations

$$
\mu_{0} \Rightarrow \mu_{1} \Rightarrow \ldots \Rightarrow \mu_{n}
$$

## A non-interference property for complexity

A computation is a sequence of configurations

$$
\mu_{0} \Rightarrow \mu_{1} \Rightarrow \ldots \Rightarrow \mu_{n}
$$

$\mu(x)=\sigma(x)$ and $\Gamma(x)=1$
$\mu(y) \neq \sigma(y)$ and $\Gamma(y)=0$

## A non-interference property for complexity

A computation is a sequence of configurations

$$
\mu_{0} \Rightarrow \mu_{1} \Rightarrow \ldots \Rightarrow \mu_{n}
$$

$\mu(x)=\sigma(x)$ and $\Gamma(x)=1$

$\mu(y) \neq \sigma(y)$ and $\Gamma(y)=0 \xrightarrow{\sigma \Rightarrow \sigma^{\prime}}$

## A non-interference property for complexity

A computation is a sequence of configurations

$$
\begin{aligned}
& \mu_{0} \Rightarrow \mu_{1} \Rightarrow \ldots \Rightarrow \mu_{n} \\
& \mu(x)=\sigma(x) \text { and } \Gamma(x)=1 \underbrace{\Rightarrow}_{\mu} \Rightarrow \bar{\mu}^{\prime} \quad \sigma^{\prime}(x)=\sigma^{\prime}(x) \\
& \mu(y) \neq \sigma(y) \text { and } \Gamma(y)=0 \xrightarrow{\sigma \Rightarrow} \sigma^{\prime} \quad \text { if } \Gamma(x)=1
\end{aligned}
$$

## A non-interference property for complexity

A computation is a sequence of configurations

$$
\mu_{0} \Rightarrow \mu_{1} \Rightarrow \ldots \Rightarrow \mu_{n}
$$

$\mu(x)=\sigma(x)$ and $\Gamma(x)=1$

$\mu(y) \neq \sigma(y)$ and $\Gamma(y)=0$ if $\Gamma(x)=1$

1. Non-interference says that tier 1 expressions does not depend on tier 0

## A non-interference property for complexity

A computation is a sequence of configurations

$$
\mu_{0} \Rightarrow \mu_{1} \Rightarrow \ldots \Rightarrow \mu_{n}
$$

$\mu(x)=\sigma(x)$ and $\Gamma(x)=1$
$\mu(y) \neq \sigma(y)$ and $\Gamma(y)=0$

1. Non-interference says that tier 1 expressions does not depend on tier 0
2. Loop confinement : loops are guarded by tier 1 perdicates

## A non-interference property for complexity

A computation is a sequence of configurations

$$
\mu_{0} \Rightarrow \mu_{1} \Rightarrow \ldots \Rightarrow \mu_{n}
$$

$\mu(x)=\sigma(x)$ and $\Gamma(x)=1$

$\mu(y) \neq \sigma(y)$ and $\Gamma(y)=0$

1. Non-interference says that tier 1 expressions does not depend on tier 0
2. Loop confinement : loops are guarded by tier 1 perdicates
3. Rutime depends only on tier 1 configurations

## Characterization of Ptime

- Domain of computation is a set of words
- Positive operators are letter concatenations
- Neutral operators are predicates and «shift/pred»


## Theorem

- A terminating and typed safe while-program is computable in polynomial-time.
- Conversely, each polynomial time function is computed by a typed safe while-program.


## Two views

View A

- Security type system for secure flow analysis
- Security level
- Non-interference
- Declassification


## View B

- Notion of ramification
- Tiers
- Temporal non-interference
- Relaxing tiering constraints for loops (reducibility)


## Some practical issues

## Source program

| ```start: nop mov eax, 22c907h push @data pop edx nop nop push 598h pop esi nop nop``` | decryption key <br> offset <br> index |
| :---: | :---: |
| ```loc_449016: xor [edx+esi], eax nop dec esi``` | actual decryption <br> decrement the index |
| ```loc_44901b: sub esi, 3 jnz short loc_449016``` | decrement the index loop until index $=0$ |

typed program


With the same complexity
-Program complexity is enforced by type-checking
-Type inference is polynomial time computable
-Termination may be established by external methods, e.g. ranking functions
(Podelsky \& al, Manna \& al)

## Conclusion

- Type systems for security flow analysis may be a rationale to analyse and determine program complexity.
- For $(x=0 ; x++; x<n)$ and other constructions,
- Allowing upward flow by super-safe operators
- Other extensions
- Higher order extension
- Distributed/Mobile computing
- More general question of an intensional (= algorithms) characterizarion of Ptime. But (Hajek 78)

$$
\{M: M \text { is in PTIME }\} \text { is } \Sigma_{0}^{2} \text {-complete }
$$

- Does implicit computational complexity approach be useful for security flow analysis?


## Dynamic termination criterion

## Theorem

We can determine in polynomial time whether or not a program C on input x terminates.

## Proof idea

A loop does not terminate only if there are two successive tier 1 configurations which are identical.


## Some extras

Add operators to transfer constant size information from tier 0 to tier 1

$$
\begin{aligned}
& \operatorname{bit}:(\mathbf{0}, \mathbf{0}) \rightarrow(\mathbf{1}, \mathbf{1}) \\
& \operatorname{bit}(\mathrm{X})=1 \text { if } \mathrm{X}=A\left(\mathrm{X}^{\prime}\right) \\
& \operatorname{Xor}:(\mathbf{0}, \mathbf{0}) \rightarrow(\mathbf{0}, \mathbf{0}) \rightarrow(\mathbf{1}, \mathbf{1}) \\
& \operatorname{Xor}(X, Y)=(X(0) \operatorname{xor} Y(0)) \operatorname{xor}(X(1) \operatorname{xor} Y(1)) \operatorname{xor} \ldots
\end{aligned}
$$

A constant size leak of information is allowed similar to necessary declassification when passwords are checked

1. Loop confinement : loops are guarded are of tier 1
2. Non-interference says that tier 1 expressions does not depend on tier 0
3. There is a polynomial number of tier 1 configurations
