Access Control Mechanisms in Klaim

Rosario Pugliese

Dipartimento di Sistemi e Informatica
Università di Firenze
e-mail: `pugliese@dsi.unifi.it`

Outline of the talk

– Motivations

– Types for Access Control

– Syntax of *Secure* KLAIM

– The type system:

type equality & canonical forms,

subtyping, type inference

– Well-typed nets

– Secure KLAIM operational semantics

– Main results

Subject reduction, run-time errors & type safety

– Future work

Security Issues

There can be *attacks* to

- **Communication channels**
  - passive (e.g. traffic analysis)
  - active (e.g. message modifications/forging)
- **Hosts**
  - modification of host resources and data
  - denial of service
- **Mobile Agents**
  - modification of agent code
  - leak of sensible data

Typical defences

Cryptography, Access Control, Activity Monitoring, ...

Aim

To exploit security tools at the level of the programming language

- Type systems have been successfully used to ensure *type safety* of programs since a long time

  type safety: there will not be *run-time errors*, e.g. data will be used consistently with their declaration

- In the last few years, some work has been made on exploring and designing type systems for security

  e.g. well-typed Java programs (and the corresponding verified bytecode) will never compromise the integrity of certain data

  e.g. type systems for the Dπ-calculus (Hennessy-Riely, Yoshida-Hennessy), and for the Ambient calculus (Cardelli-Ghelli-Gordon)

Types for Access Control

- *Models for Access Control*
  - mechanisms to *specify* the policies for access control
  - mechanisms to *enforce* such policies

- KLAIM *Capability-based Type System*
  - *types* as specification of access policies
    - to express access rights of nodes with respect to other nodes of the net
    - to describe process intentions (read, write, exec., . . .) relatively to the different localities they are willing to interact with or they want to migrate to
  - (static and dynamic) *type checking* as enforcement of access policies
    - only intentions that match access rights are allowed

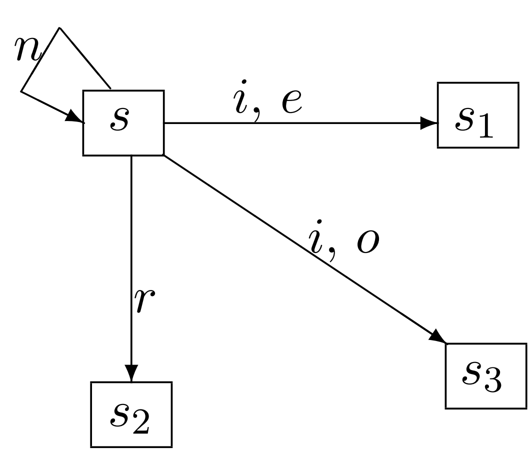
Example: Access Policy Specification

- *Capabilities:*
  r stands for **read**
  i stands for **in**
  o stands for **out**
  e stands for **eval**
  n stands for **newloc**

- The access policy specification δ_s of node s

$$
\begin{aligned}
\delta_s \quad = \quad & s : \{n\} \mapsto \bot, \\
& s1 : \{i, e\} \mapsto \delta_{\mathtt{s_1}}, \\
& s2 : \{r\} \mapsto \bot, \\
& s3 : \{i, o\} \mapsto \bot
\end{aligned}
$$

- A graphical interpretation: *access types are graphs*



Syntax of Secure Klaim

Types

$$
\begin{array}{lll}
\delta & ::= & \bot & (\textit{empty type}) \\
& \mid & \top & (\textit{universal type}) \\
& \mid & \ell : \pi \mapsto \delta & (\textit{locality-labelled arrow type}) \\
& \mid & \delta_1, \delta_2 & (\textit{union type}) \\
& \mid & \nu & (\textit{type variable}) \\
& \mid & \mu\nu.\delta & (\textit{recursive type})
\end{array}
$$

$\pi \subseteq \{r, i, o, e, n\} \quad (\pi \neq \emptyset)$ set of *capabilities*

Syntax of Secure KLAIM

$$
\begin{array}{llll}
\textsf{Nets} & N & ::= & s ::^{\delta}_{\rho} P \quad | \quad N_1 \parallel N_2 \\[2ex]
\textsf{Processes} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad X \\
 & & | & A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e} \rangle \\[2ex]
 & \multicolumn{3}{l}{(\mathit{Definition} \;\; A(\widetilde{X : \delta}, u : \widetilde{\langle \lambda, \delta \rangle}, \widetilde{x}) \stackrel{def}{=} P)} \\[2ex]
\textsf{Actions} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{in}(t)@\ell \quad | \quad \mathbf{read}(t)@\ell \\
 & & | & \mathbf{eval}(P)@\ell \quad | \quad \mathbf{newloc}(u : \widetilde{\langle \lambda, \delta \rangle}) \\[4ex]
\textsf{AccLists} & \lambda & ::= & [\ell_1 : \pi_1, \ldots, \ell_n : \pi_n] \\[2ex]
\textsf{Tuples} & t & ::= & f \mid f, t \\[2ex]
\textsf{Fields} & f & ::= & V \mid x \mid X \mid u : \langle \lambda, \delta \rangle \mid !Z \\[2ex]
\textsf{Values} & V & ::= & v \mid P \mid s : \langle \lambda, \delta \rangle \\[2ex]
\textsf{Variables} & Z & ::= & x \mid X : \delta \mid u : \langle \lambda, \delta \rangle
\end{array}
$$

Type Equality ($\cong$) & Canonical Forms

$$\ell : \pi \mapsto \delta = \ell : \pi \mapsto \bot \qquad \text{if } e \notin \pi$$

$$\ell : \pi_1 \mapsto \delta_1, \ell : \pi_2 \mapsto \delta_2 = \begin{cases} \ell : \pi_1 \cup \pi_2 \mapsto \delta_2 & \text{if } e \notin \pi_1 \\ \ell : \pi_1 \cup \pi_2 \mapsto \delta_1 & \text{if } e \notin \pi_2 \\ \ell : \pi_1 \cup \pi_2 \mapsto (\delta_1, \delta_2) & \text{otherwise} \end{cases}$$

$$\mu\nu.\nu = \bot \qquad (\textit{divergence})$$

$$\delta[\mu\nu.\delta/\nu] = \mu\nu.\delta \qquad (\textit{folding/unfolding})$$

Canonical Forms

$$\delta ::= \bot \mid \top \mid \phi_1, \ldots, \phi_n \mid \mu\nu.(\phi_1, \ldots, \phi_n) \qquad (n \geq 1)$$

$$\phi ::= \nu \mid \ell : \pi \mapsto \delta$$

Some Results

- $\cong$ is decidable
- For any type δ there is a canonical form δ' such that $\delta \cong \delta'$

Subtyping

Types have a hierarchical structure, the *subtype* relation $\preceq$, induced by an *ordering* relation over capabilities $\sqsubseteq_\Pi$

- $\{i\} \sqsubseteq_\Pi \{r\}$ $\qquad\qquad$ $\pi_2 \sqsubseteq_\Pi \pi_1$ if $\pi_1 \subseteq \pi_2$

- *Selection of subtyping rules*:

$$\frac{\pi_2 \sqsubseteq_\Pi \pi_1, \quad \delta_1 \preceq \delta_2}{\ell : \pi_1 \mapsto \delta_1 \preceq \ell : \pi_2 \mapsto \delta_2} \qquad \text{(standard on arrow types)}$$

E.g. $s_1 : \{r\} \mapsto \perp \preceq s_1 : \{i\} \mapsto \perp$

$$\delta_1 \preceq \delta_1, \delta_2 \qquad \text{(monotonicity on union types)}$$

Main Result $\quad \preceq$ is decidable

Type Inference

Selection of type inference rules

$$\frac{\gamma \vdash_{\ell} P : \delta}{\gamma \vdash_{\ell} \mathbf{out}(t)@\ell'.P : (\delta, [\![\, \ell' \,]\!]_{\ell} : \{o\} \mapsto \bot)}$$

$$\frac{upd_{\ell}(\gamma, t) \vdash_{\ell} P : \delta \qquad upd_{\ell}(\gamma, t) \vdash_{\ell} \delta\searrow_{lv(t)} = \delta'}{\gamma \vdash_{\ell} \mathbf{in}(t)@\ell'.P : (\delta', [\![\, \ell' \,]\!]_{\ell} : \{i\} \mapsto \bot)}$$

$$\frac{\gamma \vdash_{\ell} P : \delta \qquad \gamma \vdash_{[\![\, \ell' \,]\!]_{\ell}} Q : \delta'}{\gamma \vdash_{\ell} \mathbf{eval}(Q)@\ell'.P : (\delta, [\![\, \ell' \,]\!]_{\ell} : \{e\} \mapsto \delta')}$$

$\gamma \vdash_{\ell} P : \delta$ means that *within the type context* γ*, the intentions of* P *when located at* ℓ *are those specified in* δ

$$[\![\, \ell' \,]\!]_{\ell} = \begin{cases} \ell & \text{if } \ell' = \texttt{self} \\ \ell' & \text{otherwise} \end{cases}$$

Type Inference: main results

Minimal Type

If $\gamma \vdash_{\ell} P : \delta'$ then there exists a *minimal* type δ such that

$\gamma \vdash_{\ell} P : \delta$ and $\delta \preceq \delta''$ for all δ'' such that $\gamma \vdash_{\ell} P : \delta''$

Decidability

For any process P, the existence of a type δ such that

$\phi \vdash_{\ell} P : \delta$ is decidable

Well-typed Nets

Type interpretation

- Process types associate locality variables and sites to functions from sets of capabilities to process types

- Node types (access policies) associate sites to functions from sets of capabilities to node types

- To compare process types and node types, locality variables have to be *interpreted*, i.e. replaced by sites, by using site allocation environments

Well-typed Nets

- A net N_S is *well–typed* if for any node $s ::_{\rho_s}^{\delta_s} P$, there exists δ' such that $\phi \vdash_s P : \delta'$ and if δ is a minimal type for P then $[\![\, \delta \,]\!]_s^{\Theta_{N_S}} \preceq \delta_s$.

Operational Semantics

The operational semantics of *secure* KLAIM differs from that of (untyped) KLAIM in two main aspects

- Pattern-matching has to take into account the (access) types of the fields of its argument tuples

A simple example

$$\begin{array}{rcl} Server & = & \mathbf{out}(P)@\mathtt{self}.\mathbf{nil} \\ Client & = & \mathbf{read}(!X : \delta)@u_s.X \end{array}$$

If Θ_{N_S} is the interpretation function of the net and δ_c is the type (i.e. access policy) of the site of *Client*, then

Static Type Checking:

$$[\![\, \delta \,]\!]_c^{\Theta_{N_S}} \preceq \delta_c$$

Dynamic Type Checking:

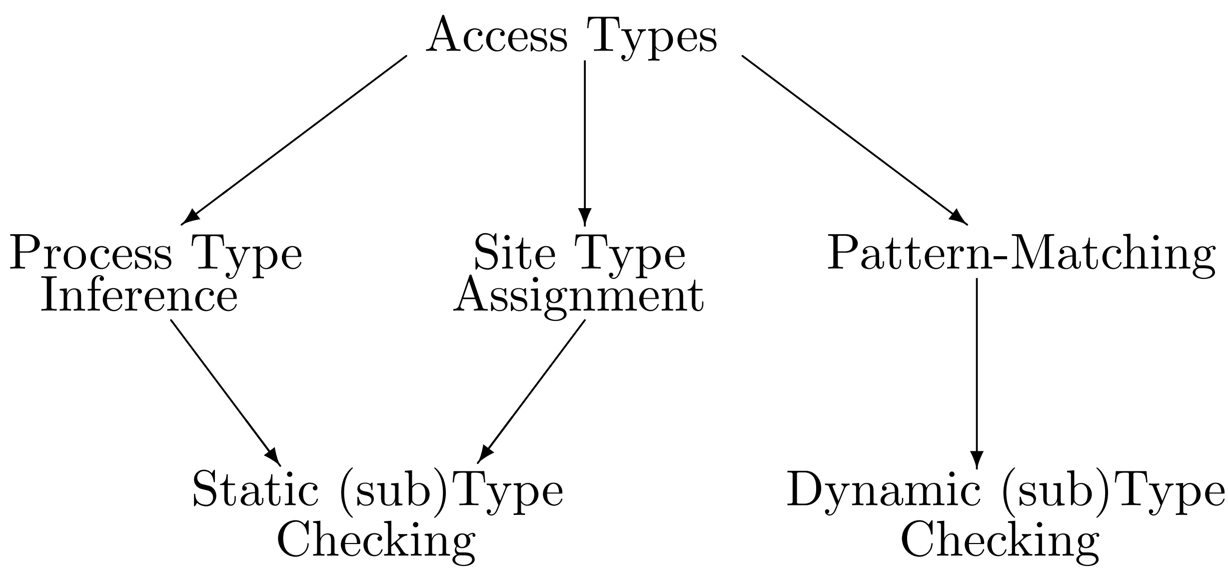
$$[\![\, \delta_P \,]\!]_c^{\Theta_{N_S}} \preceq [\![\, \delta \,]\!]_c^{\Theta_{N_S}}$$

hence

$$[\![\, \delta_P \,]\!]_c^{\Theta_{N_S}} \preceq \delta_c$$

- Node creation has to modify the (access) types of the nodes of the net in order to dynamically reconfigure the net

Ingredients



Main Results

Subject Reduction

If N is well–typed and $N \rightarrowtail N'$ then N' is well–typed

Run-time Errors

(e.g. $cap(\mathbf{read}(t)@\ell) = \{r\}$ and $loc(\mathbf{read}(t)@\ell) = \ell$)

$$\frac{cap(\delta, \rho(loc(a))) \not\sqsubseteq_{\Pi} cap(a)}{s ::_{\rho}^{\delta} a.P \xrightarrow{s} error}$$

$$\frac{N \xrightarrow{s} error}{N \parallel N' \xrightarrow{s} error}$$

$$\frac{N \equiv N' \qquad N' \xrightarrow{s} error}{N \xrightarrow{s} error}$$

Type Safety

If N is well–typed then there is no site s s.t. $N \xrightarrow{s} error$

If N is well–typed and $N \rightarrowtail^{*} N'$ then there is no site s s.t. $N' \xrightarrow{s} error$

Future Work

- Type system enrichment
  - Dynamic transmission of access rights
  - Behavioural/history dependent types
- Integration of other security mechanisms
  - secure communication and authentication
  - mobile agent protection
  - multilevel security (e.g. role-based access control)
- Extension to open systems
- Implementation of KLAIM security mechanisms (under progress)

Visit the KLAIM site:

`http://music.dsi.unifi.it/klaim.html`

KLAIM **bibliography**

- **Locality based Linda: programming with explicit localities.** R. De Nicola, G. Ferrari, R. Pugliese. *TAPSOFT'97*, LNCS 1214, 1997.

- **Coordinating Mobile Agents via Blackboards and Access Rights.** R. De Nicola, G. Ferrari, R. Pugliese. *COORDINATION'97*, LNCS 1282, 1997.

- **KLAIM: a Kernel Language for Agents Interaction and Mobility.** R. De Nicola, G. Ferrari, R. Pugliese. *IEEE Transactions on Software Engineering*, Vol.24(5), 1998.

- **Interactive Mobile Agents in XKlaim.** L. Bettini, R. De Nicola, G. Ferrari, R. Pugliese. *WETICE'98*, IEEE Society Press, 1998.

- **Types as Specifications of Access Policies.** R. De Nicola, G. Ferrari, R. Pugliese. *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, LNCS 1603, 1999.

- **Types for Access Control.** R. De Nicola, G. Ferrari, R. Pugliese, B. Venneri. *Theoretical Computer Science*, 240(1), 2000.

- **Structured Nets in Klaim.** L. Bettini, M. Loreti, R. Pugliese. *ACM SAC'2000*, ACM Press, 2000.

- **Programming Access Control: The KLAIM Experience.** R. De Nicola, G.-L. Ferrari, R. Pugliese. *CONCUR'00*, LNCS 1877, 2000.

- **A Modal Logic for Klaim.** R. De Nicola, M. Loreti. *AMAST'00*, LNCS 1816, 2000.

- **Klava: a Java Framework for Mobile Code.** L. Bettini, R. De Nicola, R. Pugliese. Draft, 2000.

Recursive Types

Recursive types are used for typing migrating recursive processes

- $P \stackrel{def}{=} \mathbf{read}(!\,x)@\mathtt{self}.\mathbf{out}(x)@l_{next}.\mathbf{eval}(P)@l_{next}.\mathbf{nil}$

  P first accesses the local tuple space to read a value, then put this value in the tuple space located at l_{next}, and, finally, migrates to l_{next}.

  The outcome of the first stage of typing analysis of P is the type

  $$\delta_P = \mathtt{self} \mapsto \{r\} \mapsto \bot, l_{next} \mapsto \{o, e\} \mapsto \delta_P$$

- Instead, the type of process

  $$Q \stackrel{def}{=} \mathbf{read}(!\,x)@\mathtt{self}.\mathbf{out}(x)@l_{next}.Q$$

  is

  $$\delta_Q = \mathtt{self} \mapsto \{r\} \mapsto \bot, l_{next} \mapsto \{o\} \mapsto \bot$$

Well-typed Nets

Type interpretation

- *Type interpretation function* of a net N_S, $\Theta_{N_S} : S \longrightarrow \mathcal{E}$:

  for all $s \in S$, $\Theta_{N_S}(s) = \rho_s$ if $s ::^{\delta_s}_{\rho_s} P \in N_S$, for some δ_s and P.

- *Interpretation* $[\![\delta]\!]^{\Theta}_s$ of δ at s by Θ:

  a canonical form of the type defined inductively as follows

  – $[\![\bot]\!]^{\Theta}_s = \bot \qquad [\![\top]\!]^{\Theta}_s = \top \qquad [\![\nu]\!]^{\Theta}_s = \nu$

  – $$[\![(\ell : \pi \mapsto \delta')]\!]^{\Theta}_s = \begin{cases} [\![\ell]\!]^{\rho_s} : \pi \mapsto [\![\delta']\!]^{\Theta}_{[\![\ell]\!]^{\Theta(s)}} & \text{if } [\![\ell]\!]^{\Theta(s)} \in S \\ \ell : \pi \mapsto \delta' & \text{otherwise} \end{cases}$$

  – $[\![(\delta_1, \delta_2)]\!]^{\Theta}_s = [\![\delta_1]\!]^{\Theta}_s, [\![\delta_2]\!]^{\Theta}_s$

  – $[\![(\mu\nu.\delta')]\!]^{\Theta}_s = \mu\nu.[\![\delta']\!]^{\Theta}_s$

Well-typed Nets

- A net N_S is *well–typed* if for any node $s ::^{\delta_s}_{\rho_s} P$, there exists δ' such that $\phi \vdash_s P : \delta'$ and if δ is a minimal type for P then $[\![\delta]\!]^{\Theta_{N_S}}_s \preceq \delta_s$.