A Type Discipline for Authorization in Distributed Systems
How can we check statically whether implementation code for a distributed system correctly implements an authorization (i.e., access control) policy?

- We assume that security-related events are annotated in the code
- We assume the policy is expressed in some authorization logic

Our approach (begun at ESOP’05) is to represent code in a process calculus with formal cryptography, and to generalize prior type and effect systems for authentication protocols.

Here, we develop our security types in two ways:
- We allow for compromised principals
- We support a wider range of cryptographic operations
Authorization by Typing

Our system from ESOP’05 but recast in a more generic form
## Ordering a Song by Proxy

<table>
<thead>
<tr>
<th>Principals</th>
<th>$s(\text{tore}), u(\text{ser}), p(\text{roxy})$</th>
</tr>
</thead>
</table>
| Static Policies | $(D) \ \forall v, t. (p \ \textbf{says} \ \text{Registered}(v) \land v \ \textbf{says} \ \text{Order}(t)) \rightarrow \text{CanDownload}(v, t)$  
 $(Ru) \ p \ \textbf{says} \ \text{Registered}(u)$ |
| Annotated Protocol | Event $u \ \textbf{says} \ \text{Order}(\text{song})$  
 Message 1 $u \rightarrow p : \text{senc}(\text{song}, k_{up})$  
 Message 2 $p \rightarrow s : \text{sign}(\langle u, \text{song} \rangle, k_p)$  
 Expectation $s \ \textbf{says} \ \text{CanDownload}(u, \text{song})$ |

- Correctness of authorization formulated as **robust safety**:  
  - that in all runs,  
  - in presence of opponent in control of network,  
  - all expectations are deducible from prior events plus static policies
Types for the Example

- Formulas, $C$, in policies and types are drawn from a given authorization logic, a parameter of our type system
  - We use Abadi’s CDD as our example logic

- A value of type $\text{Un}$ is data known to the opponent
- A value of type $\langle x_1:T_1,\ldots,x_n:T_n\rangle\{C\}$ is a tuple, satisfying the formula $C$, which is in the scope of $x_1,\ldots,x_n$
- A value of type $\text{Key } T$ is a symmetric key for $T$ plaintexts
- A value of type $\text{SK } T$ is a signing key for $T$ plaintexts

- The keys in our example are of these types:
  \[
k_{up} : T_{up} = \text{Key } \langle t:\text{Un}\rangle\{u \text{ says } \text{Order}(t)\}
  \]
  \[
k_p : T_p = \text{SK } \langle v:\text{Un}, t:\text{Un}\rangle\{v \text{ says } \text{Order}(t), p \text{ says } \text{Registered}(v)\}
  \]
The Example as a Typed Process

\[ W = S | \]
\[ \text{new } k_{up} : \text{Key} \; \langle t : \text{Un} \rangle \{ u \; \text{says} \; Order(t) \}; \]
\[ \text{new } k_p : \text{SK} \; \langle v : \text{Un}, t : \text{Un} \rangle \{ \nu \; \text{says} \; Order(t), \; p \; \text{says} \; \text{Registered}(v) \}; \]
\[ \text{export } v_p = \text{vk}(k_p); \]
\[ u[Q_u] \; | \; p[Q_p] \; | \; s[Q_s] \]

\[ S = (D) \land (Ru) \]

\[ Q_u = \text{Order(georgiaOnMyMind)} \; | \]
\[ \text{out net(senc(\langle georgiaOnMyMind, k_{up} \rangle))} \]

\[ Q_p = \text{in net(cipher)}; \]
\[ \text{let } \langle \text{song} \rangle = \text{sdec(cipher, } k_{up} \rangle) \; \text{in} \]
\[ \text{out request(sign(} \langle u, \text{song} \rangle, k_p \rangle)) \]

\[ Q_s = !\text{in request(sig)}; \]
\[ \text{let } \langle \text{user, song} \rangle = \text{verify(sig, v_p)} \; \text{in} \]
\[ \text{expect CanDownload(user, song)} \]

We use a type and effect system, where effects are logical formulas. See paper for details of typing rules.
**Theorem 4 (Part 1 of 2)**

Suppose $W = S \mid \text{new} \, \sim c : T_c; \text{export} \, z = M_z; \prod_{a \in A} a[Q_a]$. Let $E$ be the environment that maps $\text{fn}(W)$ to $\text{Un}$. $W$ is robustly safe if

1. $E, S, c : T_c \vdash M_z : \text{Un}$ for each exported variable $z$;
2. $E, S, c : T_c, z : \text{Un} \vdash a[Q_a]$ for each principal $a \in A$.

Consider $W$ our running example.
Let $E$ map the free names of $W$ to $\text{Un}$.
Let $E' = E, (D) \land (Ru), k_{up} : T_{up}, k_p : T_p$.

1. For our single exported term, we have $E' \vdash \text{vk}(k_p) : \text{Un}$.
2. For each principal $a \in \{u, p, s\}$, we have $E', v_p : \text{Un} \vdash a[Q_a]$. 
Allowing for Compromise

A realistic threat model must include partial compromise, that some principals are compromised, i.e., their secrets are known to the attacker.
For example, suppose the user $u$ has ordered no songs. Partial compromise can lead to the following attacks:

1. If $u$ but not $p$ is compromised, the attacker knows $k_{up}$ and can fake Message 1, causing $p$ to send Message 2.
2. If $p$ but not $u$ is compromised, the attacker knows $k_{up}$ and $k_{p}$, and can fake Message 2 directly.

Both are failures of robust safety, but they are different:

- Unavoidably, if authorization depends on a compromised principal, invalid decisions may arise; (1) is an example.
- Arguably, an implementation is faulty if it reaches invalid conclusions that do not logically depend on compromised principals; (2) is an example.
  - The invalid conclusion $u$ says Order(song) does not depend on $p$. 
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.

To this end, we propose the informal principle:

An invalid authorization decision by an uncompromised principal should only arise if principals on which the decision logically depends are compromised.
Let a system $W$ be **safe despite compromised principals** iff for every $B \subseteq A$, $W$ is robustly safe when for each $b \in B$

- the principal $b$ is compromised, i.e., its (intended) secrets are public,
- we augment the policy with $b$ *says* false, that is, $\forall X. b \text{ says } X$

**Case** $B = \{u\}$ corresponds to attack (1), which we discount:

- Robust safety holds if $k_{up}$ is public but we assume $u$ *says* false

**Case** $B = \{p\}$ corresponds to attack (2), which is a fault:

- Robust safety fails if $k_{up}$ and $k_p$ is public but we assume $p$ *says* false

To fix the undocumented dependency on $p$, we add policy:

$$\text{(Op)} \ \forall v,t. s\text{ says } (p\ \text{controls}\ (v\ \text{says } \text{Order}(t)))$$

where $a\ \text{controls}\ C$ is short for $(a\ \text{says}\ C) \rightarrow C$

- Process $(\text{Op})|W$ is safe despite compromised principals.
The names $c_i$ are all the secrets of the principals $a \in A$

Each $x_\sigma$ represents a credential or capability of principals $b \in B$; typically $x_\sigma$ is, or derives from, one of the names $c_i$

Point (1) decomposes the compromised principals $B$ into pure code $Q'$, plus the credentials of $B$ in the substitution $\sigma$

Point (2) requires the credentials of $B$ to be public, given the assumption that $b$ says false for all $b \in B$
Definition 4: A generic definition of Safety Despite Compromise usable in any process calculus with restriction and parallel composition

A generic rule for type-checking destructor applications, driven by a symbolic operational semantics
- Subsumes various specific rules in the ESOP‘05 paper
- Relies on using an applied pi calculus, with cryptographic expressions built from constructors and destructors

We specifically describe rules for symmetric encryption, public key signatures, and keyed hashes
- We believe our framework would smoothly handle other primitives
Partial compromise often modelled by assuming the attacker is a recognised principal (Lowe,...); often, invariants are explicitly adjusted to account for compromise, e.g.

\[
\text{expect (s says CanDownload}(u, \text{song})) \lor \text{Bad}(u) \lor \text{Bad}(p)
\]

Our notion of SDCP seeks to generalize such adjustments systematically; invariants are left as intended, and implementation dependencies need to be explicitly noted.

Still, we assume a fixed population of principals – we have yet to deal with dynamic creation and subsequent compromise (as in eg ProVerif) in any generality.

See the paper for a full discussion of related work.
Critique – Future Work

- “You develop a theory and type system for a formal calculus rather than proper programming language.”
- Yes, but a great many programming constructs can be understood as message passing between processes; so, our theory of typed message passing is potentially widely applicable. To test this, we are building a typechecker for F# based on derived rules obtained by a CPS transform to pi.

- “You don’t show many actual examples.”
- True. But see previous answer. We have a body of security sensitive F# code; we intend to check security properties of this code by type checking.
The impact of partial compromise should be apparent from the authorization policy, without study of the code.

To this end, we propose and formalize the principle:

An invalid authorization decision by an uncompromised principal should only arise if principals on which the decision logically depends are compromised.
The End