Computer Science and Privacy
Non Interference Analyzes

F. Prost
Frederic.Prost@ens-lyon.fr

Ecole Normale Supérieure de Lyon

September 2015
From a programming point of view the question of privacy becomes: how can we prove/certify that a program does not reveal secret information to the public space?
From a programming point of view the question of privacy becomes: how can we prove/certify that a program does not reveal secret information to the public space?

It is an instance of the more general problem of non-interference: $x, y$ do not interfere in $P$ if any modification on the value of $x$ cannot be “observed” on $y$. 
From a programming point of view the question of privacy becomes: how can we prove/certify that a program does not reveal secret information to the public space?

It is an instance of the more general problem of non-interference: $x, y$ do not interfere in $P$ if any modification on the value of $x$ cannot be "observed" on $y$.

Non-Interference is a very general problem:
- Proof-theory: useless hypotheses.
- Non-computational content of proofs: extraction of programs through the Curry-Howard correspondence.
- Parallelism.
- Strictness analysis.
- etc.
Non-interference analyzes

- NI analysis depends very much on the semantics and programming paradigm in use.
Non-interference analyzes

- NI analysis depends very much on the semantics and programming paradigm in use.
  - How do we model the fact that two programs are “equivalent”?
  - What is the exact nature or quality of “observations”?

It is a very strict approach to privacy: for instance a password check is an interference.

Can we define policies allowing such interferences?

Non-Interference is a yes/no approach.

Can we quantify the amount of information released?
Non-interference analyzes

- NI analysis depends very much on the semantics and programming paradigm in use.
  - How do we model the fact that two programs are “equivalent”?
  - What is the exact nature or quality of “observations”?
- It is a very strict approach to privacy: for instance a password check is an interference.
Non-interference analyzes

- NI analysis depends very much on the semantics and programming paradigm in use.
  \[\Rightarrow\] How do we model the fact that two programs are “equivalent”?
  \[\Rightarrow\] What is the exact nature or quality of “observations”?

- It is a very strict approach to privacy: for instance a password check is an interference.
  \[\Rightarrow\] Can we define policies allowing such interferences?

- Non-Interference is a yes/no approach.
  \[\Rightarrow\] Can we quantify the amount of information released?
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
Dependencies in pure λ-calculus [Abadi et al., 1996]

- What parts of a term contributes to the final result?
  Suppose $a \rightarrow^* v$, what can be removed from $a$ while still having a term reducing to $v$?

- Pure λ-terms:
  \[ t ::= x \mid \lambda x. t \mid (t_1 \ t_2) \]

- Prefixes:
  \[ p ::= _ \mid x \mid \lambda x. p \mid (p_1 \ p_2) \]
Prefix order

- How to formalize the idea of a “useless” part in a term?
Prefix order

- How to formalize the idea of a “useless” part in a term?
- Let $a, b$ be prefixes: $a \preceq b$ if $a$ can be produced by $b$ replacing some subterms with `_`.
- Example:

  $$(\lambda x.(t \_ \_) \_) \preceq (\lambda x.(t \ u) \ v)$$
Prefix order

- How to formalize the idea of a “useless” part in a term?
- Let $a, b$ be prefixes: $a \leq b$ if $a$ can be produced by $b$ replacing some subterms with `_`.
- Example:
  \[
  (\lambda x. (t \_ \_)) \leq (\lambda x. (t \ u) \ v)
  \]
- The question is: how $\leq$ behaves wrt $\beta$-reduction?
Two results about $\preceq$

Théorème (Monotonicity)

$t \preceq u$

$t' \preceq u'$
Two results about $\preceq$

**Théorème (Monotonicity)**

\[
\begin{array}{c}
t \preceq u \\
\downarrow \quad \quad \quad \downarrow \\
* t' \preceq * u'
\end{array}
\]

**Théorème (Stability)**

*If $a \rightarrow^* v$, and $v$ is in normal form, there is a minimal prefix $a_0 \preceq a$ such that $a_0 \rightarrow^* v$.***

The minimal prefix is the *mathematical* solution of the non-interference computation:
how can it be effectively computed?
Labeled terms

- Extension of the pure \( \lambda \)-calculus:

\[
l ::= x \mid \lambda x.l \mid (l_1 l_2) \mid e : l
\]
Labeled terms

- Extension of the pure λ-calculus:

\[ l ::= x \mid \lambda x.l \mid (l_1 \ l_2) \mid e : l \]

- We add the reduction rule:

\[ (e : l_1 \ l_2) \rightarrow e : (l_1 \ l_2) \]

which makes possible the usual β-reduction:

\[ (e_0 : [\lambda x.(x \ x)] \ e_1 : y) \rightarrow e_0 : (\lambda x.(x \ x) \ e_1 : y) \rightarrow e_0 : (e_1 : y \ e_1 : y) \]
Minimal prefix computation

1. Attribute a unique label to each subterm of $a$.

2. If $a$ has $\text{nf } \nu$, we write $L(a)$, the set of all labels occurring in $\nu$.

3. Define $G(a)$ as the one obtained by replacing each subterm of $a$ whom the label is not in $L(a)$ by $\_$. 

\[
(ef : (\lambda x. e_5 : 5) \ e_t : t) \rightarrow ef : (\lambda x. e_5 : 5 \ e_t : t) \rightarrow ef : e_5 : 5
\]

Hence $t$ does not interfere with the rest of the program.
NI in a typed setting [Berardi, 1996]

- How to *statically* compute the minimum prefix?
- The problem in its whole generality undecidable.
NI in a typed setting [Berardi, 1996]

- How to \textit{statically} compute the minimum prefix?
- The problem in its whole generality undecidable. Easy reduction to the halting problem.
How to \textit{statically} compute the minimum prefix?

The problem in its whole generality undecidable. Easy reduction to the halting problem.

Is it possible to statically \textit{approximate} the result?

\implies yes with a surprising use of types in the simply typed $\lambda$-calculus.
Dependencies in simply-typed $\lambda$-calculus

Simply typed $\lambda$-calculus with base type $\mathcal{N}$ and constants $S : \mathcal{N} \rightarrow \mathcal{N}$ and $0 : \mathcal{N}$.
Dependencies in simply-typed $\lambda$-calculus

- Simply typed $\lambda$-calculus with base type $\mathcal{N}$ and constants $S : \mathcal{N} \rightarrow \mathcal{N}$ and $0 : \mathcal{N}$.
- Introduction of a constant $\emptyset$, only term of type $\mathcal{U}$.
- Definition of an order relation $\preceq$ w.r.t. $\emptyset$.
- How do we define two equivalent terms?
Dependencies in simply-typed $\lambda$-calculus

- Simply typed $\lambda$-calculus with base type $\mathcal{N}$ and constants $S : \mathcal{N} \rightarrow \mathcal{N}$ and $0 : \mathcal{N}$.
- Introduction of a constant $\emptyset$, only term of type $\mathcal{U}$.
- Definition of an order relation $\preceq$ w.r.t. $\emptyset$.
- How do we define two equivalent terms?
- Two terms $t_1, t_2 : A$ are observationnally equivalents iff:

$$\forall C[^A] : \mathcal{N}, C[t_1] =^\beta C[t_2]$$
Dead code in simply typed $\lambda$-calculus

Théorème ([Berardi, 1996])

If $t, t' : A$ and $t \preceq t'$ then $t$ and $t'$ are observationally equivalent.
Dead code in simply typed λ-calculus

Théorème ([Berardi, 1996])

If $t, t' : A$ and $t \preceq t'$ then $t$ and $t'$ are observationnally equivalent.

Proof.
If $A = \mathcal{N}$ then by subject reduction, strong normalization and monotonicity we have $t \rightarrow^* v$ and $t' \rightarrow v'$ with $v \preceq v'$, but closed nf of type $\mathcal{N}$ are either 0 or $(S \ldots (S 0) \ldots)$, hence $v \equiv v'$. 
Dead code in simply typed $\lambda$-calculus

Théorème ([Berardi, 1996])

If $t, t' : A$ and $t \leq t'$ then $t$ and $t'$ are observationnally equivalent.

Proof.

If $A = N$ then by subject reduction, strong normalization and monotonicity we have $t \to^* v$ and $t' \to v'$ with $v \leq v'$, but closed nf of type $N$ are either 0 or $(S \ldots (S 0) \ldots)$, hence $v \equiv v'$.

For any other type $A$ take any closing context $C[.^A] : N$. Then $C[t] \leq C[t']$, and $C[t], C[t'] : N$, hence we can apply the previous reasonning.

An example:

$$(\lambda x : U.5 \emptyset) \leq (\lambda x : N.5 \ t)$$
How can we actually compute the minimal prefix?

- The typing tree has to be analyzed from root to leaves.
- Each type may be annotated with its privacy/dead code level (actually a fresh variable at the type of the root of the tree).
- Typing constraints are resolved and each privacy/dead code level variable that occurs at the root of the tree is set to be $\top$.
- Very similar to Caml type inference (but simpler).
How can we actually compute the minimal prefix?

- The typing tree has to be analyzed from root to leaves.
- Each type may be annotated with its privacy/dead code level (actually a fresh variable at the type of the root of the tree).
- Typing constraints are resolved and each privacy/dead code level variable that occurs at the root of the tree is set to be $\top$.
- Very similar to Caml type inference (but simpler).

\[
\Gamma, x : \mathcal{N}^\epsilon \vdash 5 : \mathcal{N}^\delta \\
\Gamma \vdash \lambda x.5 : \mathcal{N}^\epsilon \rightarrow \mathcal{N}^\delta \\
\Gamma \vdash t : \mathcal{N}^\epsilon
\]

\[
\Gamma \vdash + : \mathcal{N}^\delta \rightarrow \mathcal{N}^\beta \rightarrow \mathcal{N}^\gamma \\
\Gamma \vdash (\lambda x.5 \; t) : \mathcal{N}^\delta \\
y : \mathcal{N}^\alpha, z : \mathcal{N}^\beta \vdash (+ (\lambda x.5 \; t) \; z) : \mathcal{N}^\beta \rightarrow \mathcal{N}^\gamma \\
\Gamma \vdash \lambda y, z.(+ (\lambda x.5 \; t) \; z) : \mathcal{N}^\alpha \rightarrow \mathcal{N}^\beta \rightarrow \mathcal{N}^\gamma
\]
Problems linked with the unicity of typing

Type unicity + Conservative approximation = less accurate analyzis

\[ t = (\lambda f : N \rightarrow N. (g \ f \ (f \ 5)) \ \lambda x : N.4) \]

We would like to type first occurrence of \( f \) with \( N \rightarrow N \) and the second one with \( U \rightarrow N \).
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
The $\lambda$-cube [Barendregt, 1991]

- **Terms:** $\mathcal{T} ::= V \mid C \mid (\mathcal{T} \cdot \mathcal{T}) \mid \lambda V : \mathcal{T}.\mathcal{T} \mid \Pi V : \mathcal{T}.\mathcal{T}$
- **Parameters:**
  - $\mathcal{S}$: sorts,
  - $\mathcal{A}$, axioms of the form $c : s$,
  - $\mathcal{R}$, rules of the form $(s_1, s_2, s_3)$. We write $(s_1, s_2)$ when $s_3 = s_2$.
  
  Rules define valid product:
  \[
  \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad \frac{}{\Gamma \vdash \Pi x : A. B : s_3}
  \]
  if $(s_1, s_2, s_3) \in \mathcal{R}$.
- **Computation rule:** $(\lambda x : A. B \ C) \rightarrow_{\beta} B[x := C]$
Pure Type Systems Rules

\[ \vdash c; s \ (c : s \in A) \]

\[ \Gamma \vdash A : s \quad \Gamma, x : A \vdash x : A \ (x \not\in \Gamma) \]

\[ \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2, s_3) \in \mathcal{R} \quad \Gamma \vdash \Pi x : A. B : s_3 \]

\[ \Gamma \vdash F : (\Pi x : A. B) \quad \Gamma \vdash a : A \quad \Gamma \vdash (F \ a) : B [x := a] \]

\[ \Gamma \vdash B =_\beta B' \]

\[ \Gamma \vdash A : B \quad \Gamma \vdash C : s \ (x \not\in \Gamma) \]

\[ \Gamma, x : C \vdash A : B \]

\[ \Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B) \]

\[ \Gamma, x : A \vdash b : B \]

\[ \Gamma \vdash (\Pi x : A. B) : s \]
The $\lambda$-cube

- Take sorts: $\{\ast, \Box\}$, and axiom ($\ast : \Box$).
- We consider only rules of the form $(s_1, s_2)$.
- We have four possible rules:
  \begin{align*}
  \{(\ast, \ast), (\Box, \ast), (\ast, \Box), (\Box, \Box)\}
  \end{align*}
Intuitions behind rules

- $(\ast, \ast)$: simply typed $\lambda$-calculus.
- $(\Box, \ast)$: polymorphism.
- $(\Box, \Box)$: possibility to build connective.
- $(\ast, \Box)$: dependent types.
Higher-order Types

NI in a Purely Functional Setting

<table>
<thead>
<tr>
<th>System</th>
<th>Historical name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda \rightarrow$</td>
<td>Simply typed $\lambda$-calculus [Church, 1940]</td>
</tr>
<tr>
<td>$\lambda_2$</td>
<td>System F [Girard, 1972]</td>
</tr>
<tr>
<td>$\lambda P$</td>
<td>AUT-QE; LF [Bruijn, 1970]</td>
</tr>
<tr>
<td>$\lambda P_2$</td>
<td>[Longo and Moggi, 1988]</td>
</tr>
<tr>
<td>$\lambda \omega$</td>
<td>POLYREC</td>
</tr>
<tr>
<td>$\lambda_\omega$</td>
<td>[de Lavalette, 1992]</td>
</tr>
<tr>
<td>$\lambda C$</td>
<td>Calculus of Constructions</td>
</tr>
<tr>
<td></td>
<td>[Coquand and Huet, 1988]</td>
</tr>
</tbody>
</table>

$\lambda$-cube [Barendregt, 1991]
Variable sorts

- “$A$ is a type” is a judgment:
  \[
  \Gamma \vdash A : * 
  \]

- We introduce judgments of the form:
  \[
  \Gamma \vdash A : \alpha 
  \]
  where $\alpha$ is a sort variable ranging over $*_\bot, *\top$.

- The two different $*$s are used to denote separated universes.

- We add axiom
  \[
  *_i : \square \text{ for } i \in \{\top, \bot\} 
  \]
Sort abstraction

- Abstracting sorts w.r.t. terms:

\[
\frac{\Gamma, \alpha : \Box, \Gamma' \vdash t : B \quad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash (\lambda \alpha : \Box. t) : (\Pi \alpha : \Box. B)}
\]

- Sort application:

\[
\frac{\Gamma \vdash t : (\Pi \alpha : \Box. A) \quad \Gamma \vdash k : \Box}{\Gamma \vdash (t \ k) : A[\alpha := k]}
\]
Examples

\[ N^\alpha \overset{\text{def}}{=} \prod X : \alpha. X \rightarrow (X \rightarrow X) \rightarrow X \]

\[ n^\alpha \overset{\text{def}}{=} \lambda X : \alpha. \lambda x : X. \lambda f : X \rightarrow X. (f \ldots (f \ x) \ldots ) \]

\[ \alpha : \square, \beta : \square, y : N^\alpha \vdash 5^\beta : N^\beta \]

\[ \alpha : \square, \beta : \square \vdash \lambda y : N^\alpha. 5^\beta : (N^\alpha \rightarrow N^\beta) \]
$k$-types, $k$-constants

- $k$-type: Type where the only occurring sort is $k$. Example of $\ast_{\bot}$-type:

  $$\Pi X : \ast_{\bot} . \Pi Y : \ast_{\bot} . X \rightarrow Y \rightarrow (\Pi Z : \ast_{\bot} . Z \rightarrow X)$$

- For all $k$-type $A$, we define a constant $d_A$ of type $A$.

- We define an order $\leq_k$ w.r.t. $k$-constants:

  $$(\lambda x : \mathcal{N}^{\ast_{\bot}} . \overline{5}^{\ast_{\bot}} \ d_{\mathcal{N}^{\ast_{\bot}}}) \leq_{\ast_{\bot}} (\lambda x : \mathcal{N}^{\ast_{\bot}} . \overline{5}^{\ast_{\bot}} \ t)$$

  if $t$ is of type $\mathcal{N}^{\ast_{\bot}}$, for instance.
Non-interference and types

- A first result:

**Theorem (Non-interference)**

Let \( x : A \vdash t : B, \) \( A \) a \(*_{\bot}\)-type, \( B \) a \(*_{\top}\)-type then for all \(<\rangle\vdash t_1, t_2 : A \) one has:

\[
t[x := t_1] =_{ob} t[x := t_2]
\]

- A corollary:

**Theorem (Dead-code)**

If \(<\rangle\vdash t_1, t_2 : A, \) and \( A \) a \(*_{\top}\)-type, and \( t_1 \leq_{*_{\bot}} t_2, \) then

\[
t_1 =_{ob} t_2
\]
Example 1

Let $t$ be such that $\llbracket t : N^\ast \bot \rrbracket$, then terms

$$t_1 = (\lambda y : N^\ast \bot. 5^\ast \top t),$$

$$t_2 = (\lambda y : N^\ast \bot. 5^\ast \top d_{N^\ast \bot})$$

are both of type $N^\ast \top$.

$t_2 \leq_{\ast \bot} t_1$.

Then from theorem 2, we conclude $t_1 =_{\text{obs}} t_2$. 
Example 2

The ability to abstract over sorts introduces flexibility:

\[
t = ( \lambda f : \Pi \alpha, \beta : \Box. N^{\alpha} \to N^{\beta} (g (f \ast \top \ast \top) ((f \ast \bot \ast \top) t')) )
\]

\[
\lambda \alpha, \beta : \Box. \lambda x : N^{\alpha}.5^{\beta}
\]

with \( g \) of type \( (N^{* \top} \to N^{* \top}) \to N^{* \top} \to N^{* \top} \), and \( t' \) of type \( N^{* \bot} \). 
\( t \) is of type \( N^{\top} \), and \( t' \) analyzed as dead-code.
Sort abstraction in “cube” style

- Addition of sort: $\triangle$;
- Addition of axiom: $\Box : \triangle$;
- Addition of rule: $(\triangle, *)$;

$$
\Gamma \vdash \Box : \triangle \quad \Gamma, \alpha : \Box \vdash A : \ast \\
\Gamma \vdash (\Pi \alpha : \Box. A) : \ast
$$
Higher-order Types

(□, *) (□, □) (□, *) (□, □) (□, *)

Hyper λ-cube
It is possible to prove theorems 1 and 2 in the $\mathcal{E}$-cube: a non-interference result for the Calculus of Constructions.

The rule $(\triangle, \ast)$ expresses the logical content of type-based analyses.
Technical considerations

- Original formalism has been extended in order to have judgments like
  \[ x : X : \alpha : \Box \]

  where \( x, X, \alpha \) are variables.

- In \( \lambda \alpha : \Box . A \), \( \alpha \) is a **weak** variable, i.e. it stands either for \( \ast \top \) or \( \ast \bot \).

- The work done is of theoretical nature.

- Hint for an algorithm: ML unification modified (not complete).
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
Interferences in imperative programs
[Volpano and Smith, 1997]

- Programs input and output are classified at different security levels.
- We would like to allow the information to go up but never down w.r.t. security levels.
- The security can be expressed by comparing the memory of the computer regarding the different levels of security (different from the functional approach in which there are no variables).
- Simple imperative programming language with procedures.
- Type soundness result: if a program is well typed, then non-interference is enforced.
Some Information Leaking Programs and Non-Termination

for $i = 0$ to secret
  output $i$ on public_channel
Some Information Leaking Programs and Non-Termination

for i = 0 to secret
    output i on public_channel

for i = 0 to secret
    output i on public_channel
while true do skip
for i = 0 to secret
    output i on public_channel

for i = 0 to secret
    output i on public_channel
while true do skip

for i = 0 to maxNat {
    output i on public_channel
    if (i = secret) then (while true do skip)
}
Three kinds of types

- \( \tau \)-types: security levels.
- \( \pi \)-types: expressions and commands.
- \( \rho \)-types: types of phrases.

For instance \( \tau \in \{h, l\} \) with \( l \leq h \).

Command types have form \( \tau \text{ cmd} \). A command of type \( h \text{ cmd} \) says it does not contain assignment to low variables.

Phrase types are of the form \( \tau \text{ var} \) or \( \tau \text{ acc} \).

The subtype relation is contravariant in command and acceptor types and covariant on expressions.
Information flow

- Direct information flow: \( l := h \)
Information flow

- Direct information flow: \( l := h \)
- Indirect information flow
  
  While \( h > 0 \) do
  
  \( l := l + 1; \)
  
  \( h := h - 1; \)
  
  od
Information flow

- Direct information flow: \( l := h \)
- Indirect information flow
  
  While \( h > 0 \) do
  
  \( l := l + 1 \);
  
  \( h := h - 1 \);
  
  od

- We must have typing rules forbidding such programs:

\[
\frac{\gamma \vdash e : \tau \quad \gamma \vdash c : \tau \text{ cmd}}{\gamma \vdash \text{while } e \text{ do } c : \tau \text{ cmd}}
\]
One needs to define the operational semantics of the programming language: \[ \mu \vdash c \implies \mu' \]

One needs to define a notion of “equivalent” memories \( \mu \simeq_1 \nu \) if \( \mu \) and \( \nu \) agree on the value of low-level variables.

The non-interference property can be stated as:

- Suppose that \( \lambda \vdash c : \pi \)
- Suppose that \( \mu \vdash c \implies \mu' \)
- Suppose that \( \nu \vdash c \implies \nu' \)
- Suppose that \( \mu \simeq_\tau \nu \simeq_\tau \lambda \)

then \( \nu'(l) = \mu'(l) \) for all \( l \) such that \( \lambda(l) \leq \tau \).
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
Process interlock and information leakage

\[ \alpha \iff [c_\alpha = 0 \Rightarrow SPY := 0 ; c_\beta := 0] ; \theta \]
Process interlock and information leakage

\[ \alpha \iff [c_\alpha = 0 \Rightarrow SPY := 0; \ c_\beta := 0]; \ \theta \]

\[ \beta \iff [c_\beta = 0 \Rightarrow SPY := 1; \ c_\alpha := 0]; \ \theta \]
Process interlock and information leakage

\[
\alpha \leftarrow [c_\alpha = 0 \Rightarrow SPY := 0; c_\beta := 0]; \theta
\]
\[
\beta \leftarrow [c_\beta = 0 \Rightarrow SPY := 1; c_\alpha := 0]; \theta
\]
\[
\gamma \leftarrow ([PIN = 1 \Rightarrow c_\alpha := 0]; \theta) +
([PIN = 0 \Rightarrow c_\beta := 0]; \theta)
\]

\[\alpha \parallel \beta \parallel \gamma\]
\( \lambda_{ar} \) [Prost, 2005]: \( \lambda \)-calculus with adressed resources

- Variation of the blue-calculus of G. Boudol (variant of Milner’s polyadic \( \pi \)-calculus).

- Terms:

\[
t :::= x, a \mid (t \ t) \mid \lambda x. t \\
\mid t \parallel t \\
\mid \nu a(t) \\
\mid (t \ s) \mid (s \ t)
\]

- Adressed resources:

\[
s :::= \langle a \leftrightarrow t \rangle \mid \langle a = t \rangle \mid (s \ s)
\]
Operational Semantics

Definition (Reduction rules)

\[(\lambda x. t \ u) \rightarrow_\beta t\{x := u\}\]
\[t \ | \ \langle a \leftarrow u \rangle \rightarrow_\rho t\{a := u\}\]

- Communication example:

\[\langle a \leftarrow \lambda x. t \rangle \ | \ (a \ v) \rightarrow_\rho (\lambda x. t \ v) \rightarrow_\beta t\{x := v\}\]
\( \lambda_{ar} \) Typing

- It is possible to have a fine-grained typing of \( \lambda_{ar} \):

\[
[PPAR] \quad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t \parallel u : Pa(\tau, \sigma)(\tau, \sigma \neq \circ)}
\]

- Sort abstraction “à la” [Prost00] leads to similar result than in \( \lambda \)-calculus.
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
A lot of every-day life scenarios involve dynamic evolution of data privacy levels.
A lot of every-day life scenarios involve dynamic evolution of data privacy levels.

- Pay-per-view;
Non-Interference Dynamic policies [Prost, 2011]

- A lot of every-day life scenarios involve dynamic evolution of data privacy levels.
  - Pay-per-view;
  - Sealed auctions;
Non-Interference Dynamic policies [Prost, 2011]

- A lot of every-day life scenarios involve dynamic evolution of data privacy levels.
  - Pay-per-view;
  - Sealed auctions;
  - etc.

- Challenge: to adapt non-interference to fit with dynamic evolution of privacy?
A lot of every-day life scenarios involve dynamic evolution of data privacy levels.

- Pay-per-view;
- Sealed auctions;
- etc.

Challenge: to adapt non-interference to fit with dynamic evolution of privacy?

In our framework we propose:

1. A “security profile” for each operator: rewrite rules over privacy lattice.
Non-Interference Dynamic policies [Prost, 2011]

- A lot of every-day life scenarios involve dynamic evolution of data privacy levels.
  - Pay-per-view;
  - Sealed auctions;
  - etc.

- Challenge: to adapt non-interference to fit with dynamic evolution of privacy?

- In our framework we propose:
  1. A “security profile” for each operator: rewrite rules over privacy lattice.
  2. Rewrite rules may have actions modifying the policy.
Non-Interference Dynamic policies [Prost, 2011]

- A lot of every-day life scenarios involve dynamic evolution of data privacy levels.
  - Pay-per-view;
  - Sealed auctions;
  - etc.

- Challenge: to adapt non-interference to fit with dynamic evolution of privacy?

- In our framework we propose:
  1. A “security profile” for each operator: rewrite rules over privacy lattice.
  2. Rewrite rules may have actions modifying the policy.
  3. Definition of high/low bisimulation with dynamic policies.
Non-Interference Dynamic policies [Prost, 2011]

- A lot of every-day life scenarios involve dynamic evolution of data privacy levels.
  - Pay-per-view;
  - Sealed auctions;
  - etc.

- Challenge: to adapt non-interference to fit with dynamic evolution of privacy?

- In our framework we propose:
  1. A “security profile” for each operator: rewrite rules over privacy lattice.
  2. Rewrite rules may have actions modifying the policy.
  3. Definition of high/low bisimulation with dynamic policies.
  4. Program safety verification by abstract execution on privacy levels.
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
LINE Programming language

- Minimalistic programming language:

\[
\begin{align*}
\mathit{v} &::= x \mid 0 \mid 1 \mid 0 \mid 1 \mid \ldots \\
\mathit{t}, \mathit{b} &::= \mathit{v} \mid \mathit{f}(x_1, \ldots, x_n) \\
\mathit{P} &::= x := t \mid \mathit{P} ; \mathit{P} \mid \text{if } b \text{ then } \mathit{P} \text{ else } \mathit{P} \mid \text{while } b \text{ do } \mathit{P} \mid \text{skip}
\end{align*}
\]

- It can be seen as an intermediate language:

\[x := \mathit{f}(345, \mathit{g}(x_1, x_2)) \equiv (x_0 := 345; x_3 := \mathit{g}(x_2, x_3); x := \mathit{f}(x_0, x_3)\]

- Natural semantics \(\langle \mu, \mathit{P} \rangle \rightarrow_{\text{os}} \langle \mu', \mathit{P}' \rangle\)
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
Interference policy

- Program variables are attributed privacy levels.
- Privacy levels are elements of a lattice $\mathcal{L}$.
- Interference policies are based on authorised behavior of operators.
Interference policy

- Program variables are attributed privacy levels.
- Privacy levels are elements of a lattice $\mathcal{L}$.
- Interference policies are based on authorised behavior of operators.
- Usually it is done through types but it is too rigid.

$\Rightarrow$ We use term rewriting system on privacy levels in order to deal with concrete privacy levels used at evaluation time.
Static interference policy

- For each operator $f$ we consider $f_{DIP}$.
- $\Sigma_{DIP} = (\mathcal{V}_{DIP}, \mathcal{V} \cup \Omega_{DIP} \cup \mathcal{L})$
- $x \in \mathcal{V}_{DIP}$ are variables of the rewrite system whereas $x$ is a program variable that has to be seen as a constant in the rewrite system.
- Encryption policy, $\mathcal{SP}$:
  
  $\text{encrypt}_{DIP}(\pi_{128}, x) \rightarrow \pi_1$
  $\text{encrypt}_{DIP}(\pi_{256}, x) \rightarrow \bot$
  $\text{SPY}_{DIP} \rightarrow \bot$
  $\text{PIN}_{DIP} \rightarrow \top$
  ...

- In the program: $\text{SPY} := \text{encrypt}(K, \text{PIN})$
  The security level of $\text{encrypt}(K, \text{PIN})$ is computed using rules of $\mathcal{SP}$. 
Dynamicity

- Privacy levels may change during computation.

- Rewriting rules with actions: $l \rightarrow r; a$

  $$a ::= x \mapsto \pi \mid \overline{x} \mapsto \pi \mid \overline{x} \mapsto y \mid x \mapsto \overline{y} \mid a; a$$

- The interference policy changes through the evaluation of operator security level computation:

  $$\langle t[\sigma(l)], SP \rangle \leadsto \langle t[\sigma(r)], SP' \rangle$$
Dynamicity

- For an operator of arity $n$ we consider an operator of arity $2 \times n$ in the $\mathcal{SP}$.
  $\implies$ to make the distinction between the privacy level and the identity of a parameter.
- Hence, in fact encrypt($K_{128}$, $PIN$) is represented with encrypt$_{DIP}(\pi_{128}, K_{128}, \pi_{PIN}, PIN)$
Dynamicity

- For an operator of arity $n$ we consider an operator of arity $2 \times n$ in the $SP$. 
  - to make the distinction between the privacy level and the identity of a parameter.
- Hence, in fact encrypt($K_{128}$, PIN) is represented with encrypt$_{DIP}$($\pi_{128}$, $K_{128}$, $\pi_{PIN}$, PIN)
- Example: aging process for encryption keys (we drop the DIP subscripts)

\[
\begin{align*}
\text{encrypt}(\pi_{128/1}, \overline{x}, \overline{y}, \overline{z}) & \rightarrow \pi_{1}; \overline{x} \mapsto \pi_{128/2} \\
\text{encrypt}(\pi_{128/2}, \overline{x}, \overline{y}, \overline{z}) & \rightarrow \pi_{1}; \overline{x} \mapsto \pi_{128/3} \\
\text{encrypt}(\pi_{128/3}, \overline{x}, \overline{y}, \overline{z}) & \rightarrow \top; \overline{x} \mapsto \top \\
\text{encrypt}(\top, \overline{x}, \overline{y}, \overline{z}) & \rightarrow \top
\end{align*}
\]
Three strikes, out

- Aim: account suspended after 3 unsuccessful login attempts.
- In the program: \( ckpwd(g, PWD) \)
- For each operator \( f \) of arity \( n \) we consider \( f_{DIP} \) of arity \( 2n \).
  \[ \Rightarrow \] distinction between the name of a program variable and its privacy level.
- Privacy level of \( ckpwd(g, PWD) \) is computed by the evaluation of:
  \[
  ckpwd_{DIP}(\pi_g, g, \pi_{pwd}, PWD)
  \]
Three strikes, out

\[
\begin{array}{c}
t_0 \\
\downarrow \\
t_1 \\
\downarrow \\
t_2 \\
\downarrow \\
t_3 \\
\uparrow \\
\end{array}
\]

\[
\begin{align*}
\text{ckpwd}(\bot, \overline{g}, t_0, \overline{p}) & \rightarrow \bot; \overline{p} \rightarrow t_1 \\
\text{ckpwd}(\bot, \overline{g}, t_2, \overline{p}) & \rightarrow \bot; \overline{p} \rightarrow t_3 \\
\text{ckpwd}(\bot, \overline{g}, t_1, \overline{p}) & \rightarrow \bot; \overline{p} \rightarrow t_2 \\
\text{ckpwd}(\bot, (\overline{g}, t_3, \overline{p}) & \rightarrow \top \\
\text{ckok}(\overline{x}, \overline{y}) & \rightarrow \bot; \overline{y} \rightarrow t_0 \\
PIN_1 & \rightarrow t_0 \\
\end{align*}
\]

\[
\begin{align*}
g & \rightarrow \bot \\
PIN_2 & \rightarrow t_0 \\
\end{align*}
\]
Three strikes, out

In the program:

if ckpwd\( (g, PIN_1) \) then blah else next_try
Dynamic interference policy

Definition

A DIP, $\mathcal{SP}$, is a confluent terminating rewrite system with actions with:

1. For every $x \in V$ there is a rule $x \rightarrow \pi$ in $\mathcal{SP}$.
Dynamic interference policy

Definition

A DIP, $SP$, is a confluent terminating rewrite system with actions with:

1. For every $x \in V$ there is a rule $x \rightarrow \pi$ in $SP$.
2. For each rule $l \rightarrow r$ such that $l$ is in $V$ then $r$ is in $L$. 

F. Prost  Frederic.Prost@ens-lyon.fr  (Ecole Normale Supérieure de Lyon)  Computer Science and Privacy Non Interference  Analyzes  September 2015
Dynamic interference policy

Definition

A DIP, $\mathcal{SP}$, is a confluent terminating rewrite system with actions with:

1. For every $x \in \mathcal{V}$ there is a rule $x \rightarrow \pi$ in $\mathcal{SP}$.
2. For each rule $l \rightarrow r$ such that $l$ is in $\mathcal{V}$ then $r$ is in $\mathcal{L}$.
3. $\mathcal{SP}$ introduces no junk into $\mathcal{L}$. I.e., for all ground terms, $t$, over $\Sigma \cup \mathcal{L}$, the normal form of $t$, is in $\mathcal{L}$. 
Dynamic interference policy

Definition

A DIP, $\mathcal{SP}$, is a confluent terminating rewrite system with actions with:

1. For every $x \in V$ there is a rule $x \rightarrow \pi$ in $\mathcal{SP}$.
2. For each rule $l \rightarrow r$ such that $l$ is in $V$ then $r$ is in $\mathcal{L}$.
3. $\mathcal{SP}$ introduces no junk into $\mathcal{L}$. I.e., for all ground terms, $t$, over $\Sigma \cup \mathcal{L}$, the normal form of $t$, is in $\mathcal{L}$.
4. $\mathcal{SP}$ introduces no confusion into $\mathcal{L}$. I.e.,
   \[ \forall \tau_1, \tau_2 \in \mathcal{L}, \tau_1 \neq \tau_2 \implies \tau_1 \not\rightarrow^* \tau_2. \]
**Definition**

A DIP, $\mathcal{SP}$, is a confluent terminating rewrite system with actions with:

1. For every $x \in \mathcal{V}$ there is a rule $x \rightarrow \pi$ in $\mathcal{SP}$.
2. For each rule $l \rightarrow r$ such that $l$ is in $\mathcal{V}$ then $r$ is in $\mathcal{L}$.
3. $\mathcal{SP}$ introduces no junk into $\mathcal{L}$. I.e., for all ground terms, $t$, over $\Sigma \cup \mathcal{L}$, the normal form of $t$, is in $\mathcal{L}$.
4. $\mathcal{SP}$ introduces no confusion into $\mathcal{L}$. I.e.,
   $$\forall \tau_1, \tau_2 \in \mathcal{L}, \tau_1 \neq \tau_2 \implies \tau_1 \not\rightarrow^* \tau_2.$$
5. Functions in $\Sigma$ are monotonic w.r.t. privacy levels: $\forall \pi_i, \pi_i' \in \mathcal{L}, \pi_i \sqsubseteq \pi_i' \implies \text{nf}^{\mathcal{SP}}(f(\pi_1, \ldots, \pi_n)) \sqsubseteq \text{nf}^{\mathcal{SP}}(f(\pi_1', \ldots, \pi_n'))$.

$$\langle t, \mathcal{SP} \rangle \rightsquigarrow^* \langle \text{nf}^{\mathcal{SP}}(t), \overline{\mathcal{SP}}^t \rangle$$
Privacy level of a term wrt $SP$

- to compute the privacy level of $f(x, y)$ we consider

$$t = f_{DIP}(nf^{SP}(x), x, nf^{SP}(y), y)$$
Privacy level of a term wrt $\mathcal{SP}$

- to compute the privacy level of $f(x, y)$ we consider
  \[
  t = f_{DIP}(\text{nf}^{\mathcal{SP}}(x), x, \text{nf}^{\mathcal{SP}}(y), y)
  \]

- The evaluation of this term in $\mathcal{SP}$ gives the privacy level and a new interference policy: $\langle t, \mathcal{SP} \rangle \rightsquigarrow^{*} \langle \pi^{\mathcal{SP}}(t), \overline{\mathcal{SP}}^{t} \rangle$
Privacy level of a term wrt $SP$

- to compute the privacy level of $f(x, y)$ we consider
  $$t = f_{DIP}(nf^{SP}(x), x, nf^{SP}(y), y)$$

- The evaluation of this term in $SP$ gives the privacy level and a new interference policy: $\langle t, SP \rangle \rightsquigarrow^* \langle \pi^{SP}(t), \overline{SP}^t \rangle$

**Definition**

$\langle t, SP \rangle \rightsquigarrow \langle t', SP' \rangle$ iff there is a position $p$ and a substitution $\theta$ and a rewrite rule $l \rightarrow r$; a such that $\theta(l) = t|_p$, $\theta(l_i) \in V$ and $a = l_1 \mapsto r_1; \ldots; l_m \mapsto r_m$, then

- $t' = t[\theta(r)]_p$
- $SP' = SP \bullet \{\theta(l_i) \mid 1 \leq i \leq m\} \oplus \theta(l_1) \rightarrow nf^{SP}(\theta(r_1)) \oplus \ldots \oplus \theta(l_m) \rightarrow nf^{SP}(\theta(r_m))$
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
Program safety

- Traditionally: a program is safe if every modification of a value above \( \pi \) cannot be observed below \( \pi \):
  \[
  \langle \mu_1, P \rangle \xrightarrow{\mu_1'} \langle \mu_2, P \rangle \xrightarrow{\mu_2'} \\
  \mu_1' \equiv_\pi \mu_2'
  \]

- What to do with the policy:
  \[
  \text{encrypt} (\pi_{1024}, \top) \rightarrow \bot
  \]
  making possible program as:
  \[
  \text{SPY} := \text{encrypt}(\text{key}_{1024}, \text{PIN})
  \]
Program safety

- Traditionally: a program is safe if every modification of a value above $\pi$ cannot be observed below $\pi$:

$$\langle \mu_1, P \rangle \xrightarrow{\ast_{os}} \mu_1'$$
$$\langle \mu_2, P \rangle \xrightarrow{\ast_{os}} \mu_2'$$
$$\mu_1' \equiv_{\pi} \mu_2'$$

- What to do with the policy:

$$\text{encrypt}(\pi_{1024}, \top) \rightarrow \bot$$

making possible program as:

$$\text{SPY} := \text{encrypt}(\text{key}_{1024}, \text{PIN})$$

$$\implies$$ Use an alternate op. sem. declared leaks are treated specifically.

- Notion of declassified operational semantics.

$$\langle \mu_1, P \rangle \xrightarrow{\mu_d} \langle \mu'_1, P' \rangle$$
Memory equivalences

Definition

\( \mu, \mu' \) are equivalent up to security level \( \pi \) and security policy \( SP \) if for all \( x \in \mathcal{V} \) such that \( \pi^{SP}(x) \sqsubseteq \pi \) then \( \mu(x) = \mu'(x) \). We write \( \mu \sim_{\pi}^{SP} \mu' \)
Memory equivalences

Definition

\( \mu, \mu' \) are equivalent up to security level \( \pi \) and security policy \( S\mathcal{P} \) if for all \( x \in V \) such that \( \pi^{S\mathcal{P}}(x) \sqsubseteq \pi \) then \( \mu(x) = \mu'(x) \). We write \( \mu \simeq_{\pi}^{S\mathcal{P}} \mu' \)

Definition

Let \( S\mathcal{P}, S\mathcal{P}' \) be two DIPs such that \( \{S\mathcal{P}\} = \{S\mathcal{P}'\} \) and \( \text{Dom}(S\mathcal{P}) = \text{Dom}(S\mathcal{P}') \). We define the DIP \( S\mathcal{P} \sqcup S\mathcal{P}' \) by:

\[
|S\mathcal{P} \sqcup S\mathcal{P}'| = \{ x \rightarrow \pi'' \mid x \rightarrow \pi \in |S\mathcal{P}| \land x \rightarrow \pi' \in |S\mathcal{P}'| \land \pi' \sqcup \pi = \pi'' \}
\]

\[
\{S\mathcal{P} \sqcup S\mathcal{P}'\} = \{S\mathcal{P}\}
\]
declassifying terms

- A term is declassifying if its privacy level is lower than one of its arguments.
- Such terms will be subjected to specific rules in the declassified operational semantics.

Definition (Declassifying terms and assignments)

\[ t = f(x_1, \ldots, x_n) \text{ is declassifying wrt } S\mathcal{P}, \text{ written } S\mathcal{P} \vdash f(x_1, \ldots, x_n) \downarrow \text{ if: } \]
\[ \pi^{S\mathcal{P}}(t) \sqsubseteq \left( \bigsqcup_{i=1}^{n} \pi^{S\mathcal{P}}(t_i) \right) \]
Declassified evaluation

- \( \langle P, \mu, SP \rangle \xrightarrow{\mu_d} \langle P', \mu', SP' \rangle \)

- Declassifying assignment:

\[
SP \vdash f_{DIP}((\pi^{SP}(x), x) \downarrow \llbracket f(x) \rrbracket_{\mu_d} = v) \quad \langle f(\pi^{SP}(x), x), SP \rangle \leadsto^* \langle \pi, \overline{SP} f(\pi^{SP}(x), x) \rangle
\]

\[
\langle y := f(x), \mu, SP \rangle \xrightarrow{\mu_d} \langle \text{skip}, \mu[y := v], \overline{SP} f(\pi^{SP}(x), x) \rangle
\]
High/low bisimulation and DIPs

Definition (Bisimulation)

A $\pi$-bisimulation is a symmetric relation $\mathcal{R}$ such that:

$$\langle P_1, S\mathcal{P}_1 \rangle \mathcal{R} \langle P_2, S\mathcal{P}_2 \rangle$$

$$\langle \mu_1, P_1, S\mathcal{P}_1 \rangle \xrightarrow{\mu_1}$$

$$\langle \mu'_1, P'_1, S\mathcal{P}'_1 \rangle$$

$$\mu_1 \sim_{\pi S\mathcal{P}_1 \sqcup S\mathcal{P}_2} \mu_2$$
Defintion (Bisimulation)

A $\pi$-bisimulation is a symmetric relation $R$ such that:

$$\langle P_1, SP_1 \rangle R \langle P_2, SP_2 \rangle$$

$$\langle \mu_1, P_1, SP_1 \rangle \xrightarrow{\mu_1} \langle \mu'_1, P'_1, SP'_1 \rangle$$

$$\mu_1 \sim_{\pi} SP'_1 \sqcup SP'_2 \mu_2$$

$$\exists P'_2, SP'_2 \text{ and } \mu'_2 \text{ s.t. }$$

$$\langle \mu_2, P_2, SP_2 \rangle \xrightarrow{\mu_1}^{*} \langle \mu'_2, P'_2, SP'_2 \rangle$$

and $\mu'_1 \sim_{\pi} SP'_1 \sqcup SP'_2 \mu'_2$

and $\langle P'_1, SP'_1 \rangle R \langle P'_2, SP'_2 \rangle$
The union of two $\pi$-bisimulation is a $\pi$-bisimulation.
The biggest $\pi$-bisimulation is written $\simeq$ and is the union of all $\pi$-bisimulation.
$\pi$-bisimulations are not reflexive in general...
Relaxing Non-Interference

Program safety w.r.t. DIP

- The union of two $\pi$-bisimulation is a $\pi$-bisimulation.
- The biggest $\pi$-bisimulation is written $\simeq$ and is the union of all $\pi$-bisimulation.
- $\pi$-bisimulations are not reflexive in general...

$\langle SPY := PIN, \{PIN \rightarrow T, SPY \rightarrow \bot\} \rangle$ is not $\bot$-bisimilar to itself!
Program safety w.r.t. a DIP

- The union of two $\pi$-bisimulation is a $\pi$-bisimulation.
- The biggest $\pi$-bisimulation is written $\simeq$ and is the union of all $\pi$-bisimulation.
- $\pi$-bisimulations are not reflexive in general...

$$\langle \textit{SPY} := \textit{PIN}, \{ \textit{PIN} \rightarrow \top, \textit{SPY} \rightarrow \bot \} \rangle$$ is not $\bot$-bisimilar to itself!

**Definition (Safe program)**

A program $P$ is safe with relation DIP $\mathcal{SP}$, written $\mathcal{SP} \models P$, if for all privacy level $\pi$ $\langle P, \mathcal{SP} \rangle \simeq^{\pi} \langle P, \mathcal{SP} \rangle$. 
Plan

1 NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2 NI in an Imperative Setting

3 NI and concurrency

4 Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5 Conclusion
Abstract execution principle (1)
Abstract execution principle (1)

- Idea: to execute the program on $\mathcal{L}$.
- An abstract memory record associates variables with their privacy levels.
Abstract execution principle (1)

- Idea: to execute the program on $L$.
- An abstract memory record associates variables with their privacy levels.
- Record of the highest privacy level encountered in if-then-else and while guards to avoid indirect leaks, e.g.:

  $$\text{if } \text{PIN} = 0 \text{ then while } 0 \text{ do skip else skip; } SPY := 0$$

- Check assignments wrt $SP$ and :

  $$x := f(\ldots) \text{ implies } \pi^{SP}(x) \subseteq (\pi^{SP}(f(\ldots)) \sqcup \pi_g)$$

  raises a failure if the inequality is not satisfied.
Abstract execution principle (2)

- Moreover evaluation of terms modify the DIP.
- Problem: it is not possible to merge DIPs resulting from the branches of an if-then-else construct.
Abstract execution principle (2)

- Moreover evaluation of terms modify the DIP.
- Problem: it is not possible to merge DIPs resulting from the branches of an if-then-else construct.
  \[ \Rightarrow \] Creation of a DIP list recording DIP’s for each execution paths.
- Fixpoint problem for the while construct.
Abstract execution principle (2)

- Moreover evaluation of terms modify the DIP.
- Problem: it is not possible to merge DIPs resulting from the branches of an if-then-else construct.
  \[ \Rightarrow \text{Creation of a DIP list recording DIP's for each execution paths.} \]
- Fixpoint problem for the while construct.
  \[ \Rightarrow \text{Finite number of DIP lists.} \]
Abstract execution principle (2)

- Moreover evaluation of terms modify the DIP.
- Problem: it is not possible to merge DIPs resulting from the branches of an if-then-else construct.
  \[\Rightarrow\]  Creation of a DIP list recording DIP’s for each execution paths.
- Fixpoint problem for the while construct.
  \[\Rightarrow\]  Finite number of DIP lists.
- The abstract operational semantics is defined as a reduction on tuples made on set of couples \(\langle SP, \pi_1, \pi_2\rangle\) and programs.
- The result of an abstract execution is either a list of DIP or ♠.
Abstract Execution definition

\( F_i(\mathcal{L}, P, t) \) stands for the following formula:

\[
\{ \langle SP', (\pi'_1 \cap \pi^{SP}(t)), \pi'_2 \rangle \mid \forall \langle SP, \pi_1, \pi_2 \rangle \in \mathcal{L}. \\
\langle \{ \langle SP, (\pi_1 \sqcup \pi^{SP}(t)), \pi_2 \rangle \}, P \rangle \hookrightarrow \mathcal{L}'_{\mathcal{N}} \wedge \\
\langle SP', \pi'_1, \pi'_2 \rangle \in \mathcal{L}'_{\mathcal{N}} \}
\]

\( F_w(\mathcal{L}, P, t) \) stands for the following formula:

\[
\{ \langle SP', (\pi'_1 \cap \pi^{SP}(t)), \pi'_2 \rangle \mid \forall \langle SP, \pi_1, \pi_2 \rangle \in \mathcal{L}. \\
\langle \{ \langle SP, (\pi_1 \sqcup \pi^{SP}(t)), (\pi_2 \sqcup \pi^{SP}(t)) \rangle \}, P \rangle \hookrightarrow \mathcal{L}'_{\mathcal{N}} \wedge \\
\langle SP', \pi'_1, \pi'_2 \rangle \in \mathcal{L}'_{\mathcal{N}} \}
\]

\( G(\mathcal{L}, t) \) stands for the following formula:

\[
\forall \langle SP, \pi \rangle \in \mathcal{L}. \forall y \in b \downarrow SP. \pi \subseteq \pi^{SP}(y)
\]

F. Prost  Frederic.Prost@ens-lyon.fr (Ecole Normale Supérieure de Lyon)  Computer Science and Privacy Non Interference  September 2015 69 / 78
Some rules

\[
\neg G(\mathcal{L}, b) \quad \frac{}{\langle \mathcal{L}, \text{if } b \text{ then } P_1 \text{ else } P_2 \rangle \mapsto \spadesuit} \quad \text{IFF} \quad \mapsto
\]

\[
\mathcal{F}_w(\mathcal{L}, P, b) \not\subseteq \mathcal{L} \quad \frac{}{\langle \mathcal{L} \cup \mathcal{F}_w(\mathcal{L}, P, b), \text{while } b \text{ do } P \rangle \mapsto \mathcal{L}' \quad G(\mathcal{S}P, b) \quad \text{WX} \quad \mapsto
\]

\[
\langle \mathcal{L}, \text{while } b \text{ do } P, \pi \rangle \mapsto \mathcal{L}'
\]

\[
\mathcal{F}_w(\mathcal{L}, P, b) \subseteq \mathcal{L} \quad \frac{}{G(\mathcal{S}P, b) \quad \text{WT} \quad \mapsto}
\]

\[
\langle \mathcal{L}, \text{while } b \text{ do } P \rangle \mapsto \mathcal{L}
\]

\[
\neg G(\mathcal{S}P, b) \quad \frac{}{\langle \mathcal{L}, \text{while } b \text{ do } P \rangle \mapsto \spadesuit} \quad \text{WF} \quad \mapsto
\]

\[
\neg G(\mathcal{S}P, b)
\]
Abstract execution results

Théorème

Let $P$ be a program, and $S\mathcal{P}$ a DIP, then $\langle\{\langle S\mathcal{P}, \bot \rangle\}, P \rangle \mapsto^* \spadesuit$ or there exists $L$ such that $\langle\{\langle S\mathcal{P}, \bot \rangle\}, P \rangle \mapsto^* L$. 
Abstract execution results

Théorème

Let \( P \) be a program, and \( S\mathcal{P} \) a DIP, then \( \langle \{\langle S\mathcal{P}, \bot \rangle \}, P \rangle \hookrightarrow^* \spadesuit \) or there exists \( \mathcal{L} \) such that \( \langle \{\langle S\mathcal{P}, \bot \rangle \}, P \rangle \hookrightarrow^* \mathcal{L} \).

Theorem

\[ \exists \mathcal{L}. (\langle \{\langle S\mathcal{P}, \bot \rangle \}, P \rangle \hookrightarrow^* \mathcal{L}) \implies S\mathcal{P} \models P \]
Abstract execution results

Théorème

Let $P$ be a program, and $SP$ a DIP, then $\langle \{ \langle SP, \bot \rangle \}, P \rangle \hookrightarrow^* \spadesuit$ or there exists $L$ such that $\langle \{ \langle SP, \bot \rangle \}, P \rangle \hookrightarrow^* L$.

Theorem

\[ \exists L. (\langle \{ \langle SP, \bot \rangle \}, P \rangle \hookrightarrow^* L) \implies SP \models P \]

- Converse implication does not hold:

  if $PIN = 0$ then $SPY := 1$ else $SPY := 1$

  this safe program raises a failure in the abstract operational semantics.
Plan

1. NI in a Purely Functional Setting
   - Pure Terms and Simple Types
   - Higher-order Types

2. NI in an Imperative Setting

3. NI and concurrency

4. Relaxing Non-Interference
   - Programming framework
   - Dynamic interference policy
   - Program safety w.r.t. DIP
   - Program Verification

5. Conclusion
Non-Interference is a very abstract and powerful, but strict, approach to privacy in programming languages.

It is very different from the traditional cryptographic approach and relies on completely different techniques: programming semantics.

There has been a lot of work in order to cope with different paradigms and subtle variations around the notion of strict non-interference.

Differential privacy is a relatively new way to approach non-interference. In a nutshell: the idea is to manipulate data of a data-base in such a way that statistical properties of interest are unchanged while having indistinguishability properties (kind of non-interference) insuring the privacy (e.g. [Dwork, 2008]).


Pruning simply typed lambda-terms.  

The mathematical language AUTOMATH, its usage and some of its extensions (iria, versailles 1968).  

Church, A. (1940).  
A formulation of the simple theory of types.  
*Journal of Symbolic Logic*, 5(1).


*Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur.* 

Constructive natural deduction and its modest interpretation. 

A static calculus of dependencies for the lambda-cube.  
Bibliography V

Sort abstraction for static analyzes of mobile processes.

Enforcing dynamic interference policy.

Declassification: Dimensions and principles.