IS&C

UNIVERSITÄT DES SAARLANDES

A Type Discipline for Authorization in Distributed Systems

Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis

20th IEEE Computer Security Foundations Symposium (CSF), 2007

Presenter: Cătălin Hriţcu

Information Security & Cryptography Group

Reading Group Presentation, Saarbrücken, April 2008

A simple example

user

proxy

store

Order(alice, emo)

Registered(alice)

<alice, emo>

<alice, emo>

CanDownload(alice, emo)?

A simple example

user

proxy

store

Order(alice, emo)

Registered(alice)

<alice, emo>

<alice, emo>

CanDownload(alice, emo)?

policy = $\forall$user,$\forall$song(Order(user, song) $\wedge$ Registered(user) $\Rightarrow$ CanDownload(user, song))

A simple example

user

proxy

store

Order(alice, emo)

Registered(alice)

<alice, emo>

<alice, emo>

CanDownload(alice, emo)?

opponent

<alice, evil-song>

CanDownload(alice, evil-song)?

policy = ∀user,∀song(Order(user, song) ∧ Registered(user) ⇒ CanDownload(user, song))

A simple example

user

proxy

store

Order(alice, emo)

Registered(alice)

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

policy = ∀user,∀song(Order(user, song) ∧ Registered(user) ⇒ CanDownload(user, song))

A simple example

user

proxy

store

Order(alice, emo)

Registered(alice)

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

- Is the authorization policy respected in all executions? (safety)

  ... even in the presence of an arbitrary opponent? (robust safety)

  ... even in the presence of corrupted participants? (safety despite compromise)

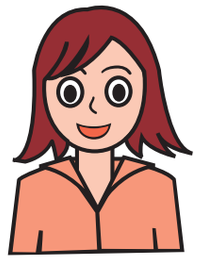
policy = $\forall$user,$\forall$song(Order(user, song) $\wedge$ Registered(user) $\Rightarrow$ CanDownload(user, song))

Outline

- Modeling protocols and policies (formally)
  - the process calculus: terms, destructors, processes, semantics
  - authorization logic: some assumptions (otherwise parametric)
  - desired security guarantees:
    - safety, robust safety, safety despite compromise
- 5 minutes break
- Type system
- Illustrating the proof technique
  - Subject-reduction ⇒ Safety
  - Safety ∧ Opponent Typability ⇒ Robust Safety
- Safety despite compromised participants
- Related and future work

Modeling protocols and policies

Modeling the example

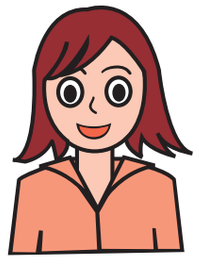


user
Order(alice, emo)

proxy
Registered(alice)

store

Modeling the example



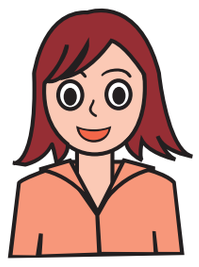
user
Order(alice, emo)

proxy
Registered(alice)

store

let user = **assume** Order(alice, emo) |

Modeling the example



user
Order(alice, emo)

proxy
Registered(alice)

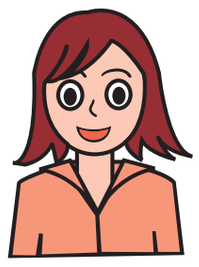
store

sign(emo, ka) →

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**

Modeling the example


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

sign(<alice, emo>, kp)

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**

Modeling the example

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

Modeling the example

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

Modeling the example


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Spi calculus with destructors

Spi calculus with destructors

- Constructors

$$f ::= \mathsf{pk}^1, \mathsf{enc}^2, \mathsf{vk}^1, \mathsf{sign}^2, \mathsf{hash}^1, \ldots$$

Spi calculus with destructors

- Constructors

$$f ::= \mathsf{pk}^1, \mathsf{enc}^2, \mathsf{vk}^1, \mathsf{sign}^2, \mathsf{hash}^1, \ldots$$

- Terms

$$M, N, K ::= a \mid x \mid f(M_1, \ldots, M_n) \mid \langle M_1, \ldots, M_n \rangle$$

Spi calculus with destructors

- Constructors

$$f ::= \mathsf{pk}^1, \mathsf{enc}^2, \mathsf{vk}^1, \mathsf{sign}^2, \mathsf{hash}^1, \ldots$$

- Terms

$$M, N, K ::= a \mid x \mid f(M_1, \ldots, M_n) \mid \langle M_1, \ldots, M_n \rangle$$

- Destructors

$$g ::= \mathsf{dec}^2, \mathsf{check}^2, \ldots$$

Spi calculus with destructors

- Constructors

$$f ::= \mathsf{pk}^1, \mathsf{enc}^2, \mathsf{vk}^1, \mathsf{sign}^2, \mathsf{hash}^1, \ldots$$

- Terms

$$M, N, K ::= a \mid x \mid f(M_1, \ldots, M_n) \mid \langle M_1, \ldots, M_n \rangle$$

- Destructors

$$g ::= \mathsf{dec}^2, \mathsf{check}^2, \ldots$$

- Reduction rules for destructors

$$\mathsf{dec}(\mathsf{enc}(M, \mathsf{pk}(K)), K) \quad \Downarrow \quad M$$

$$\mathsf{check}(\mathsf{sign}(M, K), \mathsf{vk}(K)) \quad \Downarrow \quad M$$

Processes

$P, Q ::=$	$\mathbf{out}(M, N).P$	output
	$\mathbf{in}(M, x).P$	input
	$\mathbf{new}\ a : T.P$	restriction
	$!P$	replication
	$P \mid Q$	parallel composit.
	$\mathbf{0}$	null process
	$\mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{then}\ P\ \mathbf{else}\ Q$	destructor applic.
	$\mathbf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathbf{in}\ P$	pair splitting
	$\mathbf{assume}\ C$	assume formula C
	$\mathbf{assert}\ C$	expect C to hold

Processes

$P, Q ::=$	$\mathbf{out}(M, N).P$	output
	$\mathbf{in}(M, x).P$	input
	$\mathbf{new}\ a : T.P$	restriction
	$!P$	replication
	$P \mid Q$	parallel composit.
	$\mathbf{0}$	null process
	$\mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{then}\ P\ \mathbf{else}\ Q$	destructor applic.
	$\mathbf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathbf{in}\ P$	pair splitting
	$\mathbf{assume}\ C$	assume formula C
	$\mathbf{assert}\ C$	expect C to hold

- Evaluation context

  $\mathcal{E} ::= \mathbf{new}\ a_1 : T_1 \ldots \mathbf{new}\ a_n : T_n.\ [\ ] \mid P$

Semantics

- Structural equivalence

$$
\begin{array}{rcl}
P \mid \mathbf{0} & \equiv & P \\
P \mid Q & \equiv & Q \mid P \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \\
!P & \equiv & !P \mid P \\
\mathbf{new}\ a : T.(P \mid Q) & \equiv & P \mid \mathbf{new}\ a.Q,\ \mathit{if}\ a \notin \mathit{fn}(P) \\
\mathbf{new}\ a_1.\mathbf{new}\ a_2.P & \equiv & \mathbf{new}\ a_2.\mathbf{new}\ a_1.P,\ \mathit{if}\ a_1 \neq a_2 \\
\mathcal{E}[P] & \equiv & \mathcal{E}[Q],\ \mathit{if}\ P \equiv Q
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{out}(a, M).P \mid \mathbf{in}(a, x).Q & \rightarrow & P \mid Q\{M/x\} \\
\mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{then}\ P\ \mathbf{else}\ Q & \rightarrow & P\{N/x\},\ \mathit{if}\ g(M_1, \ldots, M_n) \Downarrow N \\
\mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{then}\ P\ \mathbf{else}\ Q & \rightarrow & Q, \quad \mathit{if}\ g(M_1, \ldots, M_n) \not\Downarrow \\
\mathbf{let}\ \langle x_1, \ldots, x_n \rangle = \langle M_1, \ldots, M_n \rangle\ \mathbf{in}\ P & \rightarrow & P\{M_i/x_i\} \\
\mathcal{E}[P] & \rightarrow & \mathcal{E}[Q], \quad \mathit{if}\ P \rightarrow Q \\
P & \rightarrow & Q,\ \mathit{if}\ P \equiv P', P' \rightarrow Q', Q' \equiv Q
\end{array}
$$

Semantics

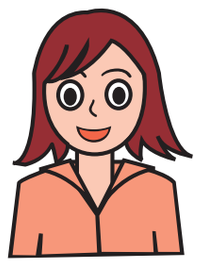
- Structural equivalence

$$
\begin{array}{rcl}
P \mid \mathbf{0} & \equiv & P \\
P \mid Q & \equiv & Q \mid P \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \\
!P & \equiv & !P \mid P \\
\mathbf{new}\ a : T.(P \mid Q) & \equiv & P \mid \mathbf{new}\ a.Q,\ \mathit{if}\ a \notin \mathit{fn}(P) \\
\mathbf{new}\ a_1.\mathbf{new}\ a_2.P & \equiv & \mathbf{new}\ a_2.\mathbf{new}\ a_1.P,\ \mathit{if}\ a_1 \neq a_2 \\
\mathcal{E}[P] & \equiv & \mathcal{E}[Q],\ \mathit{if}\ P \equiv Q
\end{array}
$$

- Internal reduction

$$
\begin{array}{rcl}
\mathbf{out}(a, M).P \mid \mathbf{in}(a, x).Q & \rightarrow & P \mid Q\{M/x\} \\
\mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{then}\ P\ \mathbf{else}\ Q & \rightarrow & P\{N/x\},\ \mathit{if}\ g(M_1, \ldots, M_n) \Downarrow N \\
\mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{then}\ P\ \mathbf{else}\ Q & \rightarrow & Q, \quad \mathit{if}\ g(M_1, \ldots, M_n) \not\Downarrow \\
\mathbf{let}\ \langle x_1, \ldots, x_n \rangle = \langle M_1, \ldots, M_n \rangle\ \mathbf{in}\ P & \rightarrow & P\{M_i/x_i\} \\
\mathcal{E}[P] & \rightarrow & \mathcal{E}[Q], \quad \mathit{if}\ P \rightarrow Q \\
P & \rightarrow & Q,\ \mathit{if}\ P \equiv P', P' \rightarrow Q', Q' \equiv Q
\end{array}
$$

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

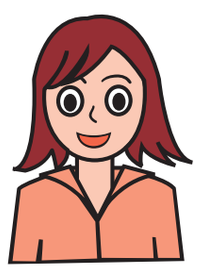
let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka) →

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

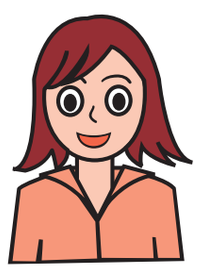
let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka) →

let user = **assume** Order(alice, emo).

let proxy = **assume** Registered(alice) |

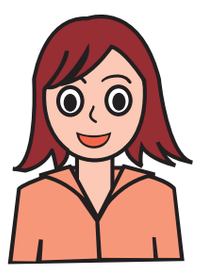
let <s> = check(sign(<emo>, ka), vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka) →

let user = **assume** Order(alice, emo).

let proxy = **assume** Registered(alice) |

let <s> = check(sign(<emo>, ka), vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

check(sign(M, K), vk(K)) ⇓ M

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka) →

let user = **assume** Order(alice, emo).

let proxy = **assume** Registered(alice) |

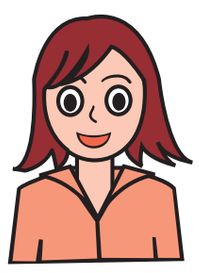
let <s> = <emo> **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

let user = **assume** Order(alice, emo).

let proxy = **assume** Registered(alice) |

let <s> = <emo> **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka) →

let user = **assume** Order(alice, emo).

let proxy = **assume** Registered(alice) |

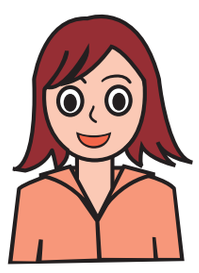
out(c, sign(<alice, emo>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

let user = **assume** Order(alice, emo).

sign(<alice, emo>, kp)

let proxy = **assume** Registered(alice) |

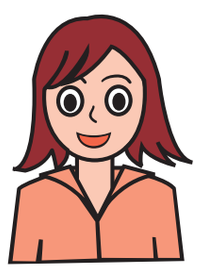
out(c, sign(<alice, emo>, kp))

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

let user = **assume** Order(alice, emo).

sign(<alice, emo>, kp)

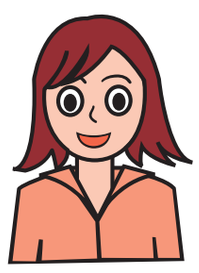
let proxy = **assume** Registered(alice).

let store =
  let <user, song> = check(sign(<alice, emo>, kp), vk(kp)) **in**
  assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
  (user | proxy | store | policy)

Executing the simple example


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

let user = **assume** Order(alice, emo).

sign(<alice, emo>, kp)

let proxy = **assume** Registered(alice).

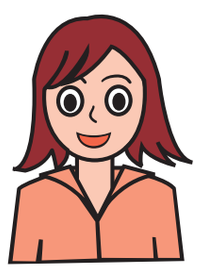
check(sign(M, K), vk(K)) ⇓ M

let store =
let <user, song> = check(sign(<alice, emo>, kp), vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

let user = **assume** Order(alice, emo).

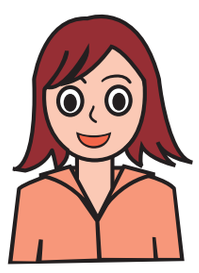
let proxy = **assume** Registered(alice).

let store =
  let <user, song> = <alice, emo> **in**
  assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
  (user | proxy | store | policy)

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

let user = **assume** Order(alice, emo).

sign(<alice, emo>, kp)

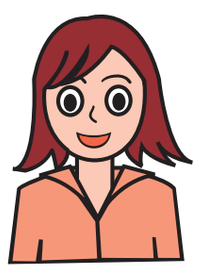
let proxy = **assume** Registered(alice).

let store =
let <user, song> = <alice, emo> **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

let user = **assume** Order(alice, emo).

sign(<alice, emo>, kp)

let proxy = **assume** Registered(alice).

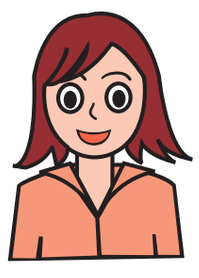
let store =

assert CanDownload(alice, emo).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

let user = **assume** Order(alice, emo).

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let proxy = **assume** Registered(alice).

let store =

assert CanDownload(alice, emo)

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo).

let proxy = **assume** Registered(alice).

let store =

assert CanDownload(alice, emo)

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Executing the simple example

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo).

let proxy = **assume** Registered(alice).

let store = {Order(alice, emo), Registered(alice),
∀u,∀s(Order(u, s) ∧ Registered(u)
⇒ CanDownload(u, s)) } |= CanDownload(alice, emo) ✔

assert CanDownload(alice, emo) ✔

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Authorization logic

- Any authorization logic - framework is "parametric"
- Conditions:

  Expansivity: $C \in A$ implies that $A \models C$;

  Monotonicity: $A \models C$ and $A \subseteq A'$ then $A' \models C$;

  Idempotence: $A \models A'$ and $A \cup A' \models C$ then $A \models C$;

  Substitution: $A \models C$ then $A\sigma \models C\sigma$;

- Examples: Datalog, CDD (more later), DCC, FOL, etc.

Safety

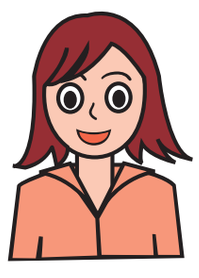
- All assertions hold in all executions
- Definition (Safety). A process P is safe iff

$$P \rightarrow^* \mathbf{new}\ a_1 : T_1 \ldots \mathbf{new}\ a_n : T_n.(\mathbf{assert}\ C \mid Q) \Rightarrow$$
$$\exists \mathcal{E} = \mathbf{new}\ b_1 : U_1 \ldots \mathbf{new}\ b_m : U_m.\ [\ ] \mid Q' \wedge$$
$$Q \equiv \mathcal{E}[\mathbf{assume}\ C_1 \mid \ldots \mid \mathbf{assume}\ C_k] \wedge$$
$$\{C_1, \ldots, C_k\} \models C \ \wedge\ fn(C) \cap \{b_1, \ldots, b_m\} = \emptyset$$

Robust safety

- Safety in the presence of an arbitrary attacker
- Definition (Opponent). A process O is an opponent iff
  - it does not contain any **assert**
  - it only uses the special type Un (i.e. opponents are untyped)
- This corresponds to the usual Dolev-Yao attacker
- Definition (Robust Safety). Process P is robustly safe iff process $P \mid O$ is safe for all opponents O.

Is our protocol robustly safe?



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

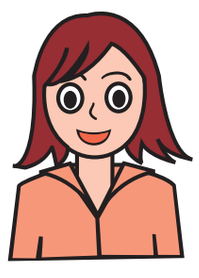
let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Is our protocol robustly safe?



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

- Probably, but with what I have shown you so far we have no easy way to prove that a protocol is robustly safe

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Is our protocol robustly safe?

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

- Probably, but with what I have shown you so far we have no easy way to prove that a protocol is robustly safe
- The type system will do the job

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Safety despite compromise

- Safety when part of the protocol is compromised
- Definition attempt (Safety Despite Compromise).

A process $\mathcal{E}[Q]$ is safe despite the compromise of Q (where $\mathcal{E} = \mathbf{new}\ c_1 : T_1 \ldots \mathbf{new}\ c_n : T_n.\ [\ ] \mid Q^*$) iff $\mathcal{E}[\mathbf{out}(c, c_1) \mid \ \ldots\ \mid \mathbf{out}(c, c_n)]$ is robustly safe

Compromising as little as possible



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Compromising as little as possible



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy | [0])

Compromise this

Compromising as little as possible



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy | [0])

- In this case safety despite compromise reduces to robust safety

Compromising as much as possible

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

```
let user = assume Order(alice, emo) |
           out(c, sign(<emo>, ka)).

let proxy = assume Registered(alice) |
            in(c,m);
            let <s> = check(m, vk(ka)) in
            out(c, sign(<alice, s>, kp)).

let store = in(c, x);
            let <user, song> = check(x, vk(kp)) in
            assert CanDownload(user, song).

let policy = assume ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process  new ka; new kp;
         [user | proxy | store | policy]
```

Compromising as much as possible

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

```
let user = assume Order(alice, emo) |
           out(c, sign(<emo>, ka)).

let proxy = assume Registered(alice) |
            in(c,m);
            let <s> = check(m, vk(ka)) in
            out(c, sign(<alice, s>, kp)).

let store = in(c, x);
            let <user, song> = check(x, vk(kp)) in
            assert CanDownload(user, song).

let policy = assume ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).


process  new ka; new kp;
            [out(c, ka) | out(c, kp) | 0 | 0]
```

Compromising as much as possible

process **new** ka; **new** kp;
[**out**(c, ka) | **out**(c, kp) | 0 | 0]

Compromising as much as possible

process **new** ka; **new** kp;
[**out**(c, ka) | **out**(c, kp) | 0 | 0]

- Compromising the whole protocol is always (vacuously) safe - no asserts

Compromising the user



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
[user] | proxy | store | policy

Compromising the user

ka

proxy
Registered(alice)

store

```
let proxy = assume Registered(alice) |
            in(c,m);
            let <s> = check(m, vk(ka)) in
            out(c, sign(<alice, s>, kp)).

let store = in(c, x);
            let <user, song> = check(x, vk(kp)) in
            assert CanDownload(user, song).

let policy = assume ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).


process  new ka; new kp;
            [out(c, ku)] | proxy | store | policy
```

Compromising the user

proxy
Registered(alice)

store

ka

sign(evil-song, ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

```
let proxy = assume Registered(alice) |
            in(c,m);
            let <s> = check(m, vk(ka)) in
            out(c, sign(<alice, s>, kp)).

let store = in(c, x);
            let <user, song> = check(x, vk(kp)) in
            assert CanDownload(user, song).

let policy = assume ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).


process  new ka; new kp;
            [out(c, ku)] | proxy | store | policy
```

Compromising the user

proxy
Registered(alice)

store

ka

sign(evil-song, ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

- The store accepts the download even if this does not respect the policy (the user never asked for the download)

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
[**out**(c, ku)] | proxy | store | policy

Compromising the user

proxy
Registered(alice)

store

ka

sign(evil-song, ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

```
let proxy = assume Registered(alice) |
            in(c,m);
            let <s> = check(m, vk(ka)) in
            out(c, sign(<alice, s>, kp)).

let store = in(c, x);
            let <user, song> = check(x, vk(kp)) in
            assert CanDownload(user, song).
```

- The store accepts the download even if this does not respect the policy (the user never asked for the download)
- This breaks robust safety, so the protocol is not safe despite the compromise of the user (not surprising)

```
let policy = assume ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process  new ka; new kp;
         [out(c, ku)] | proxy | store | policy
```

Compromising the proxy



user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [proxy] | store | policy)

Compromising the proxy

user
Order(alice, emo)

proxy
Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, ka)] | store | policy)

Compromising the proxy


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, ka)] | store | policy)

Not the right thing to compromise

Safety despite compromise

- The real definition now :)
- Definition (Safety Despite Compromise).

A process $\mathcal{E}[Q]$ is <u>safe despite the compromise</u> of Q (where $\mathcal{E} = \mathbf{new}\ c_1 : T_1 \ldots \mathbf{new}\ c_n : T_n.\ [\ ] \mid Q^*$) iff

$$\exists Q', \sigma \quad (1)\ Q = Q'\sigma \text{ with } fn(Q') \cap \{c_1, \ldots, c_n\} = \emptyset$$
$$(2)\ \mathcal{E}[\textstyle\prod_{x \in dom(\sigma)} \mathbf{out}(c, x\sigma)] \text{ is robustly safe (where } c \notin \{c_1, \ldots, c_n\})$$

Compromising the proxy


user
Order(alice, emo)


proxy
Registered(alice)


store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

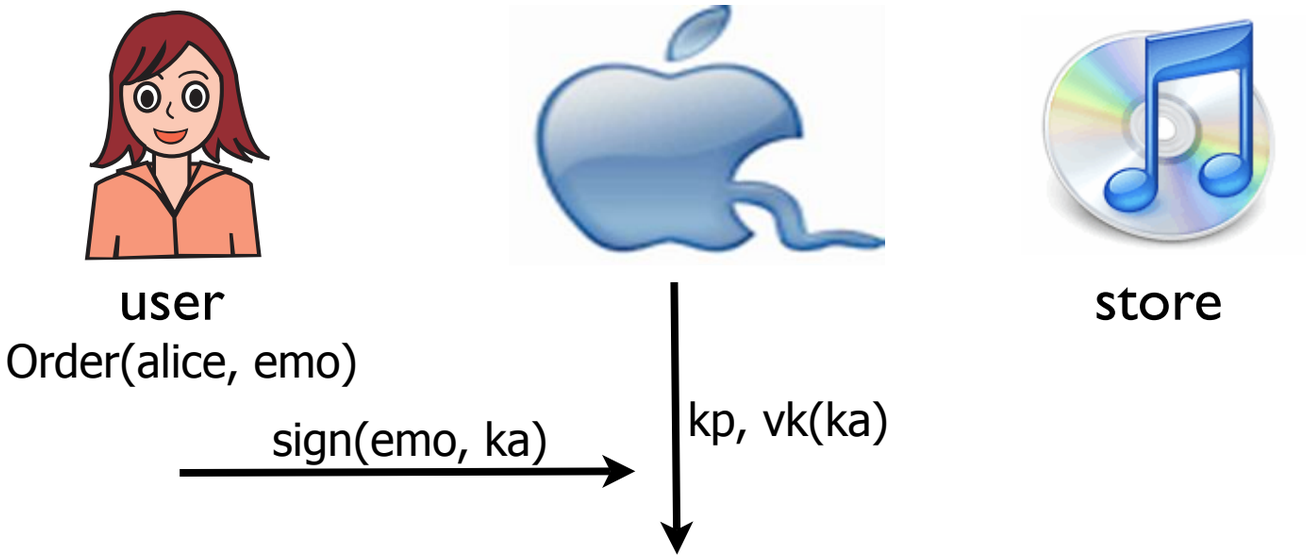
let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, ka)] | store | policy)

Not the right thing to compromise

Compromising the proxy



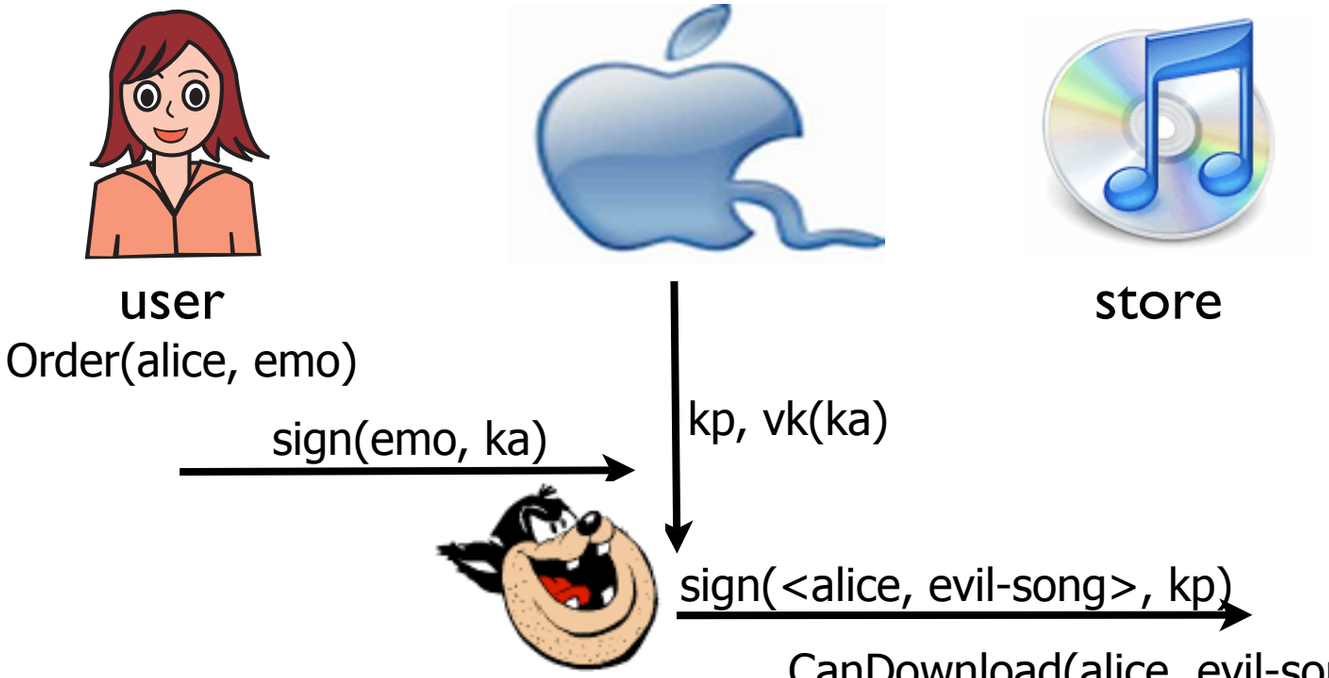
let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy



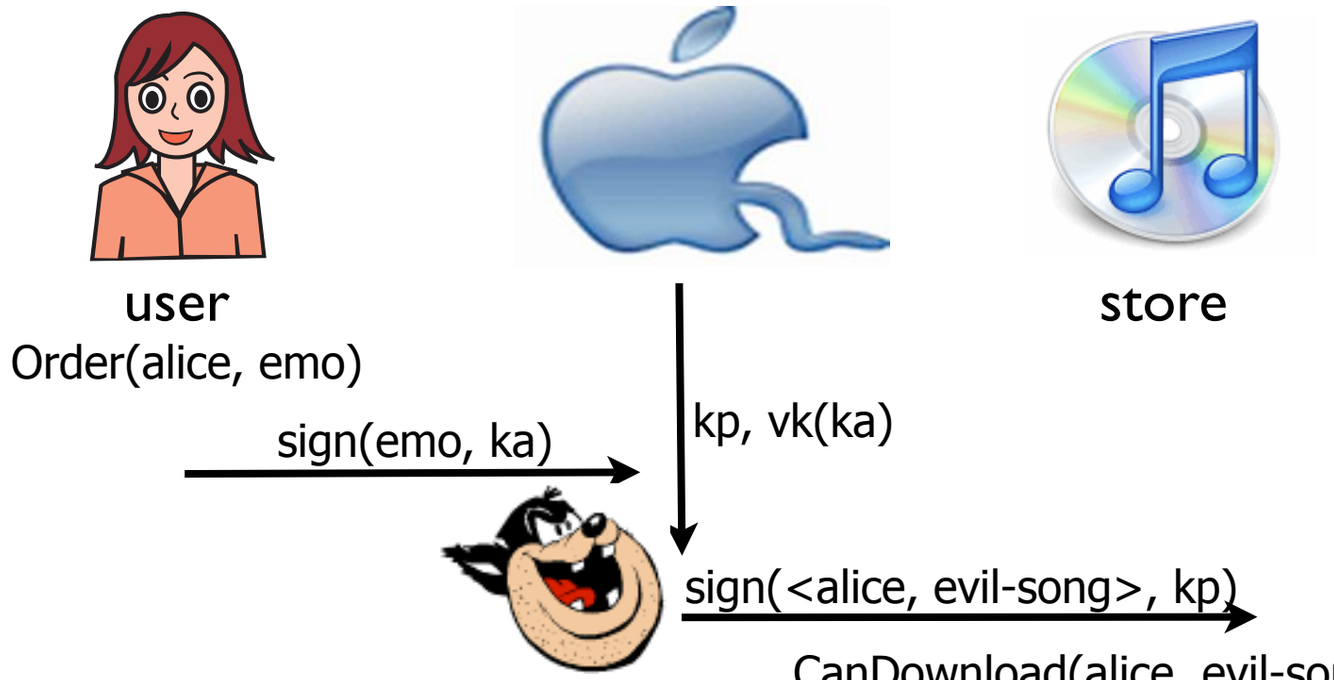
let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy



let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

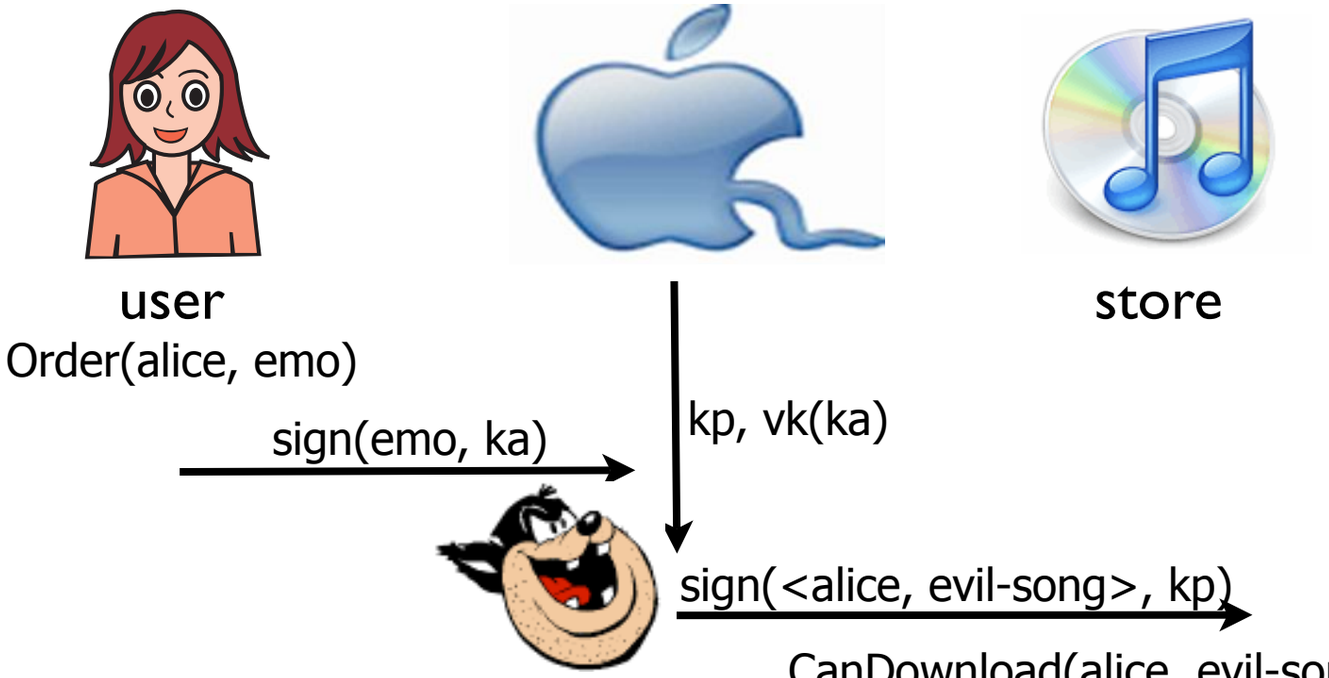
let store

- The protocol is not safe despite the compromise of the proxy

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy



let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

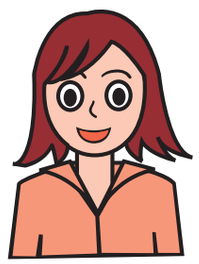
- The protocol is not safe despite the compromise of the proxy
- Still, the policy does not even talk about the proxy ... this is a bug

let store

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy



user

store

Order(alice, emo)

sign(emo, ka)

kp, vk(ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

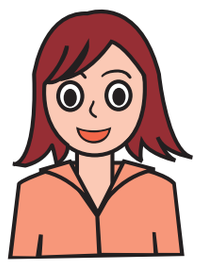
- The protocol is not safe despite the compromise of the proxy
- Still, the policy does not even talk about the proxy ... this is a bug
- Two ways to fix this:

let store

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy



user

store

Order(alice, emo)

sign(emo, ka)

kp, vk(ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

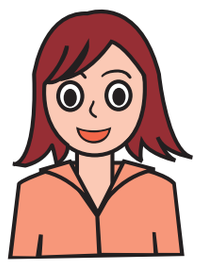
let store

- The protocol is not safe despite the compromise of the proxy
- Still, the policy does not even talk about the proxy ... this is a bug
- Two ways to fix this:
  - document the dependency by weakening the property and making the policy more explicit (what the authors do in this paper)

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy



user

store

Order(alice, emo)

sign(emo, ka)

kp, vk(ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

- The protocol is not safe despite the compromise of the proxy
- Still, the policy does not even talk about the proxy ... this is a bug
- Two ways to fix this:
  - document the dependency by weakening the property and making the policy more explicit (what the authors do in this paper)
  - strengthen the protocol (more about this at the end of the talk)

let store

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**out**(c, kp) | **out**(c, vk(ka))] | store | policy)

5 minutes break

Type system

Types

$T, U ::=$	Un	untrusted
	$\mathsf{Ch}(T)$	channel
	$\langle x_1 : T_1, \ldots, x_n : T_n \rangle \{C\}$	tuple (dependent)
	$\mathsf{PubKey}(T)$	encryption key
	$\mathsf{PrivKey}(T)$	decryption key
	$\mathsf{PubEnc}(T)$	encrypted message
	$\mathsf{SigKey}(T)$	signing key
	$\mathsf{VerKey}(T)$	verification key
	$\mathsf{Signed}(T)$	signed message
	$\mathsf{Hash}(T)$	hashed message

Types

$T, U ::=$	Un	untrusted
	$\mathsf{Ch}(T)$	channel
	$\langle x_1 : T_1, \ldots, x_n : T_n \rangle \{C\}$	tuple (dependent)
	$\mathsf{PubKey}(T)$	encryption key
	$\mathsf{PrivKey}(T)$	decryption key
	$\mathsf{PubEnc}(T)$	encrypted message
	$\mathsf{SigKey}(T)$	signing key
	$\mathsf{VerKey}(T)$	verification key
	$\mathsf{Signed}(T)$	signed message
	$\mathsf{Hash}(T)$	hashed message

- Typing environment
  - a list of type bindings $(u : T)$ and logical formulas C

Subtyping

- All messages sent over or received from the untrusted network have type Un
- For flexibility we introduce a special subtyping relation
  - $T <:$ Un $\Rightarrow T$ is **public**
    - its elements can be sent over the untrusted network
  - Un $<: T \Rightarrow T$ is **tainted**
    - elements cam be received from the untrusted network
- Subtyping = preorder on types (i.e. weak)

Typing terms

$$\textsc{Environment}\quad \frac{\Gamma \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T}$$

$$\textsc{Subsumption}\quad \frac{\Gamma \vdash M : T \qquad \Gamma \vdash T <: T'}{\Gamma \vdash M : T'}$$

$$\textsc{Tuple}\quad \frac{\forall i \in [1,n].\, \Gamma \vdash M_i : T_i \qquad forms(\Gamma) \models C\{M_i/x_i\}}{\Gamma \vdash \langle M_1, \ldots, M_n \rangle : \langle x_1 : T_1, \ldots, x_n : T_n \rangle \{C\}}$$

$$\textsc{Constructor}\quad \frac{f : (T_1, \ldots, T_n) \mapsto T \qquad \forall i \in [1,n].\, \Gamma \vdash M_i : T_i}{\Gamma \vdash f(M_1, \ldots, M_n) : T}$$

- Typing constructors

$$\mathsf{pk} : (\mathsf{PrivKey}(T)) \mapsto \mathsf{PubKey}(T) \qquad \mathsf{vk} : (\mathsf{SigKey}(T)) \mapsto \mathsf{VerKey}(T)$$

$$\mathsf{enc} : (T, \mathsf{PubKey}(T)) \mapsto \mathsf{PubEnc}(T) \qquad \mathsf{sign} : (T, \mathsf{SigKey}(T)) \mapsto \mathsf{Signed}(T)$$

$$\mathsf{hash} : (T) \mapsto \mathsf{Hash}(T)$$

Typing processes

$$\textsc{Output}\quad \frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \mathbf{out}(M, N).P}$$

$$\textsc{Input}\quad \frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma, x : T \vdash P}{\Gamma \vdash \mathbf{in}(M, x).P}$$

$$\textsc{New}\quad \frac{T \in \{\mathsf{Un}, \mathsf{Ch}(U), \mathsf{SigKey}(U), \mathsf{PrivKey}(U)\} \qquad \Gamma, a : T \vdash P}{\Gamma \vdash \mathbf{new}\ a : T.P}$$

$$\textsc{Replication}\quad \frac{\Gamma \vdash P}{\Gamma \vdash !P}$$

$$\textsc{Null}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}$$

Typing destructor applications

$$\textsc{Destructor}\quad \frac{g : (T_1, \ldots, T_n) \mapsto T \qquad \forall i \in [1, n].\, \Gamma \vdash M_i : T_i \qquad \Gamma, x : T \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{then}\ P\ \mathbf{else}\ Q}$$

- Typing destructors

$$\mathsf{dec} : (\mathsf{PubEnc}(T), \mathsf{PrivKey}(T)) \mapsto T$$
$$\mathsf{check} : (\mathsf{Signed}(T), \mathsf{VerKey}(T)) \mapsto T$$

Typing parallel compositions

$$\textsc{Parallel}\quad \frac{P \rightsquigarrow \Gamma_P \qquad \Gamma, \Gamma_P \vdash Q \qquad Q \rightsquigarrow \Gamma_Q \qquad \Gamma, \Gamma_Q \vdash P}{\Gamma \vdash P \mid Q}$$

- Environment extraction relation

$$\mathbf{assume}\ C \rightsquigarrow C$$

$$\frac{P \rightsquigarrow \Gamma_P}{\mathbf{new}\ a : T.P \rightsquigarrow a : T, \Gamma_P}$$

$$\frac{P \rightsquigarrow \Gamma_P \qquad Q \rightsquigarrow \Gamma_Q}{P \mid Q \rightsquigarrow \Gamma_P, \Gamma_Q}$$

$$\text{everything else} \rightsquigarrow \emptyset$$

Typing splits, assumes, asserts

SPLIT

$$\frac{\Gamma \vdash M : \langle y_1 : T_1, \ldots, y_n : T_n \rangle \{C\} \qquad \Gamma, x_1 : T_1, \ldots, x_n : T_n, \langle x_1, \ldots, x_n \rangle = M, C\{x_i/y_i\} \vdash P}{\Gamma \vdash \mathbf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathbf{in}\ P}$$

ASSUME

$$\frac{\Gamma, C \vdash \diamond}{\Gamma \vdash \mathbf{assume}\ C}$$

ASSERT

$$\frac{\Gamma \vdash \diamond \qquad \mathit{forms}(\Gamma) \models C}{\Gamma \vdash \mathbf{assert}\ C}$$

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process new ka : SigKey(<song:Un>{Order(alice, song)});
new kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)});
(user | proxy | store | policy)

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ |- **out**(c, sign(<emo>, ka))

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

$$\frac{\Gamma \vdash c : Ch(Un) \qquad \Gamma \vdash sign(\langle emo \rangle, ka) : Un}{\Gamma \vdash \textbf{out}(c, sign(\langle emo \rangle, ka))}$$

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

$$\dfrac{\dfrac{\Gamma \vdash c : Un \quad \Gamma \vdash Un <: Ch(Un)}{\Gamma \vdash c : Ch(Un)} \qquad \Gamma \vdash sign(<emo>, ka) : Un}{\Gamma \vdash \textbf{out}(c, sign(<emo>, ka))}$$

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

sign : $(T, \text{SigKey}(T)) \mapsto \text{Signed}(T)$

$$\dfrac{\dfrac{\Gamma \vdash c : Un \quad \Gamma \vdash Un <: Ch(Un)}{\Gamma \vdash c : Ch(Un)} \quad \dfrac{\Gamma \vdash sign(<emo>, ka) : Signed(T) \quad \Gamma \vdash Signed(T) <: Un}{\Gamma \vdash sign(<emo>, ka) : Un}}{\Gamma \vdash \textbf{out}(c, sign(<emo>, ka))}$$

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

$$\text{sign} : (T, \text{SigKey}(T)) \mapsto \text{Signed}(T)$$

$$\dfrac{\dfrac{\Gamma \vdash c : Un \quad \Gamma \vdash Un <: Ch(Un)}{\Gamma \vdash c : Ch(Un)} \quad \dfrac{\Gamma \vdash \text{sign}(<emo>, ka) : \text{Signed}(T) \quad \dfrac{T <: Un}{\Gamma \vdash \text{Signed}(T) <: Un}}{\Gamma \vdash \text{sign}(<emo>, ka) : Un}}{\Gamma \vdash \textbf{out}(c, \text{sign}(<emo>, ka))}$$

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

$$\text{sign} : (T, \text{SigKey}(T)) \mapsto \text{Signed}(T)$$

$$\dfrac{\dfrac{\Gamma \vdash c : Un \quad \Gamma \vdash Un <: Ch(Un)}{\Gamma \vdash c : Ch(Un)} \quad \dfrac{\dfrac{\Gamma \vdash \langle emo \rangle : T \quad \Gamma \vdash ka : SigKey(T)}{\Gamma \vdash sign(\langle emo \rangle, ka) : Signed(T)} \quad \dfrac{T <: Un}{\Gamma \vdash Signed(T) <: Un}}{\Gamma \vdash sign(\langle emo \rangle, ka) : Un}}{\Gamma \vdash \textbf{out}(c, sign(\langle emo \rangle, ka))}$$

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

T= <song:Un>{Order(alice, song)}

$$\dfrac{\dfrac{\Gamma \vdash c : Un \quad \Gamma \vdash Un <: Ch(Un)}{\Gamma \vdash c : Ch(Un)} \quad \dfrac{\dfrac{\Gamma \vdash \langle emo \rangle : T \quad \Gamma \vdash ka : SigKey(T)}{\Gamma \vdash sign(\langle emo \rangle, ka) : Signed(T)} \quad \dfrac{T <: Un \;\checkmark}{\Gamma \vdash Signed(T) <: Un}}{\Gamma \vdash sign(\langle emo \rangle, ka) : Un}}{\Gamma \vdash \mathbf{out}(c, sign(\langle emo \rangle, ka))}$$

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

T= <song:Un>{Order(alice, song)}

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash c : Un \quad \Gamma \vdash Un <: Ch(Un)}{\Gamma \vdash c : Ch(Un)} \quad \dfrac{\dfrac{\Gamma \vdash emo : Un \qquad forms(\Gamma) \models Order(alice, emo)}{\Gamma \vdash \langle emo \rangle : T} \quad \Gamma \vdash ka : SigKey(T)}{\Gamma \vdash sign(\langle emo \rangle, ka) : Signed(T)} \quad \dfrac{T <: Un}{\Gamma \vdash Signed(T) <: Un}}{\Gamma \vdash sign(\langle emo \rangle, ka) : Un}}{\Gamma \vdash \textbf{out}(c, sign(\langle emo \rangle, ka))}$$

✔

Type-checking the example

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

T= <song:Un>{Order(alice, song)}

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash c : Un \quad \Gamma \vdash Un <: Ch(Un)}{\Gamma \vdash c : Ch(Un)} \quad \dfrac{\dfrac{\Gamma \vdash emo : Un \qquad forms(\Gamma) \models Order(alice, emo)}{\Gamma \vdash \langle emo \rangle : T} \quad \Gamma \vdash ka : SigKey(T)}{\Gamma \vdash sign(\langle emo \rangle, ka) : Signed(T)} \quad \dfrac{T <: Un}{\Gamma \vdash Signed(T) <: Un}}{\Gamma \vdash sign(\langle emo \rangle, ka) : Un}}{\Gamma \vdash \textbf{out}(c, sign(\langle emo \rangle, ka))}$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s)).

process new ka : SigKey(<song:Un>{Order(alice, song)});
new kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)});
(user | proxy | store | policy)

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ |- **in**(c,m); **let** <s> = check(m, vk(ka)) **in out**(c, sign(<alice, s>, kp)).

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ |- c : Ch(Un)

Γ |- **in**(c,m); **let** <s> = check(m, vk(ka)) **in out**(c, sign(<alice, s>, kp)).

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

$$\frac{\Gamma \vdash c : Ch(Un) \qquad \Gamma' \vdash \textbf{let } \langle s \rangle = check(m, vk(ka)) \textbf{ in out}(c, sign(\langle alice, s \rangle, kp))}{\Gamma \vdash \textbf{in}(c,m); \textbf{let } \langle s \rangle = check(m, vk(ka)) \textbf{ in out}(c, sign(\langle alice, s \rangle, kp)).}$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, m : Un

$$\text{check} : (\text{Signed}(T), \text{VerKey}(T)) \mapsto T$$

$$\frac{\Gamma \vdash c : \text{Ch}(\text{Un}) \qquad \Gamma' \vdash \textbf{let } \langle s \rangle = \text{check}(m, \text{vk}(ka)) \textbf{ in out}(c, \text{sign}(\langle \text{alice}, s \rangle, kp))}{\Gamma \vdash \textbf{in}(c,m); \textbf{let } \langle s \rangle = \text{check}(m, \text{vk}(ka)) \textbf{ in out}(c, \text{sign}(\langle \text{alice}, s \rangle, kp)).}$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, m : Un

$$\text{check} : (\text{Signed}(T), \text{VerKey}(T)) \mapsto T$$

$$\frac{\Gamma' \vdash m : \text{Signed}(T) \quad \Gamma' \vdash vk(ka) : \text{VerKey}(T)}{\Gamma \vdash c : \text{Ch}(\text{Un}) \qquad \Gamma' \vdash \textbf{let } \langle s \rangle = \text{check}(m, vk(ka)) \textbf{ in out}(c, \text{sign}(\langle alice, s \rangle, kp))}$$

$$\Gamma \vdash \textbf{in}(c,m); \textbf{let } \langle s \rangle = \text{check}(m, vk(ka)) \textbf{ in out}(c, \text{sign}(\langle alice, s \rangle, kp)).$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

$$\mathsf{vk} : (\mathsf{SigKey}(T)) \mapsto \mathsf{VerKey}(T)$$

$$\frac{\Gamma' \text{ |- } m : \mathsf{Signed}(T) \qquad \dfrac{\Gamma' \text{ |- } ka : \mathsf{SigKey}(T)}{\Gamma' \text{ |- } \mathsf{vk}(ka) : \mathsf{VerKey}(T)}}{\Gamma \text{ |- } c : \mathsf{Ch}(\mathsf{Un}) \qquad \Gamma' \text{ |- } \textbf{let } \langle s \rangle = \mathsf{check}(m, \mathsf{vk}(ka)) \textbf{ in out}(c, \mathsf{sign}(\langle alice, s \rangle, kp))}$$

$$\Gamma \text{ |- } \textbf{in}(c,m); \textbf{let } \langle s \rangle = \mathsf{check}(m, \mathsf{vk}(ka)) \textbf{ in out}(c, \mathsf{sign}(\langle alice, s \rangle, kp)).$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

T = <song:Un>{Order(alice, song)}

$$\mathsf{vk} : (\mathsf{SigKey}(T)) \mapsto \mathsf{VerKey}(T)$$

$$\frac{\Gamma' \vdash m : \mathsf{Signed}(T) \quad \dfrac{\Gamma' \vdash ka : \mathsf{SigKey}(T)}{\Gamma' \vdash \mathsf{vk}(ka) : \mathsf{VerKey}(T)}}{\Gamma \vdash c : \mathsf{Ch}(\mathsf{Un}) \qquad \Gamma' \vdash \textbf{let } \langle s \rangle = \mathsf{check}(m, \mathsf{vk}(ka)) \textbf{ in out}(c, \mathsf{sign}(\langle alice, s \rangle, kp))}$$

$$\Gamma \vdash \textbf{in}(c,m); \textbf{let } \langle s \rangle = \mathsf{check}(m, \mathsf{vk}(ka)) \textbf{ in out}(c, \mathsf{sign}(\langle alice, s \rangle, kp)).$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

Γ'' = Γ', s : Un, Order(alice,s)

T = <song:Un>{Order(alice, song)}

$$\mathsf{vk} : (\mathsf{SigKey}(T)) \mapsto \mathsf{VerKey}(T)$$

$$\frac{\Gamma' \text{ |- m : Signed(T)} \quad \dfrac{\Gamma' \text{ |- ka : SigKey(T)}}{\Gamma' \text{ |- vk(ka) : VerKey(T)}} \qquad \Gamma'' \text{ |- } \mathbf{out}(c, \text{sign}(\langle alice, s\rangle, kp))}{\Gamma \text{ |- c : Ch(Un)} \qquad \Gamma' \text{ |- } \mathbf{let} \langle s\rangle = \text{check}(m, \text{vk}(ka)) \mathbf{\ in\ out}(c, \text{sign}(\langle alice, s\rangle, kp))}$$

$$\Gamma \text{ |- } \mathbf{in}(c,m); \mathbf{let} \langle s\rangle = \text{check}(m, \text{vk}(ka)) \mathbf{\ in\ out}(c, \text{sign}(\langle alice, s\rangle, kp)).$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

Γ'' = Γ', s : Un, Order(alice,s)

T = <song:Un>{Order(alice, song)}

$$\dfrac{\Gamma \vdash c : Ch(Un) \qquad \dfrac{\Gamma' \vdash m : Signed(T) \qquad \dfrac{\Gamma' \vdash ka : SigKey(T)}{\Gamma' \vdash vk(ka) : VerKey(T)} \qquad \dfrac{\Gamma'' \vdash sign(\langle alice, s\rangle, kp) : Signed(T') \quad T' <: Un}{\Gamma'' \vdash \mathbf{out}(c, sign(\langle alice, s\rangle, kp))}}{\Gamma' \vdash \mathbf{let}\ \langle s\rangle = check(m, vk(ka))\ \mathbf{in}\ \mathbf{out}(c, sign(\langle alice, s\rangle, kp))}}{\Gamma \vdash \mathbf{in}(c,m);\ \mathbf{let}\ \langle s\rangle = check(m, vk(ka))\ \mathbf{in}\ \mathbf{out}(c, sign(\langle alice, s\rangle, kp)).}$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

Γ'' = Γ', s : Un, Order(alice,s)

T = <song:Un>{Order(alice, song)}

sign : $(T', \text{SigKey}(T')) \mapsto \text{Signed}(T')$

$$\dfrac{\Gamma' \vdash m : \text{Signed}(T) \quad \dfrac{\Gamma' \vdash ka : \text{SigKey}(T)}{\Gamma' \vdash vk(ka) : \text{VerKey}(T)} \quad \dfrac{\Gamma'' \vdash \text{sign}(\langle alice, s\rangle, kp) : \text{Signed}(T') \quad T' <: Un}{\Gamma'' \vdash \textbf{out}(c, \text{sign}(\langle alice, s\rangle, kp))}}{\dfrac{\Gamma \vdash c : \text{Ch}(Un) \qquad \Gamma' \vdash \textbf{let} \langle s\rangle = \text{check}(m, vk(ka)) \textbf{ in out}(c, \text{sign}(\langle alice, s\rangle, kp))}{\Gamma \vdash \textbf{in}(c,m); \textbf{let} \langle s\rangle = \text{check}(m, vk(ka)) \textbf{ in out}(c, \text{sign}(\langle alice, s\rangle, kp)).}}$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, m : Un

T = <song:Un>{Order(alice, song)}

Γ″ = Γ′, s : Un, Order(alice,s)

sign : $(T', \mathsf{SigKey}(T')) \mapsto \mathsf{Signed}(T')$

$$\dfrac{\dfrac{\Gamma'' \vdash \text{<alice, s>}: T' \qquad \Gamma'' \vdash kp : \mathsf{SigKey}(T')}{\Gamma'' \vdash \mathsf{sign}(\text{<alice, s>}, kp) : \mathsf{Signed}(T') \quad T' <: \mathsf{Un}} \qquad \dfrac{\Gamma' \vdash ka : \mathsf{SigKey}(T)}{\Gamma' \vdash \mathsf{vk}(ka) : \mathsf{VerKey}(T)}}{\Gamma'' \vdash \mathbf{out}(c, \mathsf{sign}(\text{<alice, s>}, kp))} \qquad \Gamma' \vdash m : \mathsf{Signed}(T)$$

$$\dfrac{\Gamma \vdash c : \mathsf{Ch}(\mathsf{Un}) \qquad \Gamma' \vdash \mathbf{let}\ \text{<s>} = \mathsf{check}(m, \mathsf{vk}(ka))\ \mathbf{in}\ \mathbf{out}(c, \mathsf{sign}(\text{<alice, s>}, kp))}{\Gamma \vdash \mathbf{in}(c,m);\ \mathbf{let}\ \text{<s>} = \mathsf{check}(m, \mathsf{vk}(ka))\ \mathbf{in}\ \mathbf{out}(c, \mathsf{sign}(\text{<alice, s>}, kp)).}$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

Γ'' = Γ', s : Un, Order(alice,s)

T = <song:Un>{Order(alice, song)}

T' = <user:Un, song:Un>
{Order(user, song)
$\wedge$ Registered(user)}

$$\dfrac{\dfrac{\Gamma \vdash c : Ch(Un) \qquad \dfrac{\Gamma' \vdash m : Signed(T) \qquad \dfrac{\Gamma' \vdash ka : SigKey(T)}{\Gamma' \vdash vk(ka) : VerKey(T)}}{\Gamma' \vdash \textbf{let } \langle s \rangle = check(m, vk(ka)) \textbf{ in out}(c, sign(\langle alice, s \rangle, kp))}}{\Gamma \vdash \textbf{in}(c,m); \textbf{let } \langle s \rangle = check(m, vk(ka)) \textbf{ in out}(c, sign(\langle alice, s \rangle, kp))} \qquad \dfrac{\dfrac{\Gamma'' \vdash \langle alice, s \rangle : T' \qquad \Gamma'' \vdash kp : SigKey(T')}{\Gamma'' \vdash sign(\langle alice, s \rangle, kp) : Signed(T')} \quad T' <: Un}{\Gamma'' \vdash \textbf{out}(c, sign(\langle alice, s \rangle, kp))}}{\Gamma \vdash \textbf{in}(c,m); \textbf{let } \langle s \rangle = check(m, vk(ka)) \textbf{ in out}(c, sign(\langle alice, s \rangle, kp))}.$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

Γ'' = Γ', s : Un, Order(alice,s)

T = <song:Un>{Order(alice, song)}

T' = <user:Un, song:Un>
{Order(user, song)
$\wedge$ Registered(user)}

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'' \vdash alice : Un} \quad \dfrac{}{\Gamma'' \vdash s : Un} \quad \dfrac{}{forms(\Gamma'') \models Order(alice, s) \wedge Registered(alice)}}{\Gamma'' \vdash \langle alice, s\rangle : T'} \quad \dfrac{}{\Gamma'' \vdash kp : SigKey(T')}}{\Gamma'' \vdash sign(\langle alice, s\rangle, kp) : Signed(T')} \quad \dfrac{}{T' <: Un}}{\Gamma'' \vdash \mathbf{out}(c, sign(\langle alice, s\rangle, kp))}}{\Gamma' \vdash let\ \langle s\rangle = check(m, vk(ka))\ \mathbf{in}\ \mathbf{out}(c, sign(\langle alice, s\rangle, kp))}}{\Gamma \vdash \mathbf{in}(c,m);\ let\ \langle s\rangle = check(m, vk(ka))\ \mathbf{in}\ \mathbf{out}(c, sign(\langle alice, s\rangle, kp))}$$

$$\dfrac{\dfrac{}{\Gamma' \vdash m : Signed(T)} \quad \dfrac{\dfrac{}{\Gamma' \vdash ka : SigKey(T)}}{\Gamma' \vdash vk(ka) : VerKey(T)}}{\Gamma \vdash c : Ch(Un)}$$

Type-checking the proxy

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, m : Un

Γ'' = Γ', s : Un, Order(alice,s)

T = <song:Un>{Order(alice, song)}

T' = <user:Un, song:Un>
{Order(user, song)
$\wedge$ Registered(user)}

$$\dfrac{\dfrac{\Gamma'' \vdash \text{alice} : \text{Un} \quad \Gamma'' \vdash \text{s} : \text{Un} \quad \text{forms}(\Gamma'') \models \text{Order(alice, s)} \wedge \text{Registered(alice)} \;\checkmark \qquad \dfrac{\Gamma'' \vdash \langle \text{alice, s} \rangle : T' \qquad \Gamma'' \vdash \text{kp} : \text{SigKey}(T')}{\dfrac{\Gamma' \vdash \text{ka} : \text{SigKey}(T)}{\Gamma' \vdash \text{vk(ka)} : \text{VerKey}(T)} \quad \dfrac{\Gamma'' \vdash \text{sign}(\langle \text{alice, s} \rangle, \text{kp}) : \text{Signed}(T') \quad T' <: \text{Un}}{\Gamma'' \vdash \textbf{out}(\text{c}, \text{sign}(\langle \text{alice, s} \rangle, \text{kp}))}}}{\Gamma' \vdash \text{m} : \text{Signed}(T)}}{\Gamma \vdash \text{c} : \text{Ch(Un)} \qquad \Gamma' \vdash \textbf{let} \langle \text{s} \rangle = \text{check(m, vk(ka))} \textbf{ in out}(\text{c}, \text{sign}(\langle \text{alice, s} \rangle, \text{kp}))}$$

$$\Gamma \vdash \textbf{in}(\text{c,m}); \textbf{let} \langle \text{s} \rangle = \text{check(m, vk(ka))} \textbf{ in out}(\text{c}, \text{sign}(\langle \text{alice, s} \rangle, \text{kp})).$$

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process new ka : SigKey(<song:Un>{Order(alice, song)});
Γ |- **new** kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)});
(user | proxy | store | policy)

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ |- **in**(c, x); **let** <user, song> = check(x, vk(kp)) **in assert** CanDownload(user, song)

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, x : Un

$$\frac{\Gamma' \vdash \textbf{let } \langle user, song\rangle = check(x, vk(kp)) \textbf{ in assert } CanDownload(user, song)}{\Gamma \vdash \textbf{in}(c, x);\ \textbf{let } \langle user, song\rangle = check(x, vk(kp)) \textbf{ in assert } CanDownload(user, song)}$$

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, x : Un

$$\text{check} : (\text{Signed}(T), \text{VerKey}(T)) \mapsto T$$

$$\frac{\Gamma' \vdash \textbf{let } \langle \text{user}, \text{song} \rangle = \text{check}(x, \text{vk}(kp)) \textbf{ in assert } \text{CanDownload}(\text{user}, \text{song})}{\Gamma \vdash \textbf{in}(c, x); \textbf{ let } \langle \text{user}, \text{song} \rangle = \text{check}(x, \text{vk}(kp)) \textbf{ in assert } \text{CanDownload}(\text{user}, \text{song})}$$

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, x : Un

$$\frac{\Gamma' \vdash x : \mathrm{Signed}(T) \qquad \Gamma' \vdash \mathrm{vk}(kp) : \mathrm{VerKey}(T)}{\Gamma' \vdash \textbf{let } \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp)) \textbf{ in assert } \mathrm{CanDownload}(user, song)}$$

$$\frac{}{\Gamma \vdash \textbf{in}(c, x); \textbf{let } \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp)) \textbf{ in assert } \mathrm{CanDownload}(user, song)}$$

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, x : Un

T = <user:Un, song:Un>
{Order(user, song)
∧ Registered(user)}

$$\frac{\Gamma' \vdash x : \mathrm{Signed}(T) \qquad \Gamma' \vdash \mathrm{vk}(kp) : \mathrm{VerKey}(T)}{\Gamma' \vdash \textbf{let } \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp)) \textbf{ in assert } \mathrm{CanDownload}(user, song)}$$

$$\frac{}{\Gamma \vdash \textbf{in}(c, x); \textbf{ let } \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp)) \textbf{ in assert } \mathrm{CanDownload}(user, song)}$$

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, x : Un

Γ″ = Γ′, user : Un, song : Un,
Order(user,song) ∧ Registered(user)

T = <user:Un, song:Un>
{Order(user, song)
∧ Registered(user)}

$$\frac{\Gamma' \vdash x : \mathrm{Signed}(T) \qquad \Gamma' \vdash \mathrm{vk}(kp) : \mathrm{VerKey}(T) \qquad \Gamma'' \vdash \textbf{assert}\ \mathrm{CanDownload}(user, song)}{\Gamma' \vdash \textbf{let}\ \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp))\ \textbf{in assert}\ \mathrm{CanDownload}(user, song)}$$

$$\frac{\Gamma' \vdash \textbf{let}\ \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp))\ \textbf{in assert}\ \mathrm{CanDownload}(user, song)}{\Gamma \vdash \textbf{in}(c, x);\ \textbf{let}\ \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp))\ \textbf{in assert}\ \mathrm{CanDownload}(user, song)}$$

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) $\wedge$ Registered(user)})
Order(alice, emo), Registered(alice),
$\forall$ u, s. (Order(u, s) $\wedge$ Registered(u) $\Rightarrow$ CanDownload(u, s))

Γ' = Γ, x : Un

Γ'' = Γ', user : Un, song : Un,
Order(user,song) $\wedge$ Registered(user)

T = <user:Un, song:Un>
{Order(user, song)
$\wedge$ Registered(user)}

$$\frac{\Gamma' \vdash x : \mathrm{Signed}(T) \qquad \Gamma' \vdash \mathrm{vk}(kp) : \mathrm{VerKey}(T) \qquad \dfrac{\mathrm{forms}(\Gamma'') \models \mathrm{CanDownload}(user, song)}{\Gamma'' \vdash \textbf{assert}\ \mathrm{CanDownload}(user, song)}}{\Gamma' \vdash \textbf{let}\ \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp))\ \textbf{in assert}\ \mathrm{CanDownload}(user, song)}$$

$$\Gamma \vdash \textbf{in}(c, x);\ \textbf{let}\ \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp))\ \textbf{in assert}\ \mathrm{CanDownload}(user, song)$$

Type-checking the store

Γ = c : Un, emo : Un, alice : Un, ka : SigKey(<song:Un>{Order(alice, song)}),
kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)})
Order(alice, emo), Registered(alice),
∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s))

Γ′ = Γ, x : Un

Γ″ = Γ′, user : Un, song : Un,
Order(user,song) ∧ Registered(user)

T = <user:Un, song:Un>
{Order(user, song)
∧ Registered(user)}

$$\dfrac{\Gamma' \vdash x : \mathrm{Signed}(T) \qquad \Gamma' \vdash \mathrm{vk}(kp) : \mathrm{VerKey}(T) \qquad \dfrac{\mathrm{forms}(\Gamma'') \models \mathrm{CanDownload}(user, song)\ \checkmark}{\Gamma'' \vdash \textbf{assert}\ \mathrm{CanDownload}(user, song)}}{\Gamma' \vdash \textbf{let}\ \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp))\ \textbf{in assert}\ \mathrm{CanDownload}(user, song)}$$

$$\Gamma \vdash \textbf{in}(c, x);\ \textbf{let}\ \langle user, song\rangle = \mathrm{check}(x, \mathrm{vk}(kp))\ \textbf{in assert}\ \mathrm{CanDownload}(user, song)$$

Type-checking done ✔

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).


✔

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).


✔

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).


✔

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process new ka : SigKey(<song:Un>{Order(alice, song)});
new kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)});
(user | proxy | store | policy)

Type-checking done ✔

- The type system enforces robust safety (we will prove it soon)
- So successfully type-checking the protocol means it is **robustly safe**

let user = **assume** Order(alice, emo) |
out(c, sign(<emo>, ka)).

✔

let proxy = **assume** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

✔

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

✔

let policy = **assume** ∀ u, s. (Order(u, s) ∧ Registered(u) ⇒ CanDownload(u, s)).

process new ka : SigKey(<song:Un>{Order(alice, song)});
new kp : SigKey(<user:Un, song:Un>{Order(user, song) ∧ Registered(user)});
(user | proxy | store | policy)

Illustrating the proof technique

(on the board)

Proving safety

- Theorem (Subject-reduction)

  1. If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$;
  2. If $\Gamma \vdash P$ and $P \rightarrow Q$ then $\Gamma \vdash Q$.

- Lemma (Correct Environment Extraction)

  If $Q \rightsquigarrow \Gamma_Q$ then $\exists \mathcal{E} = \mathbf{new}\ b_1 : U_1 \ldots \mathbf{new}\ b_m : U_m.\ [\,] \mid Q'$ s.t. $\{b_1, \ldots, b_m\} = dom(\Gamma_Q)$, $Q \equiv \mathcal{E}[\mathbf{assume}\ C_1 \mid \ldots \mid \mathbf{assume}\ C_n]$, and $forms(\Gamma_Q) = \{C_1, \ldots, C_n\}$.

- Theorem (Safety)

  If $\vdash_{\mathsf{Un}} P$ then P is safe.

  ($\vdash_{\mathsf{Un}} P$ denotes $u_1 : \mathsf{Un}, \ldots, u_n : \mathsf{Un} \vdash P$ for $u_i \in free(P)$)

Proving robust safety

- Theorem (Safety)

  If $\vdash_{\mathsf{Un}} P$ then P is safe.

- Lemma (Universal Type)

  If $\Gamma \vdash \diamond$ then $\Gamma \vdash \mathsf{Un} <:> T$ for all

  $T \in \{\mathsf{Ch}(\mathsf{Un}), \langle x_1 : \mathsf{Un}, \ldots, x_n : \mathsf{Un}\rangle\{\mathsf{true}\}, \mathsf{SigKey}(\mathsf{Un}), \mathsf{VerKey}(\mathsf{Un}),$
  $\mathsf{Signed}(\mathsf{Un}), \mathsf{PubKey}(\mathsf{Un}), \mathsf{PrivKey}(\mathsf{Un}), \mathsf{PubEnc}(\mathsf{Un}), \mathsf{Hash}(\mathsf{Un})\}$

- Lemma (Term Un-typability) $\forall M.\ \vdash_{\mathsf{Un}} M : \mathsf{Un}$
- Lemma (Opponent Typability)

  For all opponents O we have $\vdash_{\mathsf{Un}} O$.

- Theorem (Robust Safety).

  If $\vdash_{\mathsf{Un}} P$ then P is robustly safe.

Safety despite compromised participants

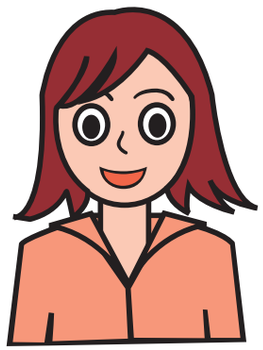
(slides I needed to skip)

Informal principle

- An invalid authorization decision by an uncompromised participant should only arise if participants on which the decision logically depends are compromised.
- In general, we check conformance of code to a logical policy, so that we can focus security reviews on the policy
- Hence, the impact of partial compromise should be apparent from the policy, without study of the code

Expressing protocol participants

- Extend the authorization logic with **says** operator



user

Order(alice, emo)

proxy

Registered(alice)

store

sign(emo, ka)

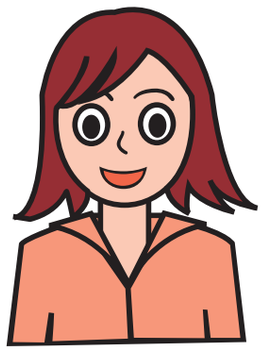
sign(<alice, emo>, kp)

CanDownload(alice, emo)?

policy = ∀user,∀song(Order(user, song) ∧ Registered(user) ⇒ CanDownload(user, song))

Expressing protocol participants

- Extend the authorization logic with **says** operator



user

proxy

store

alice **says** Order(emo)

proxy **says** Registered(alice)

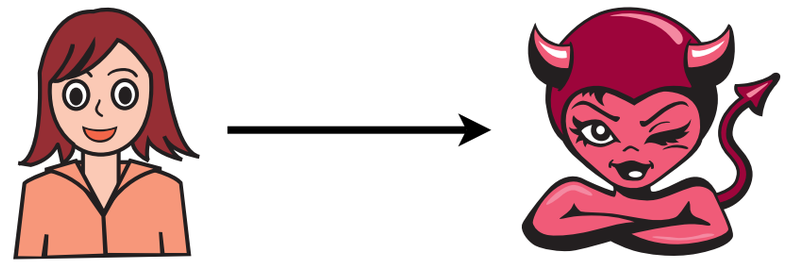
sign(emo, ka) →

sign(<alice, emo>, kp) →

CanDownload(alice, emo)?

policy = ∀user,∀song(user **says** Order(song) ∧ proxy **says** Registered(user) ⇒ CanDownload(user, song))

Safety despite compromised participants



- Definition

  Let $\mathcal{E} = \mathbf{new}\ c_1 : T_1 \ldots \mathbf{new}\ c_m : T_m.\ [\,] \mid R$
  and $P_{b_1}, \ldots, P_{b_n}$ be n protocol participants
  (P_{b_i} only contains formulas of the form b_i says C).

  $\mathcal{E}[P_{b_1} \mid \ldots \mid P_{b_n}]$ is safe despite $\{b_1, \ldots, b_n\}$ iff $\exists Q', \sigma$

  (1) $P_{b_1} \mid \ldots \mid P_{b_n} = Q'\sigma$ with $fn(Q') \cap \{c_1, \ldots, c_n\} = \emptyset$
  (2) **assume** $\wedge_{b \in \{b_1,\ldots,b_n\}}\ b$ says **false** $\mid\ \mathcal{E}[\prod_{x \in dom(\sigma)} \mathbf{out}(c, x\sigma)]$
  is robustly safe (where $c \notin \{c_1, \ldots, c_n\}$)

- The type system can also enforce this property
  - Basically all we need to check is that all variables in the domain of σ can be given type Un

Compromising the user



user
alice **says** Order(alice, emo)

proxy
proxy **says** Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** alice **says** Order(emo) |
out(c, sign(<emo>, ka)).

let proxy = **assume** proxy **says** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s.(u **says** Order(s) $\wedge$ proxy **says** Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
[user] | proxy | store | policy

Compromising the user

alice **says false**

ka

proxy

proxy **says** Registered(alice)

store

```
let proxy = assume proxy says Registered(alice) |
            in(c,m);
            let <s> = check(m, vk(ka)) in
            out(c, sign(<alice, s>, kp)).

let store = in(c, x);
            let <user, song> = check(x, vk(kp)) in
            assert CanDownload(user, song).

let policy = assume ∀ u, s.(u says Order(s) ∧ proxy says Registered(u) ⇒ CanDownload(u, s)).


process  new ka; new kp;
            assume alice says false | [out(c, ku)] | proxy | store | policy
```

Compromising the user

alice **says false**

ka

proxy

proxy **says** Registered(alice)

store

sign(evil-song, ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

```
let proxy = assume proxy says Registered(alice) |
            in(c,m);
            let <s> = check(m, vk(ka)) in
            out(c, sign(<alice, s>, kp)).

let store = in(c, x);
            let <user, song> = check(x, vk(kp)) in
            assert CanDownload(user, song).

let policy = assume ∀ u, s.(u says Order(s) ∧ proxy says Registered(u) ⇒ CanDownload(u, s)).


process  new ka; new kp;
            assume alice says false | [out(c, ku)] | proxy | store | policy
```

Compromising the user

alice **says false**

ka

proxy

proxy **says** Registered(alice)

store

sign(evil-song, ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let proxy = **assume** proxy **says** Registered(alice).|
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

With the new definition the system is safe despite the compromise of the user

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s.(u **says** Order(s) ∧ proxy **says** Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
assume alice **says false** | [**out**(c, ku)] | proxy | store | policy

Compromising the user

alice **says false**

ka

proxy

proxy **says** Registered(alice)

store

sign(evil-song, ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let proxy = **assume** proxy **says** Registered(alice).|
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

- With the new definition the system is safe despite the compromise of the user
- All assertions only depending on alice saying something will hold (since alice says false)

let policy = **assume** ∀ u, s.(u **says** Order(s) ∧ proxy **says** Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
assume alice **says false** | [**out**(c, ku)] | proxy | store | policy

Compromising the user

alice **says false**

ka

proxy

proxy **says** Registered(alice)

store

sign(evil-song, ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let proxy = **assume** proxy **says** Registered(alice).
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

- With the new definition the system is safe despite the compromise of the user
- All assertions only depending on alice saying something will hold (since alice says false)
- This is **not** a problem

let policy = **assume** ∀ u, s.(u **says** Order(s) ∧ proxy **says** Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
assume alice **says false** | [**out**(c, ku)] | proxy | store | policy

Compromising the proxy



user
alice **says** Order(alice, emo)

proxy
proxy **says** Registered(alice)

store

sign(emo, ka)

sign(<alice, emo>, kp)

CanDownload(alice, emo)?

let user = **assume** alice **says** Order(emo) |
out(c, sign(<emo>, ka)).

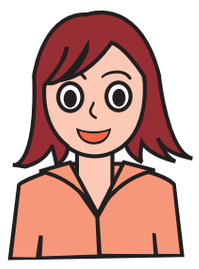
let proxy = **assume** proxy **says** Registered(alice) |
in(c,m);
let <s> = check(m, vk(ka)) **in**
out(c, sign(<alice, s>, kp)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** $\forall$ u, s.(u **says** Order(s) $\wedge$ proxy **says** Registered(u) $\Rightarrow$ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | proxy | store | policy)

Compromising the proxy



user

alice **says** Order(alice, emo)

sign(emo, ka)

proxy **says false**

kp, vk(ka)

store

let user = **assume** alice **says** Order(emo) |
out(c, sign(<emo>, ka)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s.(u **says** Order(s) ∧ proxy **says** Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**assume** proxy **says false** | **out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy

user

alice **says** Order(alice, emo)

proxy **says false**

store

sign(emo, ka)

kp, vk(ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let user = **assume** alice **says** Order(emo) |
out(c, sign(<emo>, ka)).

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s.(u **says** Order(s) ∧ proxy **says** Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**assume** proxy **says false** | **out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy

user

alice **says** Order(alice, emo)

proxy **says false**

store

sign(emo, ka)

kp, vk(ka)

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let user = **assume** alice **says** Order(emo) |
out(c, sign(<emo>, ka)).

- The system is still **not** safe despite the compromise of the proxy

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s.(u **says** Order(s) ∧ proxy **says** Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**assume** proxy **says false** | **out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Compromising the proxy

user

alice **says** Order(alice, emo)

sign(emo, ka)

proxy **says false**

kp, vk(ka)

store

sign(<alice, evil-song>, kp)

CanDownload(alice, evil-song)?

let user = **assume** alice **says** Order(emo) |
out(c, sign(<emo>, ka)).

- The system is still **not** safe despite the compromise of the proxy
- These are the kind of bugs we are actually looking for

let store = **in**(c, x);
let <user, song> = check(x, vk(kp)) **in**
assert CanDownload(user, song).

let policy = **assume** ∀ u, s.(u **says** Order(s) ∧ proxy **says** Registered(u) ⇒ CanDownload(u, s)).

process **new** ka; **new** kp;
(user | [**assume** proxy **says false** | **out**(c, kp) | **out**(c, vk(ka))] | store | policy)

Making the policy more explicit

- To fix the undocumented dependency on the proxy we can change the policy
  - policy = ∀user,∀song(proxy **controls** (user **says** Order(song)) ∧ proxy **says** Registered(user) ⇒ CanDownload(user, song))

    where "a **controls** C" means "(a **says** C) ⇒ C"

- With respect to this policy the protocol is safe despite the compromise of the proxy
- But this doesn't make the protocol any safer
  - it only documents all the implications of compromising the proxy

Related and future work