

Types for Cryptographic Protocols

Christian Haack (on joint work with Alan Jeffrey and previous work of Gordon and Jeffrey)

September 20, 2007

Verifying Cryptographic Protocols

Problem.

- Verifying secrecy and authenticity in the Dolev–Yao model.

Verifying Cryptographic Protocols

Problem.

- Verifying secrecy and authenticity in the Dolev–Yao model.

Verification techniques applied to this.

- logics for authentication (BAN logic)
- fully automatic techniques:
 - model-checking (e.g. Caspar, Scyther)
 - automatic theorem proving (e.g. ProVerif)
 - control flow analysis (e.g. LySa)
- interactive theorem proving (e.g. Isabelle)
- type systems (e.g. Abadi/Blanchet, Gordon/Jeffrey)

Type Systems

Implicitly typed systems.

- strong secrecy (non-interference) for symmetric crypto (Abadi [Aba99])
- secrecy for asymmetric crypto (Abadi/Blanchet [AB01])
- translation of type inference problem to Horn clauses ([AB02])

Explicitly typed systems.

- authenticity for symmetric crypto (Gordon/Jeffrey [GJ01, GJ02b])
- authenticity for asymmetric crypto ([GJ02a])
- secrecy and authenticity for nested encryption, hashing, keyed hashing (Haack/Jeffrey [HJ06])
- short-term keys and timeliness ([HJ05])

Type Systems

Implicitly typed systems.

- strong secrecy (non-interference) for symmetric crypto (Abadi [Aba99])
- secrecy for asymmetric crypto (Abadi/Blanchet [AB01])
- translation of type inference problem to Horn clauses ([AB02])

Explicitly typed systems.

- authenticity for symmetric crypto (Gordon/Jeffrey [GJ01, GJ02b])
- authenticity for asymmetric crypto ([GJ02a])
- secrecy and authenticity for nested encryption, hashing, keyed hashing (Haack/Jeffrey [HJ06])
- short-term keys and timeliness ([HJ05])

I'll focus on the explicitly typed systems.

Informal Protocol Narrations

$$A \rightarrow B : (M, A)$$
$$B \rightarrow A : N$$
$$A \rightarrow B : \{\#\!(M, B, N)\!\}_{sA}$$

- This describes the message sequence of a regular protocol run.
- Informal narrations are no good for formal analysis.

The Spi-calculus

Messages.

$$M, N, L, K ::= x \mid n \mid \dots$$

Processes.

$$P, Q, O ::= \text{stop} \mid \text{out } L M; P \mid \text{inp } L x; P \mid \text{new } n; P \\ \mid \text{if } M = N \text{ then } P \mid P \mid Q \mid !P$$

The Spi-calculus

Messages.

$$M, N, L, K ::= x \mid n \mid \dots$$

Processes.

$$P, Q, O ::= \text{stop} \mid \text{out } L M; P \mid \text{inp } L x; P \mid \text{new } n; P \\ \mid \text{if } M = N \text{ then } P \mid P \mid Q \mid !P$$

Reduction semantics $P \rightarrow P'$.

$$(\text{out } L M; P) \mid (\text{inp } L x; Q) \rightarrow P \mid Q\{M/x\}$$

The Spi-calculus: Example

An informal narration of a one-message protocol.

$$A \rightarrow B : s$$

As a formal narration in spi.

$$\begin{aligned} P_A &\triangleq \text{new } s; \text{ out } net\ s; \text{ stop} \\ P_B &\triangleq \text{inp } net\ x; \text{ stop} \\ P &\triangleq !P_A \mid !P_B \end{aligned}$$

Message Constructors

Message constructors.

$$M ::= \dots \mid () \mid (M_1, M_2) \mid \{M\}_K \mid \#(M) \\ \mid \{\!|M|\!\}_K \mid \text{Enc}(K) \mid \text{Dec}(K)$$

Message Constructors

Message constructors.

$$M ::= \dots \mid () \mid (M_1, M_2) \mid \{M\}_K \mid \#(M) \\ \mid \{\{M\}\}_K \mid \text{Enc}(K) \mid \text{Dec}(K)$$

Message destructors.

$$P ::= \dots \mid \text{split } M \text{ is } (x_1, x_2); P \mid \text{decrypt } M \text{ is } \{x\}_K; P \\ \mid \text{decrypt } M \text{ is } \{\{x\}\}_{K^{-1}}; P$$

Message Constructors

Message constructors.

$$M ::= \dots \mid () \mid (M_1, M_2) \mid \{M\}_K \mid \#(M) \\ \mid \{\!\{M\}\!\}_K \mid \text{Enc}(K) \mid \text{Dec}(K)$$

Message destructors.

$$P ::= \dots \mid \text{split } M \text{ is } (x_1, x_2); P \mid \text{decrypt } M \text{ is } \{x\}_K; P \\ \mid \text{decrypt } M \text{ is } \{\!\{x\}\!\}_{K^{-1}}; P$$

Reduction rules.

$$\begin{aligned} \text{split } (M_1, M_2) \text{ is } (x_1, x_2); P &\rightarrow P\{M_1, M_2/x_1, x_2\} \\ \text{decrypt } \{M\}_K \text{ is } \{x\}_K; P &\rightarrow P\{M/x\} \\ \text{decrypt } \{\!\{M\}\!\}_{\text{Enc}(K)} \text{ is } \{\!\{x\}\!\}_{\text{Dec}(K)^{-1}}; P &\rightarrow P\{M/x\} \end{aligned}$$

Specifying Secrecy

Specifying Secrecy

Secrecy claims.

$P ::= \dots \mid \text{secret}(n)$ (“ n must remain secret”)

Specifying Secrecy

Secrecy claims.

$P ::= \dots \mid \text{secret}(n)$ (“ n must remain secret”)

- For example:

$A \rightarrow B : s$

$P_A \triangleq \text{new } s; (\text{secret}(s) \mid \text{out } net\ s; \text{stop})$

$P_B \triangleq \text{inp } net\ x; \text{stop}$

$P \triangleq !P_A \mid !P_B$

- The secrecy assertion in this example is obviously violated.

Safety for Secrecy

Definition (Safety for Secrecy).

A closed process P is **safe for secrecy** iff $P \rightarrow^* Q$ implies that Q does not have the following form:

- $Q = \text{new } \vec{k}; (\text{secret}(k) \mid \text{out } n \ k; Q' \mid Q'')$ where $n \notin \vec{k}$.

Safety for Secrecy

Definition (Safety for Secrecy).

A closed process P is **safe for secrecy** iff $P \rightarrow^* Q$ implies that Q does not have the following form:

- $Q = \text{new } \vec{k}; (\text{secret}(k) \mid \text{out } n \ k; Q' \mid Q'')$ where $n \notin \vec{k}$.

$$A \rightarrow B : s$$
$$P_A \triangleq \text{new } s; (\text{secret}(s) \mid \text{out } net \ s; \text{stop})$$
$$P_B \triangleq \text{inp } net \ x; \text{stop}$$
$$P \triangleq !P_A \mid !P_B$$

- This protocol is **not** safe for secrecy:
- $P \equiv P_A \mid P \equiv \text{new } s; (\text{secret}(s) \mid \text{out } net \ s; \text{stop} \mid P)$

Robust Safety for Secrecy

- What we are really interested in is **safety in the presence of attackers**.
- **Intuitively**: A process is **robustly safe for secrecy** if no opponent can trick it into publishing any of its secrets or any data from which he can compute any of its secrets.

Robust Safety for Secrecy

- What we are really interested in is **safety in the presence of attackers**.
- **Intuitively**: A process is **robustly safe for secrecy** if no opponent can trick it into publishing any of its secrets or any data from which he can compute any of its secrets.

Definition (Opponent Processes).

An **opponent process** is a closed process that does not contain secrecy assertions.

Robust Safety for Secrecy

- What we are really interested in is **safety in the presence of attackers**.
- **Intuitively**: A process is **robustly safe for secrecy** if no opponent can trick it into publishing any of its secrets or any data from which he can compute any of its secrets.

Definition (Opponent Processes).

An **opponent process** is a closed process that does not contain secrecy assertions.

Definition (Robust Safety for Secrecy).

A closed process P is **robustly safe for secrecy** iff for all opponent processes O the parallel composition $P \mid O$ is safe for secrecy.

Example: A Violation of Robust Safety

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

This protocol is not robustly safe!

Example: A Violation of Robust Safety

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

This protocol is not robustly safe!

An attack:

$A \rightarrow O : (\{s, p\}_k, \{p', s'\}_k) \quad // \text{opponent intercepts}$

Example: A Violation of Robust Safety

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

This protocol is not robustly safe!

An attack:

$A \rightarrow O : (\{s, p\}_k, \{p', s'\}_k) \quad // \text{opponent intercepts}$

$O \rightarrow B : (\{p', s'\}_k, \{s, p\}_k) \quad // \text{opponent swaps ciphers}$

Example: A Violation of Robust Safety

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

This protocol is not robustly safe!

An attack:

$A \rightarrow O : (\{s, p\}_k, \{p', s'\}_k)$ // opponent intercepts

$O \rightarrow B : (\{p', s'\}_k, \{s, p\}_k)$ // opponent swaps ciphers

$B \rightarrow A : (s', s)$ // B publishes secrets

Example: A Violation of Robust Safety

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

This protocol is not robustly safe!

As a formal narration:

$P_A \triangleq$ new s ; new p ; new p' ; new s' ;
(secret(s) | secret(s') | out net ($\{(s, p)\}_k, \{(p', s')\}_k$); |
inp net x ; if $x = (p, p')$ then stop)

$P_B \triangleq$ inp net pair; split pair is (x, y);
decrypt x is $\{z\}_k$; split z is (s, p);
decrypt y is $\{z\}_k$; split z is (p', s'); out net (p, p');

$P \triangleq$ new k ; (! P_A | ! P_B)

$O \triangleq$ inp net x ; split x is (y, z); out net (z, y);

Fixing the Example

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

This protocol is not robustly safe!

- This protocol can easily be fixed:

$A \rightarrow B : (\{s, p\}_k, \{s', p'\}_k)$

$B \rightarrow A : (p, p')$

Fixing the Example

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

This protocol is not robustly safe!

- This protocol can easily be fixed:

$A \rightarrow B : (\{s, p\}_k, \{s', p'\}_k)$

$B \rightarrow A : (p, p')$

- Is the protocol now robustly safe?
- To show robust safety we have to consider all possible attackers that our model permits.
- Showing robust safety naively is tedious.

Showing Robust Safety by Type Checking

- Type systems for secrecy formalize a few basic rules that are easy to check and that guarantee robust safety.

Basic Types

Basic types.

$$T, U, V ::= \text{Public} \mid \text{Secret} \mid \text{Un} \mid \dots$$

- **Public.** Data that may be published.
- **Secret.** Data that must be kept secret.
- **Un.** Data received from untrusted channels.

Subtyping to Control Data Flow

Subtyping controls data flow.

- If $T <: U$, then data of type T may flow into channels of type U .
- If $T \not<: U$, then data of type T must not flow into channels of type U .

Subtyping to Control Data Flow

Subtyping controls data flow.

- If $T <: U$, then data of type T may flow into channels of type U .
- If $T \not<: U$, then data of type T must not flow into channels of type U .

Basic subtyping rules.

- Untrusted channels are of type Un .
- $Public <: Un$
- $Secret \not<: Un$

Subtyping to Control Data Flow

Subtyping controls data flow.

- If $T <: U$, then data of type T may flow into channels of type U .
- If $T \not<: U$, then data of type T must not flow into channels of type U .

Basic subtyping rules.

- Untrusted channels are of type Un .
- $\text{Public} <: \text{Un}$
- $\text{Secret} \not<: \text{Un}$

Example.

- *Assume* $\text{net} : \text{Un}$.
- $\text{new } p : \text{Public}; \text{out } \text{net } p;$ *ok because* $\text{Public} <: \text{Un}$
- $\text{new } s : \text{Secret}; \text{out } \text{net } s;$ *not ok because* $\text{Secret} \not<: \text{Un}$

Type for Symmetric Keys

Type for symmetric keys.

$$T ::= \dots \mid \text{SymKey}(T)$$

Symmetric keys for encrypting data of type T .

Type for Symmetric Keys

Type for symmetric keys.

$$T ::= \dots \mid \text{SymKey}(T)$$

Symmetric keys for encrypting data of type T .

Subtyping rule for symmetric keys.

$$\text{SymKey}(T) <: \text{Secret}$$

Note: $\text{SymKey}(T) \not<: \text{Un.}$

Typing Rules for Symmetric Encryption

Symmetric encryption.

$$\begin{array}{c} \text{(Sym Enc)} \\ \frac{E \vdash M : T \quad E \vdash K : \text{SymKey}(T)}{E \vdash \{M\}_K : \text{Un}} \end{array}$$

Typing Rules for Symmetric Encryption

Symmetric encryption.

$$\frac{\text{(Sym Enc)} \quad E \vdash M : T \quad E \vdash K : \text{SymKey}(T)}{E \vdash \{M\}_K : \text{Un}}$$

Symmetric decryption.

$$\frac{\text{(Sym Dec)} \quad E \vdash M : \text{Un} \quad E \vdash K : \text{SymKey}(T) \quad E, x : T \vdash P : \text{ok}}{E \vdash \text{decrypt } M \text{ is } \{x\}_K; P : \text{ok}}$$

Our Example

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{s', p'\}_k)$

$B \rightarrow A : (p, p')$

Our Example

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{s', p'\}_k)$

$B \rightarrow A : (p, p')$

As a type-annotated formal narration:

$P_A \triangleq$ new $s : \text{Secret}$; new $p : \text{Public}$;
new $p' : \text{Public}$; new $s' : \text{Secret}$;
(secret(s) | secret(s') | out net ($\{(s, p)\}_k, \{(s', p')\}_k$); |
inp net x ; if $x = (p, p')$ then stop)

$P_B \triangleq$ inp net pair; split pair is (x, y);
decrypt x is $\{z\}_k$; split z is (s, p);
decrypt y is $\{z\}_k$; split z is (s', p'); out net (p, p');

$P \triangleq$ new $k : \text{SymKey}(\text{Secret}, \text{Public})$; (! P_A | ! P_B)

Our Example

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{s', p'\}_k)$

$B \rightarrow A : (p, p')$

As a type-annotated formal narration:

$P_A \triangleq$ new $s : \text{Secret}$; new $p : \text{Public}$;
new $p' : \text{Public}$; new $s' : \text{Secret}$;
(secret(s) | secret(s') | out net ($\{(s, p)\}_k, \{(s', p')\}_k$); |
inp net x ; if $x = (p, p')$ then stop)

$P_B \triangleq$ inp net pair; split pair is (x, y);
decrypt x is $\{z\}_k$; split z is (s, p);
decrypt y is $\{z\}_k$; split z is (s', p'); out net (p, p');

$P \triangleq$ new $k : \text{SymKey}(\text{Secret}, \text{Public})$; (! P_A | ! P_B)

- This protocol type-checks, i.e., $\text{net} : \text{Un} \vdash P : \text{ok}$

Where Does Typing of the Flawed Protocol Fail?

A generates secrets s, s' and publics p, p'

$A \rightarrow B : (\{s, p\}_k, \{p', s'\}_k)$

$B \rightarrow A : (p, p')$

An attempt to assign a type to the key k .

- For the first ciphertext, k needs type **SymKey(Secret, Public)**.
- For the second ciphertext, k needs type **SymKey(Public, Secret)**.
- But these two types are incompatible. This cannot be resolved.

Type Soundness

Theorem (Type Soundness).

If $\vec{n} : \text{Un} \vdash P : \text{ok}$, then P is robustly safe for secrecy.

Type Soundness

Theorem (Type Soundness).

If $\vec{n} : \text{Un} \vdash P : \text{ok}$, then P is robustly safe for secrecy.

Type-based verification.

- 1 Annotate P with types.
- 2 Check if P is well-typed in an environment that assigns Un to its free names.

Preventing Confusion by Tagging

Preventing Confusion by Tagging

A quote from Abadi/Needham [AN96]:

“It seems important that principals recognize messages for what they are, and can associate them correctly with the current step of whatever protocol they are executing. There are two possible forms of confusion: first, between the current message and a similar message from a previous protocol run, and second, between the current message and a message belonging to either elsewhere in the protocol, or to another protocol.”

- The first form of confusion can be prevented by **nonces or timestamps**. More on that later.
- The second form of confusion can be prevented by **tagging**.

Tagging

Tagging is a prudent engineering practice for security protocols.
It prevents confusion attacks.

Tagging

Tagging is a prudent engineering practice for security protocols. It prevents confusion attacks.

- Senders attach a tag to a message, which identifies the protocol that the message belongs to and its message number within a protocol run.
- Receivers check if received messages carry the expected tags.

Tagging

Tagging is a prudent engineering practice for security protocols. It prevents confusion attacks.

- Senders attach a tag to a message, which identifies the protocol that the message belongs to and its message number within a protocol run.
- Receivers check if received messages carry the expected tags.

For instance:

$$\begin{aligned} A \rightarrow B & : (\{msg1, s, p\}_k, \{msg2, p', s'\}_k) \\ B \rightarrow A & : (p, p') \end{aligned}$$

- *msg1* and *msg2* are tags that *B* checks on receipt.
- This protocol is robustly safe.

Tagging

Tagging.

$$M ::= \dots \mid L(M)$$

“message M tagged by L ”

Tagging

Tagging.

$M ::= \dots \mid L(M)$ “message M tagged by L ”

Untagging.

$P ::= \dots \mid \text{untag } M \text{ is } L(x); P$

Tagging

Tagging.

$M ::= \dots \mid L(M)$ “message M tagged by L ”

Untagging.

$P ::= \dots \mid \text{untag } M \text{ is } L(x); P$

Reduction rule.

$\text{untag } L(M) \text{ is } L(x); P \rightarrow P\{M/x\}$

Types for Tagging

Tag types.

$$T ::= \dots \mid \text{Tagged} \mid T \rightarrow \text{Tagged}$$

Tagged: Tagged messages.

$T \rightarrow \text{Tagged}$: Tags for tagging messages of type T .

Types for Tagging

Tag types.

$$T ::= \dots \mid \text{Tagged} \mid T \rightarrow \text{Tagged}$$

Tagged: Tagged messages.

$T \rightarrow \text{Tagged}$: Tags for tagging messages of type T .

Subtyping.

$$T \rightarrow \text{Tagged} <: \text{Un}$$

Typing Rules for Tagging

Tagging.

$$\begin{array}{c} \text{(Tag)} \\ \frac{E \vdash L : T \rightarrow \text{Tagged} \quad E \vdash M : T}{E \vdash L(M) : \text{Tagged}} \end{array}$$

Typing Rules for Tagging

Tagging.

$$\frac{\text{(Tag)} \quad E \vdash L : T \rightarrow \text{Tagged} \quad E \vdash M : T}{E \vdash L(M) : \text{Tagged}}$$

Untagging.

$$\frac{\text{(Untag)} \quad E \vdash M : \text{Tagged} \quad E \vdash L : T \rightarrow \text{Tagged} \quad E, x : T \vdash P : \text{ok}}{E \vdash \text{untag } M \text{ is } L(x); P : \text{ok}}$$

Example

$A \rightarrow B : (\{msg1(s, p)\}_k, \{msg2(p', s')\}_k)$

$B \rightarrow A : (p, p')$

Example

$$\begin{aligned} A \rightarrow B & : (\{msg1(s, p)\}_k, \{msg2(p', s')\}_k) \\ B \rightarrow A & : (p, p') \end{aligned}$$

As a type-annotated formal narration:

$$\begin{aligned} P_A & \triangleq \dots \text{ out } net (\{msg1(s, p)\}_k, \{msg2(p', s')\}_k); \dots \\ P_B & \triangleq \text{ inp } net \text{ pair}; \text{ split } pair \text{ is } (x, y); \\ & \text{ decrypt } x \text{ is } \{z\}_k; \\ & \text{ untag } z \text{ is } msg1(z'); \text{ split } z' \text{ is } (s, p); \\ & \text{ decrypt } y \text{ is } \{z\}_k; \\ & \text{ untag } z \text{ is } msg2(z'); \text{ split } z' \text{ is } (p', s'); \text{ out } net (p, p'); \\ P & \triangleq \text{ new } k : \text{SymKey}(\text{Tagged}); \\ & \text{ new } msg1 : (\text{Secret}, \text{Public}) \rightarrow \text{Tagged}; \\ & \text{ new } msg2 : (\text{Public}, \text{Secret}) \rightarrow \text{Tagged}; \\ & (!\text{out } net (msg1, msg2); \mid !P_A \mid !P_B) \end{aligned}$$

Specifying Authenticity

Specifying Authenticity

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

Specifying Authenticity

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- Insert begin- and end-markers into the protocol.
- Require that they match up in protocol runs.

Specifying Authenticity

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- Insert begin- and end-markers into the protocol.
- Require that they match up in protocol runs.
- As spi processes:
 - $P_A \triangleq$ new m : Secret;
begin!(Send(A, m, B));
out net (A, {(B, m)}_{kas});
 - $P_B \triangleq$ inp net {(a : Public, m : Secret)}_{kbs};
end(Send(a, m, B))

Specifying Authenticity

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- Insert begin- and end-markers into the protocol.
- Require that they match up in protocol runs.
- As spi processes:
 - $P_A \triangleq$ new m : Secret;
begin!(Send(A, m, B));
out net (A, {(B, m)}_{kas});
 - $P_B \triangleq$ inp net {(a : Public, m : Secret)}_{kbs};
end(Send(a, m, B))
- The begin- and end-markers are called **correspondence assertions**.

Robust Safety: Informal Definition

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- P is safe for authenticity if in every run every $\text{end}(M)$ is preceded by a matching $\text{begin!}(M)$.

Robust Safety: Informal Definition

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- P is **safe for authenticity** if in every run every $\text{end}(M)$ is preceded by a matching $\text{begin!}(M)$.
- An **opponent** is a process that does not contain **end-assertions**.

Robust Safety: Informal Definition

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- P is **safe for authenticity** if in every run every $\text{end}(M)$ is preceded by a matching $\text{begin!}(M)$.
- An **opponent** is a process that does not contain **end-assertions**.
- P is **robustly safe for authenticity** if $P \mid O$ is safe for authenticity for all opponents O .

An Authenticity Violation

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- If each agent acts both in the initiator and receiver role, then there is a type confusion attack:

An Authenticity Violation

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- If each agent acts both in the initiator and receiver role, then there is a type confusion attack:

- *A begins! Send(A, m, B)*

$A \text{ as initiator} \rightarrow O : A, \{B, m\}_{kas}$

$O \rightarrow A \text{ as receiver} : \{B, m\}_{kas}$

A ends Send(B, m, A)

An Authenticity Violation

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- If each agent acts both in the initiator and receiver role, then there is a type confusion attack:
 - *A begins! Send(A, m, B)*
A as initiator $\rightarrow O : A, \{B, m\}_{kas}$
 $O \rightarrow A$ as receiver : $\{B, m\}_{kas}$
A ends Send(B, m, A)
- *begin!(Send(A, m, B))* and *end(Send(B, m, A))* do not match.
- A ends a *Send(B, m, A)*-session that B has never begun.

An Authenticity Violation

A begins! Send(A, m, B)

$A \rightarrow S : A, \{B, m\}_{kas}$

$S \rightarrow B : \{A, m\}_{kbs}$

B ends Send(A, m, B)

- If each agent acts both in the initiator and receiver role, then there is a type confusion attack:
 - *A begins! Send(A, m, B)*
A as initiator $\rightarrow O : A, \{B, m\}_{kas}$
 $O \rightarrow A$ as receiver : $\{B, m\}_{kas}$
A ends Send(B, m, A)
- *begin!(Send(A, m, B))* and *end(Send(B, m, A))* do not match.
- A ends a *Send(B, m, A)*-session that B has never begun.
- The protocol can be fixed by tagging the two encrypted messages: $\{msg1(B, m)\}_{kas}$ and $\{msg2(A, m)\}_{kbs}$.

Types for Authenticity

Types for Authenticity

Abadi/Needham's Principle 1 [AN96].

“Every message should say what it means: the interpretation of the message should depend only on its content. It should be possible to write down a straightforward English sentence describing the content—though if there is a suitable formalism available that is good, too.”

- Types for authenticity enforce this principle using dependent types and effects.

Tracking Correspondences

Enriched type environments.

$$E ::= \epsilon \mid E, x : T \mid E, !\text{begun}(M)$$

Tracking Correspondences

Enriched type environments.

$$E ::= \epsilon \mid E, x : T \mid E, !\text{begin}(M)$$

(Begin)

$$\frac{E, !\text{begin}(M) \vdash P : \text{ok}}{E \vdash \text{begin}!(M); P : \text{ok}}$$

Tracking Correspondences

Enriched type environments.

$$E ::= \epsilon \mid E, x : T \mid E, !\text{begun}(M)$$

(Begin)

$$\frac{E, !\text{begun}(M) \vdash P : \text{ok}}{E \vdash \text{begin}!(M); P : \text{ok}}$$

(End)

$$\frac{E \vdash \text{begun}(M) \quad E \vdash P : \text{ok}}{E \vdash \text{end}(M); P : \text{ok}}$$

Types with Latent Effects

Effect types.

$$T ::= \dots \mid T[!begin(M)]$$

Types with Latent Effects

Effect types.

$$T ::= \dots \mid T[! \text{begun}(M)]$$

(Attach Begun)

$$\frac{E \vdash M : T \quad E \vdash ! \text{begun}(M)}{E \vdash M : T[! \text{begun}(M)]}$$

Types with Latent Effects

Effect types.

$$T ::= \dots \mid T[!begun(M)]$$

(Attach Begun)

$$\frac{E \vdash M : T \quad E \vdash !begun(M)}{E \vdash M : T[!begun(M)]}$$

(Use Begun)

$$\frac{E \vdash M : T[!begun(M)] \quad E, !begun(M) \vdash P : \text{ok}}{E \vdash P : \text{ok}}$$

Dependent Pair Types

Dependent pair types.

$$T ::= \dots \mid (x : T, U)$$

Dependent Pair Types

Dependent pair types.

$$T ::= \dots \mid (x : T, U)$$

(Pair)

$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (x : T, U)}$$

Dependent Pair Types

Dependent pair types.

$$T ::= \dots \mid (x : T, U)$$

(Pair)

$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (x : T, U)}$$

(Split)

$$\frac{E \vdash M : (x : T, U) \quad E, x : T, y : U \vdash P : \text{ok}}{E \vdash \text{split } M \text{ is } (x, y); P : \text{ok}}$$

Matching

Matching the first component of a pair.

$$P ::= \dots \mid \text{match } M \text{ is } (N, x); P$$

Matching

Matching the first component of a pair.

$$P ::= \dots \mid \text{match } M \text{ is } (N, x); P$$

Reduction rule.

$$\text{match } (N, M) \text{ is } (N, x); P \rightarrow P\{M/x\}$$

Matching

Matching the first component of a pair.

$$P ::= \dots \mid \text{match } M \text{ is } (N, x); P$$

Reduction rule.

$$\text{match } (N, M) \text{ is } (N, x); P \rightarrow P\{M/x\}$$

(Match)

$$\frac{E \vdash M : (x : T, U) \quad E \vdash N : T \quad E, y : U\{N/y\} \vdash P : \text{ok}}{E \vdash \text{match } M \text{ is } (N, y); P : \text{ok}}$$

Parametric Tag Types

Parametric tag types.

$$T ::= \dots \mid \text{Tagged}(\vec{M}) \mid \forall \vec{x}. T \rightarrow \text{Tagged}(\vec{x})$$

Parametric Tag Types

Parametric tag types.

$$T ::= \dots \mid \text{Tagged}(\vec{M}) \mid \forall \vec{x}. T \rightarrow \text{Tagged}(\vec{x})$$

(Tag)

$$\frac{E \vdash L : \forall \vec{x}. T \rightarrow \text{Tagged}(\vec{x}) \quad E \vdash M : T\{\vec{N}/\vec{x}\}}{E \vdash L(M) : \text{Tagged}(\vec{N})}$$

Parametric Tag Types

Parametric tag types.

$$T ::= \dots \mid \text{Tagged}(\vec{M}) \mid \forall \vec{x}. T \rightarrow \text{Tagged}(\vec{x})$$

(Tag)

$$\frac{E \vdash L : \forall \vec{x}. T \rightarrow \text{Tagged}(\vec{x}) \quad E \vdash M : T\{\vec{N}/\vec{x}\}}{E \vdash L(M) : \text{Tagged}(\vec{N})}$$

(Untag)

$$\frac{E \vdash M : \text{Tagged}(\vec{N}) \quad E \vdash L : \forall \vec{x}. T \rightarrow \text{Tagged}(\vec{x}) \quad E, x : T\{\vec{N}/\vec{x}\} \vdash P : \text{ok}}{E \vdash \text{untag } M \text{ is } L(x); P : \text{ok}}$$

Types for the Fixed Protocol

A begins! Send(A, m, B)

$A \rightarrow S : A, \{msg1(B, m)\}_{kas}$

$S \rightarrow B : \{msg2(A, m)\}_{kbs}$

B ends Send(A, m, B)

Types for the Fixed Protocol

A begins! $Send(A, m, B)$

$A \rightarrow S : A, \{msg1(B, m)\}_{kas}$

$S \rightarrow B : \{msg2(A, m)\}_{kbs}$

B ends $Send(A, m, B)$

- $kas : SymKey(Tagged(A))$
- $kbs : SymKey(Tagged(B))$

Types for the Fixed Protocol

A begins! $Send(A, m, B)$
 $A \rightarrow S : A, \{msg1(B, m)\}_{kas}$
 $S \rightarrow B : \{msg2(A, m)\}_{kbs}$
 B ends $Send(A, m, B)$

- $kas : \text{SymKey}(\text{Tagged}(A))$
- $kbs : \text{SymKey}(\text{Tagged}(B))$
- $msg1 : \forall a. (b:\text{Un}, m:\text{Secret})[!\text{begun}(Send(a, m, b))]$
 $\rightarrow \text{Tagged}(a)$
- $msg2 : \forall a. (b:\text{Un}, m:\text{Secret})[!\text{begun}(Send(b, m, a))]$
 $\rightarrow \text{Tagged}(a)$
- Note that the tag types differ.

Another Flawed Protocol

A begins! $Send(A, m, B)$

$A \rightarrow S : A, B, \{msg1(m)\}_{kas}$

$S \rightarrow B : A, \{msg2(m)\}_{kbs}$

B ends $Send(A, m, B)$

Another Flawed Protocol

A begins! $Send(A, m, B)$

$A \rightarrow S : A, B, \{msg1(m)\}_{kas}$

$S \rightarrow B : A, \{msg2(m)\}_{kbs}$

B ends $Send(A, m, B)$

- An attack: A begins! “ $Send(A, m, B)$ ”

$A \rightarrow S : A, B, \{msg1(m)\}_{kas}$

$S \rightarrow O : A, \{msg2(m)\}_{kbs}$

$O \rightarrow B : C, \{msg2(m)\}_{kbs}$

B ends “ $Send(C, m, B)$ ”

Another Flawed Protocol

A begins! $Send(A, m, B)$

$A \rightarrow S : A, B, \{msg1(m)\}_{kas}$

$S \rightarrow B : A, \{msg2(m)\}_{kbs}$

B ends $Send(A, m, B)$

- An attack: A begins! “ $Send(A, m, B)$ ”

$A \rightarrow S : A, B, \{msg1(m)\}_{kas}$

$S \rightarrow O : A, \{msg2(m)\}_{kbs}$

$O \rightarrow B : C, \{msg2(m)\}_{kbs}$

B ends “ $Send(C, m, B)$ ”

- Attempting to assign a type to $msg2$:

- $msg2 : \forall b. (m:Secret)[!begun(a, m, b)] \rightarrow Tagged(b)$
- **Type error: a is out of scope!**

Another Flawed Protocol

A begins! $Send(A, m, B)$
 $A \rightarrow S : A, B, \{msg1(m)\}_{kas}$
 $S \rightarrow B : A, \{msg2(m)\}_{kbs}$
B ends $Send(A, m, B)$

- An attack: A begins! “ $Send(A, m, B)$ ”
 $A \rightarrow S : A, B, \{msg1(m)\}_{kas}$
 $S \rightarrow O : A, \{msg2(m)\}_{kbs}$
 $O \rightarrow B : C, \{msg2(m)\}_{kbs}$
B ends “ $Send(C, m, B)$ ”
- Attempting to assign a type to $msg2$:
 - $msg2 : \forall b. (m:Secret)[!begun(a, m, b)] \rightarrow Tagged(b)$
 - **Type error: a is out of scope!**
- Dependent types enforce Principle 1.

What Else? Pattern-Matching Spi

- Strictly syntax-directed type systems are sometimes too unflexible. We sometimes need to write types like this:

$$(\{\#\}(\exists m : T, \exists x))\}_{\exists y-1}, \exists a : U, \exists b : U][!begin(m, a, b)]$$

What Else? Pattern-Matching Spi

- Strictly syntax-directed type systems are sometimes too inflexible. We sometimes need to write types like this:

$$(\{\#\!(\exists m : T, \exists x)\}\}_{\exists y^{-1}}, \exists a : U, \exists b : U)[!begin(m, a, b)]$$

- Gordon/Jeffrey's type systems do not allow nested encryption, e.g., sign-then-encrypt.

What Else? Pattern-Matching Spi

- Strictly syntax-directed type systems are sometimes too inflexible. We sometimes need to write types like this:

$$(\{\#\}(\exists m : T, \exists x))\}_{\exists y^{-1}}, \exists a : U, \exists b : U][!begin(m, a, b)]$$

- Gordon/Jeffrey's type systems do not allow nested encryption, e.g., sign-then-encrypt.
- We have a that system resolves these limitations by using a “pattern-matching” type system.

What Else? Pattern-Matching Spi

- Strictly syntax-directed type systems are sometimes too inflexible. We sometimes need to write types like this:
$$(\{\#\!(\exists m : T, \exists x)\}\}_{\exists y^{-1}}, \exists a : U, \exists b : U)[!begin(m, a, b)]$$
- Gordon/Jeffrey's type systems do not allow nested encryption, e.g., sign-then-encrypt.
- We have a that system resolves these limitations by using a “pattern-matching” type system.
- The current implementation of the Cryptyc type-checker is based on this system.
 - <http://www.cryptyc.org>

Reference: Haack/Jeffrey [HJ06].

What Else? Injective Agreement

$\text{begin!}(M)$		one begin may end many M -sessions
$\text{begin}(M)$		one begin may end one M -session

What Else? Injective Agreement

$\text{begin!}(M)$		one begin may end many M -sessions
$\text{begin}(M)$		one begin may end one M -session

- Injective agreement disallows replays.

What Else? Injective Agreement

$\text{begin!}(M)$ | one begin may end many M -sessions
 $\text{begin}(M)$ | one begin may end one M -session

- Injective agreement disallows replays.
- Nonces are the tool to achieve injective agreement.

What Else? Injective Agreement

$\text{begin!}(M)$ | one begin may end many M -sessions
 $\text{begin}(M)$ | one begin may end one M -session

- Injective agreement disallows replays.
- Nonces are the tool to achieve injective agreement.
- Nonce types: $\text{Chall}(\bar{A})$, $\text{Resp}(\bar{B})$

What Else? Injective Agreement

$\text{begin!}(M)$ | one begin may end many M -sessions
 $\text{begin}(M)$ | one begin may end one M -session

- Injective agreement disallows replays.
- Nonces are the tool to achieve injective agreement.
- Nonce types: $\text{Chall}(\bar{A})$, $\text{Resp}(\bar{B})$

(Nonce Cast)

$$\frac{E \vdash n : \text{Chall}(\bar{A}), \bar{A}, \bar{B}}{E \vdash n : \text{Resp}(\bar{B})}$$

What Else? Injective Agreement

$\text{begin!}(M)$ | one begin may end many M -sessions
 $\text{begin}(M)$ | one begin may end one M -session

- Injective agreement disallows replays.
- Nonces are the tool to achieve injective agreement.
- Nonce types: $\text{Chall}(\bar{A})$, $\text{Resp}(\bar{B})$

(Nonce Cast)

$$\frac{E \vdash n : \text{Chall}(\bar{A}), \bar{A}, \bar{B}}{E \vdash n : \text{Resp}(\bar{B})}$$

(Nonce Use)

$$\frac{E_1 \vdash n : \text{Chall}(\bar{A}), n : \text{Resp}(\bar{B}), \text{fresh}(n) \quad E_2, \bar{A}, \bar{B} \vdash P : \text{ok}}{E_1, E_2 \vdash P : \text{ok}}$$

References: Gordon/Jeffrey [GJ03a, GJ03b].

What Else? Timeliness

- A type system for a discretely timed spi-calculus.

What Else? Timeliness

- A type system for a discretely timed spi-calculus.
- A primitive for key-cracking.

What Else? Timeliness

- A type system for a discretely timed spi-calculus.
- A primitive for key-cracking.
- One time unit represents an epoch, which is the time needed for cracking a key.

What Else? Timeliness

- A type system for a discretely timed spi-calculus.
- A primitive for key-cracking.
- One time unit represents an epoch, which is the time needed for cracking a key.
- Can verify secrecy and authenticity in the presence of key-compromise (in an abstract model).

What Else? Timeliness

- A type system for a discretely timed spi-calculus.
- A primitive for key-cracking.
- One time unit represents an epoch, which is the time needed for cracking a key.
- Can verify secrecy and authenticity in the presence of key-compromise (in an abstract model).
- Can verify protocols that use timestamps.

Reference: Haack/Jeffrey [HJ05].

Future Work

- Bringing the following closer together:
 - Official informal protocol specifications (RFCs).
 - Abstract protocol narrations verified by current tools.
 - Protocol implementations.

Future Work

- Bringing the following closer together:
 - Official informal protocol specifications (RFCs).
 - Abstract protocol narrations verified by current tools.
 - Protocol implementations.
- Types for primitives with associated algebraic equalities (especially Diffie-Hellman).

Future Work

- Bringing the following closer together:
 - Official informal protocol specifications (RFCs).
 - Abstract protocol narrations verified by current tools.
 - Protocol implementations.
- Types for primitives with associated algebraic equalities (especially Diffie-Hellman).
- Type inference.

Future Work

- Bringing the following closer together:
 - Official informal protocol specifications (RFCs).
 - Abstract protocol narrations verified by current tools.
 - Protocol implementations.
- Types for primitives with associated algebraic equalities (especially Diffie-Hellman).
- Type inference.
- Is Gordon/Jeffrey's authenticity type system robustly safe in the presence of compromised hosts?

References



M. Abadi and B. Blanchet.

Secrecy types for asymmetric communication.

In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2001.



M. Abadi and B. Blanchet.

Analyzing security protocols with secrecy types and logic programs.

In *29th ACM Symposium on Principles of Programming Languages*, pages 33–44, 2002.



M. Abadi.

Secrecy by typing in security protocols.

Journal of the ACM, 46(5):749–786, September 1999.



M. Abadi and R. Needham.

Prudent engineering practice for cryptographic protocols.

IEEE Transactions on Software Engineering, 22(1):6–15, 1996.



A.D. Gordon and A. Jeffrey.

Authenticity by typing for security protocols.

In *14th IEEE Computer Security Foundations Workshop*, pages 145–159. IEEE Computer Society Press, 2001.



A.D. Gordon and A. Jeffrey.

Types and effects for asymmetric cryptographic protocols.

In *15th IEEE Computer Security Foundations Workshop*, pages 77–91. IEEE Computer Society Press, 2002.



A.D. Gordon and A. Jeffrey.

Typing one-to-one and one-to-many correspondences in security protocols.

In *Proc. Int. Software Security Symp.*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, 2002.

References (cont.)



A.D. Gordon and A. Jeffrey.

Authenticity by typing for security protocols.

J. Computer Security, 11(4):451–521, 2003.



A.D. Gordon and A. Jeffrey.

Types and effects for asymmetric cryptographic protocols.

J. Computer Security, 12(3/4):435–484, 2003.



C. Haack and A. Jeffrey.

Timed spi-calculus with types for secrecy and authenticity.

In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2005.



C. Haack and A. Jeffrey.

Pattern-matching spi-calculus.

Information and Computation, 204:1195–1263, 2006.