Modular Information Flow Analysis for Process Calculi

Sylvain CONCHON

OGI School of Science and Engineering

Oregon Health and Science university

Noninterference for process calculi

Information flow analysis is used to guarantee secrecy and integrity of information.

Traditionally, information is assigned a security level, say high (secret information) and low (public information), and the goal of this analysis is to prove the **noninterference** property [Goguen-Meseguer, 1982]:

> High-level information never interferes with low-level one, or similarly, low-level information does not depend on high-level one.

In the setting of **process calculi**, this independence property is meant to model the absence of information flow from the secret components of a process to its (public) interactions with the environment.

Type-based information flow analysis

Recently, this problem has been reformulated as a typing problem, and a number of type systems have been developed to ensure the noninterference property.

In a type-based information flow analysis, security levels are formalized as types and new type systems, combining **information-flow-specific** and **standard type-theoretic** aspects, are designed to guarantee the noninterfence property.

In these systems, the nonintefeerence property guarantees secure information flow for well-typed processes. It is expressed as a **soundness** result based on a notion of **process equivalence**:

> Two well-typed processes with distinct high-level components but common low-level structure cannot be distinguished by an observer.

A modular proof technique

Our goal is to **extend** existing type systems of process calculi with security annotations to ensure the noninterference property.

However,

- we **don't want** to re-establish the proof of the *standard part* of the original system which may support for instance polymorphism and type reconstruction.
- We **want** to prove the correctness of these extended systems, including noninterference, with a minimal proof effort.

We propose a modular approach which consists in three steps:

1. **Annote** terms with security labels, and let the semantics propagate them dynamically: a dynamic noninterference result shows that labelled are propagated in a meaningful way;
2. **Implement** this instrumented semantics by translating the labelled calculus in the standard one: a simulation result shows the correctness of the translation;
3. **Choose** a standard type system and use it to analyze the terms output by the translation: a static noninterference result is derived from steps 1. and 2.

An example: the Join-Calculus

We formalize our framework within the **Join-Calculus**, a model of concurrent and distributed systems based on channel abstraction (communication) and join-patterns (synchronization), which enjoys a Hindley/Milner typing discipline with a family of rich polymorphic constraint-based type systems with type reconstruction.

Syntax:

$$P ::= 0 \mid (P \mid P) \mid u\langle \vec{v} \rangle \mid \texttt{def}\ D\ \texttt{in}\ P$$

$$D ::= J \triangleright P \mid D\ \texttt{or}\ D$$

$$J ::= u\langle \vec{x} \rangle \mid (J \mid J)$$

Semantics:

$$\begin{array}{lcl} & & \texttt{def}\ \ldots \\ \texttt{def}\ a\langle x \rangle \mid b\langle y \rangle \triangleright y\langle x \rangle & \nearrow & \texttt{in}\ u\langle v \rangle \\ \quad \texttt{or}\ a\langle x \rangle \triangleright \mathrm{out}\langle x \rangle & & \\ \texttt{in}\ a\langle v \rangle \mid b\langle u \rangle & \searrow & \texttt{def}\ \ldots \\ & & \texttt{in}\ \mathrm{out}\langle v \rangle \mid b\langle u \rangle \end{array}$$

The Join-Calculus (cont'd)

Formally, the operational semantics of the Join-Calculus is defined using a *standard* reduction semantics with evaluation contexts.

Contexts:

$$\mathcal{R} ::= [\ ] \mid (P \mid \mathcal{R}) \mid (\mathcal{R} \mid P) \mid \texttt{def}\ D\ \texttt{in}\ \mathcal{R}$$
$$\mathcal{D} ::= [\ ] \mid D\ \texttt{or}\ \mathcal{D} \mid \mathcal{D}\ \texttt{or}\ D$$

Operational semantics:

$$\texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[J\sigma] \rightarrow \texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[P\sigma]$$

What does "P and Q are indistinguishable" mean ?

Our results are based on the weak-barbed congruence (WB-congruence), which is defined in a standard way:

- Define a **basic notion of observation** which detects the ability of a process to interact with its environment. The strong barb predicate $P \downarrow_u$ (P is directly observable at u) is defined if P emits a message on some free name u.
- Abstract from **internal reduction**. The weak predicate $P \Downarrow_u$ is defined if there exists a process Q such that $P \to^* Q$ and $Q \downarrow_u$.
- The **WB-bisimulation** is defined such that $P \stackrel{\bullet}{\approx} Q$ implies:
  1. $P \Downarrow_u$ if and only if $Q \Downarrow_u$.
  2. If $P \to P'$ then $\exists Q'$, $Q \to^* Q'$ and $P' \stackrel{\bullet}{\approx} Q'$
  3. If $Q \to Q'$ then $\exists P'$, $P \to^* P'$ and $Q' \stackrel{\bullet}{\approx} P'$
- Finally, this equivalence is refined with a congruence property to define the **WB-congruence**:

$$P \approx Q \stackrel{\text{def}}{=} \forall \mathcal{R}.\ \mathcal{R}[P] \stackrel{\bullet}{\approx} \mathcal{R}[Q]$$

Step 1. Do It Dynamically

A labelled Join-Calculus

We enrich the syntax of the Join-Calculus with labelled messages. For simplicity, we assume a fixed lattice of labels $\mathcal{L} = (\{\mathbf{L}, \mathbf{H}\}, \mathbf{L} \leq \mathbf{H})$.

$$P ::= \ldots \mid \alpha : u \langle \vec{v} \rangle \qquad \alpha \in \mathcal{L}$$

The basic reduction step $\rightarrow$ is extended to propagate the label α of the parallel composition of labelled messages $(\alpha \bullet J\sigma)$ to the guarded process P of J.

$$\texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[\alpha \bullet J\sigma] \rightarrow \texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[\alpha \bullet P\sigma]$$

We add a second rule $\mapsto$ to set the labels α_i of messages which match a join-pattern J, up to the same label $\alpha = \bigsqcup_{i \in I} \alpha_i$.

$$\texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[\prod_{i \in I} (\alpha_i : u_i \langle \vec{v}_i \rangle)] \mapsto \texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[\prod_{i \in I} (\alpha : u_i \langle \vec{v}_i \rangle)]$$

Noninterference up to WB-bisimulation

We prove the *well-foundedness* of our propagation rules by showing a noninterference result which relies on an equality between **unlabelled processes**.

An *erasure* function is generated by $\lfloor \mathbf{H} : u \langle \vec{v} \rangle \rfloor = 0$ and a *strip* function is generated by $\mathrm{strip}(\alpha : u \langle \vec{v} \rangle) = u \langle \vec{v} \rangle$.

We extend our observation predicates to detect the security levels of barbs, and we say that a processus is public if all its barbs are annotated with $\mathbf{L}$.

Two public processes which differ only in high-level messages are WB-bisimilar.

Dynamic noninterference. if P and Q are public and $\lfloor P \rfloor = \lfloor Q \rfloor$ then $\mathrm{strip}(P) \stackrel{\bullet}{\approx} \mathrm{strip}(Q)$.

Step 2. Translation

Our target calculus

The second step of our framework consists to define an *implementation* of our instrumented semantics as a **translation** from the labelled calculus to the standard one.

The translation requires some *extra constructs* in the target calculus. The set of names is extended with label constants that represent security labels, and a primitive, @, is added to concatenate labels.

$$u, v ::= n \mid \alpha \mid u@v \qquad \alpha \in \mathcal{L}$$

We extend the operational semantics of the Join-Calculus with a δ-rule which states that the primitive @ returns the **least upper bound** of its arguments.

$$\alpha@\beta \rightarrow \alpha \sqcup \beta \qquad \alpha, \beta \in \mathcal{L}$$

From the labelled calculus to the unlabelled one

The basic idea of our encoding is to use the **polyadicity** of the Join-Calculus to map every labelled message to an unlabelled one.

$$[\![\alpha : u\,\langle\,\vec{v}\,\rangle]\!] = u\,\langle\,\alpha, \vec{v}\,\rangle$$

Following this idea, every join-pattern is mapped to a pair of join-patterns defining the same channels with an extra argument representing the security level.

$$\left[\!\!\left[\prod_{i\in 1..n} (u_i\,\langle\,\vec{x_i}\,\rangle) \triangleright P\right]\!\!\right] = \begin{cases} \displaystyle\prod_{i\in 1..n} (u_i\,\langle\,\alpha, \vec{x_i}\,\rangle) \triangleright \alpha \star [\![P]\!] \\ \texttt{or} \\ \displaystyle\prod_{i\in 1..n} (u_i\,\langle\,\alpha_i, \vec{x_i}\,\rangle) \triangleright \prod_{i\in 1..n}\left(u_i\left\langle \mathop{@}_{i\in 1..n} \alpha_i\;,\; \vec{x_i}\right\rangle\right) \end{cases}$$

The first join-pattern mimics the extended reduction $\rightarrow$ by propagating the label α into $\alpha \star [\![P]\!]$. The second mimics the reduction $\mapsto$ by promoting the security arguments α_i of each message send on u_i to the concatenation of all security levels.

Simulation. If $P(\rightarrow \cup \mapsto)^* Q$ then $[\![P]\!] \rightarrow^* [\![Q]\!]$

Step 3. Chose a Static Approximation

A simply-typed Join-Calculus with subtyping

We illustrate our approach with a simple type system with subtyping, whose types $t \in T$ are ordered by $\preccurlyeq$.

$$\textsc{Msg}\quad \frac{\Gamma \vdash u : \langle \vec{t} \rangle \qquad \Gamma \vdash \vec{v} : \vec{t}}{\Gamma \vdash u \langle \vec{v} \rangle} \qquad\qquad \textsc{SubName}\quad \frac{\Gamma \vdash u : t' \qquad t' \preccurlyeq t}{\Gamma \vdash u : t}$$

$$\textsc{Join}\quad \frac{\Gamma + \Delta \vdash P \qquad \Delta(\vec{x_i}) = \vec{t_i}}{\Gamma \vdash \prod_{i \in I} (u_i \langle \vec{x_i} \rangle) \triangleright P :: (u_i : \langle \vec{t_i} \rangle)^{i \in I}}$$

This simple system must also be able to deal with the label constants and the primitive @.

$$\mathcal{L} \subseteq T \qquad\qquad \frac{\alpha \leq \beta}{\alpha \preccurlyeq \beta} \qquad\qquad \textsc{Label}\quad \Gamma \vdash \alpha : \alpha \qquad\qquad \textsc{At}\quad \frac{\Gamma \vdash u : \alpha \qquad \Gamma \vdash v : \beta}{\Gamma \vdash u@v : (\alpha \sqcup \beta)}$$

Putting it all together

By **composing** our encoding with this simple type system, we define a type system for our labelled calculus such that a labelled process is well-typed if and only if its translation is well-typed:

$$\Gamma \vdash P \text{ holds if and only if } \Gamma \vdash [\![P]\!]$$

The soundness proof of this system only relies on the soundness property of the original system and the simulation property of the translation

This type system enjoys a noninterference result based on **WB-congruence**, provided we admit only *well-typed contexts*. If $\mathbf{L}$ is the only security level which appears in the typing environment Γ, then

Static noninterference. If $\lfloor P \rfloor = \lfloor Q \rfloor$ and $\Gamma \vdash P$ and $\Gamma \vdash Q$ then $\mathrm{strip}(P) \approx \mathrm{strip}(Q)$.

The proof only relies on the dynamic noninterference property of the labelled Join-Calculus and the soundness of the type system

Conclusion

Conclusion

I have presented a 3-steps type-based information flow analysis for process calculi. By giving a semantic-based model of information-flow and by translating this particular model into a standard semantics:

- we provide a fair degree of **modularity** which facilitates the correctness proofs;
- we prove a dynamic noninterference result once and we obtain a set of static noninterference results for a variety of type systems.

Down-side of the approach: Not all analysis can be decomposed in this way.

For instance, since process calculi are low-level programming languages, control is encoded through the use of communications which are often linear. It is crucial to take advantage of this information to improve the accuracy of our analysis.

However, is it possible to encode a linearity analysis dynamically ?