Static semantics of secret channel abstractions

Marco Giunti

University of Porto and University of Beira Interior, Portugal

NordSec, October 16 2014

Motivation

- The pi calculus and its variants based on cryptographic operations are often used for protocol analysis
- E.g. googling - "pi calculus" protocol - returns 50k hits
- All pi calculus variants make use of the **new** (restriction) operator
- The **new** operator allows to
    1. create a channel name and limit its use within a certain scope
    2. enlarge the channel's scope by communicating the channel to others

Security problems

- The scope extrusion mechanism allows the mobility of the communication structure (and the great expressiveness of the pi calculus), but comports security problems
- Restricted channels cannot be implemented as dedicated channels, and open channels are not secure by default
- The spi calculus and the applied pi calculus do not rely on restriction for secure communication and use cryptographic encryption

Motivating example

A simple protocol to exchange a confidential information

$$P = (\mathsf{new}\, c)((\mathsf{new}\, s)(\overline{c}\langle s\rangle.\overline{s}\langle \mathrm{pwd}\rangle) \mid c(x).x(y).\overline{p}\langle x\rangle)$$

- Two parallel threads communicating over restricted channel c
- The left thread generates a (secure) channel s to send the password, and forwards s over c
- The right thread receives a channel x from c, uses x to retrieve some data, and releases x over a public (free) channel p
- How to implement this protocol in an open network ?

Example: naive implementation

To avoid dedicated channels we use public key cryptography.
– $(\text{new}\, s)$ mapped into generation of keys $(\text{new}\, s^+, s^-)$
–Aim: to encrypt the password: $\{\mathrm{pwd}\}_{s^+}$

pi: $c(x).x(y).\overline{p}\langle x\rangle$ (1)

spi: $net(z).\text{decrypt } z \text{ as } \{x^+, x^-\}_{c^-} \text{ in}$ (2)

$net(w).\text{decrypt } w \text{ as } \{y\}_{x^-} \text{ in } \overline{p}\langle x^+, x^-\rangle$

- (2) is the (spi calculus) code of the receiver in (1)
- Keys sent on the network through the packet $\{s^+, s^-\}_{c^+}$
- To retrieve s^+, s^- processes must use the decryption key c^-

Example: naive implementation

Lack of forward secrecy

The implementation above suffers from a number of problems.

- The most serious is the lack of forward secrecy
- Informally: password in $\{\mathrm{pwd}\}_{s^+}$ can be retrieved by buffering the message and subsequently using the key s^-
- Formally: the behavioral equation of pi calculus below is not preserved by the spi calculus translation

$$P = (\mathsf{new}\, c)((\mathsf{new}\, s)(\overline{c}\langle s\rangle.\overline{s}\langle \mathrm{pwd}\rangle) \mid c(x).x(y).\overline{p}\langle x\rangle)$$
$$P \cong (\mathsf{new}\, s)(\overline{p}\langle s\rangle) \qquad\qquad (p \in \mathsf{fv}(P))$$

- The equation ensures a well-known fact: in the pi calculus restricted communications are invisible

A secret pi calculus (Sπ)

- To avoid this problem in EXPRESS/SOS'12 we introduced a pi calculus featuring both a **new** and a **hide** operator
- The **new** operator does not ensure any secrecy: that is, in secret pi:

$$P \not\cong (\text{new } s)(\overline{p}\langle s \rangle)$$

- To recover the equation programmers must use the **hide** operator:

$$H = (\text{new } c)([\text{hide } s][\overline{c}\langle s \rangle.\overline{s}\langle \mathrm{pwd} \rangle \mid c(x).x(y).\overline{p}\langle x \rangle])$$
$$H \cong_{\mathsf{S}\pi} [\text{hide } s][\overline{p}\langle s \rangle]$$

- The brackets delimit the **static** scope of **hide**, which includes the receiver. Note: s **cannot be extruded** (e.g. by $\overline{p}\langle s \rangle$)

A type system to control the scope of channels

- In the secret pi calculus the scope of channels protected by **hide** is managed by the reduction system
- The runtime system can be interpreted as a specialized middleware for secure communications featuring local channels
- This talk: a type system for a standard pi calculus that disallows the extrusion of channels "declared" as *static*
- Our construction can be seen as an API for secure programming:

  – channels protected by **hide** are translated into typed channels with static scope

  – processes trying to leak secret (static) channels are rejected

Syntax of pi calculus types and processes

T ::=		Types:	P ::=		Processes:
	$m\,\mathsf{chan}\langle T \rangle$	channel		$x(y \div B).P$	input
	$\top$	top		$\cdots$	pi
m ::=		Modalities	B ::=		Blocked entry:
	s	static		$\emptyset$	empty
	d	dynamic		$B \cup \{T\}$	type

- I/o types are decorated with a scope modality
- Input processes decorated with blocked types to instruct the type checker: semantics unaffected
- When B is empty: $x(y).P \stackrel{\text{def}}{=} x(y \div \emptyset).P$

Example, typed syntax

- We rewrite the secret pi calculus process

$$H = (\text{new}\, c)([\text{hide}\, s][H'])$$
$$H' = \overline{c}\langle s\rangle.\overline{s}\langle \mathrm{pwd}\rangle \mid c(x).x(y).\overline{p}\langle x\rangle$$

- Typed syntax:

$$P = (\text{new}\, c\colon \mathsf{d\ chan}\langle T_2\rangle)((\text{new}\, s\colon \mathsf{s\ chan}\langle \top\rangle)(H'))$$
$$T_2 = \mathsf{d\ chan}\langle \top\rangle$$

- Note: An upcast mechanism allows to send s over c by changing the type of c to $\mathsf{d\ chan}\langle \mathsf{s\ chan}\langle \top\rangle\rangle$

An (abstract) API for secure programming

- We let programmers write code with the secret pi syntax

$$H = (\mathsf{new}\, c)([\mathsf{hide}\, s][H'])$$
$$H' = \overline{c}\langle s\rangle.\overline{s}\langle \mathrm{pwd}\rangle \mid c(x).x(y).\overline{p}\langle x\rangle$$

- Code translated by guessing payload types of channels, scope modalities inferred automatically
- E.g. pwd has top type, s brings values of top type, ...

$$[\![H]\!] = (\mathsf{new}\, c \colon \mathsf{d\ chan}\langle T_2\rangle)((\mathsf{new}\, s \colon \mathsf{s\ chan}\langle \top\rangle)(H'))$$

- Payload types different from top have a dynamic modality, e.g. $T_2 = \mathsf{d\ chan}\langle \top\rangle$

Static type checking

- Given the expected (dynamic) type T for p, we have

$$p\colon T \vdash [\![H]\!]$$
$$[\![H]\!] = (\mathsf{new}\, c\colon \mathsf{d\ chan}\langle T_2\rangle)(\mathsf{new}\, s\colon \mathsf{s\ chan}\langle \top\rangle)$$
$$(\overline{c}\langle s\rangle.\overline{s}\langle \mathrm{pwd}\rangle) \mid c(x).x(y).\overline{p}\langle x\rangle)$$

- More interestingly, the type system rejects attempts to leak channel s from p
- Specifically: the composition $[\![H]\!] \mid p(x)$ is ill-typed
- This is mandatory, as the reduction semantics of the pi calculus would allow the interaction of the two threads
- How we obtain this?

Downcasting to the rescue

- To type check $[\![H]\!]$ the payload type T_2 of c in the left thread must be upcasted to the type $\mathsf{s}\ \mathsf{chan}\langle\top\rangle$ (*)

$$[\![H]\!] = (\mathsf{new}\, c\colon \mathsf{d}\ \mathsf{chan}\langle T_2\rangle)(\mathsf{new}\, s\colon \mathsf{s}\ \mathsf{chan}\langle\top\rangle)$$
$$(\overline{c}\langle s\rangle.\overline{s}\langle \mathrm{pwd}\rangle) \mid c(x).x(y).\overline{p}\langle x\rangle)$$

- The right thread must assign T_2 as payload type of c as well, since channel c is used in i/o (specifically, it is used in input)
- In turn, the variable x gains type (*), and the "final" type of p is downcasted to the special type $\bullet$ (void) to disallow extrusion
- The void type is not accessible to the programmer and is used in *return* environments to forbid the leak of static channels

Tracking the usage of channels

- We use return environments to keep track of the effective usage of channels
- Our judgements have the form

$$\Gamma \vdash P \triangleright \Delta$$

where Δ is a type environment with codomain$=$ $\mathit{Types} \cup \{\bullet\}$
- The technique is reminding of those for algorithmic type checking of linear systems
- The typing rule for parallel crucially asks that return environments can be composed

$$\frac{\Gamma \vdash P_1 \triangleright \Delta_1 \qquad \Gamma \vdash P_2 \triangleright \Delta_2}{\Gamma \vdash P_1 \triangleright \Delta_1 \otimes \Delta_2}$$

- Otimes: a void type can only be composed with top

Running example

- Given a suitable type T, we have that

$$p\colon T \not\vdash [\![H]\!] \mid p(x) \triangleright \Delta'$$

$$[\![H]\!] = (\mathsf{new}\, c\colon \mathsf{d\ chan}\langle T_2\rangle)(\mathsf{new}\, s\colon \mathsf{s\ chan}\langle \top\rangle) \\ (\overline{c}\langle s\rangle.\overline{s}\langle \mathrm{pwd}\rangle) \mid c(x).x(y).\overline{p}\langle x\rangle)$$

for any Δ' since:

- $p\colon T \vdash [\![H]\!] \triangleright p\colon \bullet$
- $p\colon T \vdash p(x) \triangleright \Delta$ with $\Delta(p) \neq \top$
- In contrast:
  $p\colon T \vdash [\![H]\!] \mid (\mathsf{new}\, p'\colon T')p'(x) \triangleright p\colon \bullet$ since
  $p\colon T \vdash (\mathsf{new}\, p'\colon T')p'(x) \triangleright p\colon \top$

Blocked types in input

- Following standard lines, we consider a pi calculus with reduction semantics and structural congruence ($\equiv$)
- Blocked types in input inserted in $\equiv$ scope extrusion rule
- Example:

$$[\![H]\!] \mid p(z \div \emptyset) \equiv (\mathsf{new}\, c \colon \mathsf{d\ chan}\langle T_2 \rangle)(\mathsf{new}\, s \colon \mathsf{s\ chan}\langle \top \rangle) \\ (\overline{c}\langle s \rangle.\overline{s}\langle \mathrm{pwd} \rangle) \mid c(x).x(y).\overline{p}\langle x \rangle \mid p(z \div \{\mathsf{s\ chan}\langle \top \rangle\}))$$

- Process $p(z \div \{\mathsf{s\ chan}\langle \top \rangle\})$ cannot upcast the required payload type since it is blocked
- In detail: types must have identifiers in order to avoid clashes: $(\mathsf{new}\, c \colon \mathsf{d\ chan}\langle T_2 \rangle_\forall)(\mathsf{new}\, s \colon \mathsf{s\ chan}\langle \top \rangle_n)(\dots)$ $\quad n$ perfect id

Soundness and expressiveness

- Typed processes reduce to typed processes (SR)
- Operational correspondence among (a fragment of) secret π-calculus processes and their typed translation

  Assume Γ, Δ such that $\Gamma \vdash [\![H]\!] \triangleright \Delta$.

  1. $H \to H'$ implies $[\![H]\!] \to [\![H']\!]$
  2. $[\![H]\!] \to Q$ implies $H \to H'$ with $[\![H']\!] \equiv Q$
- Note the typability assumption, essential to switch from middleware to software support of secret channels

Applications: protection against 3rd party code

Example: malicious list handler

$$\{\!|\, () \,|\!\}_z = \overline{z}\langle \bot, \bot, \bot \rangle$$

$$\{\!|\, (\langle a_0, b_0\rangle, \ldots, \langle a_n, b_n\rangle) \,|\!\}_z = (\mathsf{new}\, z')(\overline{z}\langle a_0, b_0, z'\rangle \mid \{\!|\, (\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle) \,|\!\}_{z'})$$

$$\textsc{Add}(x, y, z) = z(h_1, h_2, z').((\mathsf{new}\, z'')(\overline{z}\langle x, y, z''\rangle \mid \overline{z''}\langle h_1, h_2, z'\rangle) \mid \overline{\mathrm{port888}}\langle h_1, h_2\rangle) \qquad \text{\%\% Suspicious}$$

Fix: re-program the list, compile and ...

$$\textsc{StoreSecCh}(H, y) = [\mathsf{hide}\, x][H \mid (\mathsf{new}\, z)(\{\!|\, () \,|\!\}_z \mid \textsc{Add}(x, y, z))]$$

ask to the type-checker! $\Gamma \overset{?}{\vdash} [\![\textsc{StoreSecCh}(H, y)]\!] \mid Q$

Applications: Mandatory access control

DBUS is an *IPC* system using private and public bus for communication
–Previous versions: bug allows users to listen private bus

```
[marco]# echo $DBUS_SESSION_BUS_ADDRESS > Public/address
[guest]# dbus-monitor --address /home/marco/Public/address
```

- We interpret this issue as MAC problem
- The private session bus address cannot be disclosed by its owner
- Fix: program the bus with hide. All users trying to leak the channel will be rejected

Limitations

- We just deal with direct information flows
  –We need protection against indirect flows, covert channels...
- Typed analysis does not scale
  $\Gamma \vdash P$ and $\Gamma \vdash Q$ does not imply $\Gamma \vdash P \mid Q$
- Static typing is too demanding
  – We would need lightweight (dynamic) typing integrated with advanced functionalities
  – E.g. contracts, certificates, functions, cryptographic operations, ...

Extensions

- To understand better the static semantics of programs we need typed behavioural equivalences, typed bisimulation, ...
- The system has been designed to be easily integrated with other type systems
  –E.g. linear types, affine types, session types, ...
- Further design choice: keep the system algorithmic as possible
  –Algorithmic type checking and inference obtainable easily (by extending code of previous tools)

Thanks!

Questions?

Recent related work

- Myself: Algorithmic type checking for a pi-calculus with name matching and session types. J. Log. Algebr. Program. 82(8)
- Myself, Antonio Ravara: Towards Static Deadlock Resolution in the pi-Calculus. TGC 2013: 136-155
- Myself, Catuscia Palamidessi, Frank D. Valencia: Hide and New in the Pi-Calculus. EXPRESS/SOS 2012