Privacy and Access Control

Rocco De Nicola
Dip. Sistemi e Informatica
Università di Firenze

denicola@dsi.unifi.it

Outline

- Motivations
- Privacy in DistributedTuple Spaces:
  - Extending Linda with primitives for privacy in distributed and mobile applications
  - KriptoKlava
- Access Control via Typing
- μ-Klaim
- Types for μ-Klaim

Motivation

Process mobility poses to a lot of security problems

- secrecy and integrity of transmitted data and program code
- Malicious agents can attempt to access/modify private information of the nodes hosting them
- Malicious hosts can try to compromise agent's integrity/secrecy

Several programming/process languages with code mobility come equipped with security mechanisms (e.g. type systems, mechanisms for data/control flow analysis)

Attacks

- 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, … Types

Our Defenses

- Cryptography
- Types

Security Problems

- Linda provides no access protection to a tuple space
- No way to determine the issuer of an operation to the tuple space
- A process may retrieve/remove data that do not belong to it
- Shared data can be easily modified and corrupted

Our Proposal

- Extend Linda operations with cryptography:
  - Tuples can contain encrypted data
  - New Primitives for encryption/decryption
- Aims:
  - Change as little as possible the original Linda model
  - Make it Suitable for distributed application and mobile agent based application

Privacy, not Security

- Our principal aim is not to avoid that wrong data be retrieved
- Our aim is that even if data is eavesdropped or stolen, still it cannot be read
- A sort of PGP for Linda
- Smooth extension of Linda
  - The impact on the Linda model is minimal
  - Previous applications continue to work

Cryptography & Tuple Spaces

- Operations for inserting/retrieving encrypted tuples to/from tuple spaces (**ink**, **readk**, **outk**)
- Operations for encrypting/decrypting tuple contents (**encode** & **decode**)
- Operations for signing/verifying mobile agent code

ink & readk

1. look for and possibly retrieve a matching tuple,
2. attempt a decryption of the encrypted fields of the retrieved tuple
3. if the decryption fails:
    1. if the operation was an ink then put the retrieved tuple back in the tuple space,
    2. look for alternative matching tuples,
4. if all these attempts fail, then block until another matching tuple is available.

Extended Pattern Matching

- The original pattern matching has to be extended
- Two stages pattern matching:
  - In the first stage an encrypted field is seen as an ordinary field with the type “encrypted” and it can match only another “encrypted” field
  - In the second stage decryption takes place, and a further matching is performed with the decrypted clear-text fields

Keys and Mobile Agents

- symmetric and asymmetric key encryption techniques rely on the secrecy of private keys
- It is important that mobile code and mobile agents do not carry private keys when migrating to remote sites

Finer Grain Mechanisms

- Explicit operations: **enc** & **dec** acting on single tuple fields
- Mobile agents retrieve encrypted tuples with standard Linda operations (e.g. without decrypting them)
- Actual decryption will take place only at the home site (where the private key is stored) by stationary agents
- Wrong tuples retrieved by mistake have to be explicitly put back

Information Retrieval Agents

- Mobile agents can safely transport and use public keys also on remote sites
- Intermediate results can be encrypted so that they cannot be eavesdropped by other sites
- They can be decrypted only by the home site

CryptoKlava

- A subpackage of Klava providing these new modular extensions
- Based on Sun JCE (Java Cryptography Extension) providing basic interfaces and API for encryption
- Extended classes and extended operations

Types as security tools

Type systems have been successfully used to ensure *type safety* of programs since a long time to avoid *run-time errors*, and guarantee tha data be used consistently with the espected operations.

Recently, work has been done on exploring and designing type systems for security:

- well-typed Java programs (and the corresponding verified bytecode) will never compromise the integrity of certain data
- Type systems for Dπ-calculus (Hennessy-Riely, Yoshida-Hennessy), and (variants of) Ambient calculus (Cardelli-Ghelli-Gordon) have been proposed to control interaction

Type for access and code mobility control

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

- Capability-based Type System [DFPV – TCS2000]
  - Types as specification tool
    - to express control policies of nodes relatively to resource access and code mobility)
    - to abstract process intentions (read, out, spawn, …) relatively to the different localities they interact with or migrate to
  - (static and dynamic) type checking as enforcement mechanism
    - only intentions that match security policies are allowed}
    - only processes whose types comply with network nodes security policies are admissible (well-typed-ness)

Types and Security Policies

- Each node, that indicates a physical machine, or a logical partition has an associated type:

$$l ::^{\delta} P$$

- Type δ describes the security policy of the node, i.e. what process P may do when running at site *l*.
- We have capabilities/privileges in correspondence of each process action

  $\{i, r, o, e, n\}$

Types and Security Policies

- A type is a partial function

$$\delta : Loc \rightharpoonup \Pi$$

Where ***Loc*** indicates the set of *localities* and Π is a collection of non-empty set of *capabilities.*

Examples

- LEGAL $\quad l ::^{[l_1 \mapsto \{i,o\},\ldots]} \mathbf{in}(\ldots)@l_1.\mathbf{nil}$

- NOT LEGAL $l ::^{[l_1 \mapsto \{i,o\},\ldots]} \mathbf{eval}(\ldots)@l_1.\mathbf{nil}$

The role of ypes

- The type of a node is set by a net coordinator and determines the access policy of the node in terms of access rights;
- Type inference permits determining *processes intentions*
- Type checking guarantees that only processes whose intentions match the rights granted by the coordinators are allowed to proceed.
- Example: privilege $[l' \mapsto \{e\}]$ in the type of locality l will enable processes running at l to perform an **eval** actions over l'.

The role of types

Apart from occurring in the specification of a node, type related information are introduced in two other syntactic constructs:

- in action **newloc**(u :δ) where δ specifies the security policy of the new node,
- in templates of formal parameter !u:π where π specifies the access rights corresponding to the operations that the receiving process wants to perform at u.

In both cases, the type information is not strictly necessary: it increases the flexibility of **newloc** (otherwise, some kind of `default policy' should be assigned to the newly created node) and permits a simpler static type checking.

μ_Klaim: A core calculus for Klaim

- We take away :
  - distinction between logical and physical localities/addresses
  - allocation environments
  - higher order communication
  - types with global information
- For types, we have:
  - types with only local information
  - privilege exchanges
  - dynamic modifications of security policies
  - efficient type handling
  - simpler semantics and type systems

μ_Klaim syntax

- Nets

$$N \quad ::= \quad \mathbf{0} \quad \Big| \quad l ::^{\delta} P \quad \Big| \quad N_1 \parallel N_2$$

- Processes

$$P \quad ::= \quad \mathbf{nil} \quad \Big| \quad a.P \quad \Big| \quad P_1 \mid P_2 \quad \Big| \quad A \qquad (A \triangleq P)$$

- Actions

$$\begin{array}{lllll} a & ::= & \mathbf{read}(T)@\ell & \Big| \quad \mathbf{in}(T)@\ell & \Big| \quad \mathbf{out}(t)@\ell \\ & \Big| & \mathbf{eval}(P)@\ell & \Big| \quad \mathbf{newloc}(u : \delta) & \end{array}$$

Tuples and Templates

Templates	T	$::=$	$F \mid F, T$
Tem.Fields	F	$::=$	$f \mid\ !\,x \mid\ !\,u : \pi$
Tuples	t	$::=$	$f \mid f, t$
Tuple Fields	f	$::=$	$e \mid \ell : \mu$
Expressions	e	$::=$	$V \mid x \mid \ldots$

Matching Rules

$$\text{(M}_1\text{)}\quad match(V,V)=\epsilon \qquad\qquad \text{(M}_2\text{)}\quad match(!\,x,V)=[{}^{V}\!/\!{}_{x}]$$

$$\text{(M}_3\text{)}\quad match(l,l)=\epsilon \qquad\qquad \text{(M}_4\text{)}\quad match(!\,u,l)=[{}^{l}\!/\!{}_{u}]$$

$$\text{(M}_5\text{)}\quad \frac{match(F,f)=\sigma_1 \quad match(T,t)=\sigma_2}{match(\ (F,T)\ ,\ (f,t)\ )=\sigma_1\circ\sigma_2}$$

Structural Congruence

$$
\begin{array}{rl}
(\mathsf{Com}) & N_1 \parallel N_2 \equiv N_2 \parallel N_1 \\
(\mathsf{Assoc}) & (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3) \\
(\mathsf{Abs}) & l :: P \equiv l :: (P | \mathbf{nil}) \\
(\mathsf{PrInv}) & l :: A \equiv l :: P \qquad \textit{if} \;\; A \stackrel{\triangle}{=} P \\
(\mathsf{Clone}) & l :: (P_1 | P_2) \equiv l :: P_1 \parallel l :: P_2
\end{array}
$$

Untyped μ-Klaim Semantics

$$\frac{[\![\, t \,]\!] = et}{l :: \mathbf{out}(t)@l'.P \parallel l :: P' \xrightarrow{\mathbf{o}(l,et,l')} l :: P \parallel l :: P' \parallel l' :: \langle et \rangle}$$

$$l :: \mathbf{eval}(Q)@l'.P \parallel l' :: P' \xrightarrow{\mathbf{e}(l,\;,l')} l :: P \parallel l' :: P' | Q$$

$$\frac{match([\![\, T \,]\!], et) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle et \rangle \xrightarrow{\mathbf{i}(l,et,l')} l :: P\sigma \parallel l' :: \mathbf{nil}}$$

$$\frac{match([\![\, T \,]\!], et) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle et \rangle \xrightarrow{\mathbf{r}(l,et,l')} l :: P\sigma \parallel l' :: \langle et \rangle}$$

$$\frac{l' \notin L}{L \vdash l :: \mathbf{newloc}(u).P \xrightarrow{\mathbf{n}(l,-,l')} L \cup \{l'\} \vdash l :: P[^{l'}/_{u}] \parallel l' :: \mathbf{nil}}$$

Typed μ-Klaim Semantics

- We consider now a few of the previous rules by taking into account types.
- We ignore labels, these are not needed in this framework. Labelled semantics is useful for open systems and for logical specification.
- Notation $\delta \vdash_{l} P$ indicates that process *P*, located at *l* complies with the restrictions imposed by type δ.
- A net is well typed if each node, say *l*, complies with $\delta \vdash_{l} P$

Eval

$$\frac{\delta' \vdash_{l'} Q}{l::^{\delta} \mathbf{eval}(Q)@l'.P \parallel l'::^{\delta'} P' \succ\!\!\longrightarrow l::^{\delta} P \parallel l'::^{\delta'} P'|Q}$$

Process Q must be dynamically type checked against the policy of node l', this is necessary since no a-priori knowledge of the target node policy can be assumed, no static checking performed in l over the spawned process can be useful.

newloc

$$\frac{l' \notin L}{L \vdash l ::^{\delta} \mathbf{newloc}(u : \delta').P \succ\!\!\longrightarrow L \cup \{l'\} \vdash l ::^{\delta[l' \mapsto \delta(l)]} P[^{l'}/_{u}] \parallel l' ::^{\delta'[^{l'}/_{u}]} \mathbf{nil}}$$

where $\delta_1[\delta_2]$ denotes the pointwise union of functions δ_1 and δ_2.

It is assumed that the creating node has over the created node all the privileges it has on itself.

The check that $\delta' \preceq \delta[u \mapsto \delta(l)]$ (i.e. the specified access policy δ′ is in agreement with the access policy δ of the node executing the operation extended with the ability of performing over *l*′ all the operations allowed locally) is left to static type inference.

This check prevents a malicious node *l* from forging capabilities by creating a new node with more powerful privileges and then sending malicious process that takes advantage of capabilities not owned by *l*.

in

$$\frac{match_\delta([\![\, T \,]\!], t) = \sigma}{l ::^{\delta} \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle \rightarrowtail l ::^{\delta} P\sigma \parallel l' :: \mathbf{nil}}$$

The new pattern matching function ***match*~δ~** is defined like ***match*** but it also verifies that process Pσ does not perform illegal actions w.r.t. δ. Because of the static inference, the definition of ***match*~δ~** simply requires the following change to untyped *match*.

$$\frac{\pi \subseteq \delta(l')}{match_\delta(!\, u : \pi, l') = [^{l'}/_{u}]}$$

Type Soundness

- processes running in well-typed nets do not attempt to execute actions that are not allowed by the capabilities they own (*type safety*)
- The above property is preserved along reductions (*subject reduction*).

Dynamic security policies

- One of our goal is to enable dynamic modifications of security policies. We want to permit transition like the following where

$$
\begin{array}{l}
l_1 ::^{[l_2 \mapsto \{i\}]} \ \mathbf{in}(!u : \{o\})@l_2.\mathbf{out}(100)@u.\mathbf{nil} \ \ \| \\
l_2 ::^{[l_2 \mapsto \{o\},\ldots]} \ \mathbf{out}(l)@l_2.\mathbf{nil}
\end{array}
$$

$$\succ\!\xrightarrow{\mathbf{out}}\succ\!\xrightarrow{\mathbf{in}}$$

$$
\begin{array}{l}
l_1 ::^{[l_2 \mapsto \{i\}, l \mapsto \{o\}]} \ \mathbf{out}(100)@l.\mathbf{nil} \ \ \| \\
l_2 ::^{[l_2 \mapsto \{o\},\ldots]} \ \mathbf{nil}
\end{array}
$$

l_2 grants l_1 the capability of performing an out at l.

- Problem: How to guarantee that capabilities are not forged?

Exchanging Privileges - 1

1. out: each locality is annotated with the capabilities passed along with it

$$N_1 \triangleq l_1 ::^{[l_2 \mapsto \{i\}]} \mathbf{in}(!u : \{o\})@l_2.\mathbf{out}(100)@u.\mathbf{nil} \;\; \| \;\; l_2 ::^{\delta} \mathbf{out}(l : [l_1 \mapsto \{o,e\}, l_3 \mapsto \{i\}])@l_2.\mathbf{nil}$$

2. When out is performed it is checked that the capabilities passed along with the localities be really owned by the node performing the out

$$N_1 \xrightarrow{\mathbf{out}} l_1 ::^{[l_2 \mapsto \{i\}]} \mathbf{in}(!u : \{o\})@l_2.\mathbf{out}(100)@u.\mathbf{nil} \;\; \| \;\; l_2 ::^{\delta} \mathbf{tuple}(l : [l_1 \mapsto \{o,e\}, l_3 \mapsto \{i\}])$$

only if $\{o,e,i\} \subseteq \delta(l)$

Exchanging Privileges - 2

3. When a read/in is performed (communication takes place) it is verified that the accessed tuple can pass all the capabilities required in the template to the locality performing the read/in

$$
\begin{array}{l}
\qquad l_1 ::^{[l_2 \mapsto \{i\}]} \ \mathbf{in}(!u : \{o\})@l_2.\mathbf{out}(100)@u.\mathbf{nil} \ \ \| \\
\qquad l_2 ::^{\delta} \mathbf{tuple}(l : [l_1 \mapsto \{o, e\}, l_3 \mapsto \{i\}]) \\
\xrightarrow{\mathbf{in}} \\
\qquad l_1 ::^{[l_2 \mapsto \{i\}, l \mapsto \{o\}]} \ \mathbf{out}(100)@l.\mathbf{nil} \ \ \| \\
\qquad l_2 ::^{\delta} \mathbf{nil} \\
\text{because } \{o\} \subseteq \{o, e\}
\end{array}
$$

Additional Information

- More in a paper just presented at ICALP by D. Gorla and R. Pugliese.

Klaim site

http://music.dsi.unifi.it

- A few papers
- Current Implementation:
  - KriptoKlava
  - Type Checker for Access Control (?)