A Multi-Role Translation of Protocol Narration into the Spi-Calculus with Correspondence Assertions[⋆]

Eijiro Sumii and Yuji Sato

Graduate School of Information Sciences, Tohoku University
sumii@ecei.tohoku.ac.jp

Abstract. We present an interpretation of protocol narrations by means of translation into the spi-calculus. Our translation allows participants to play multiple roles in parallel, leading to a more general interpretation that considers a wider range of attacks than previous work. We test the validity of our translation by introducing correspondence assertions [Woo and Lam, S&P 1993] to both the protocol narrations and the spi-calculus, and verifying a number of examples by using SpiCA2 [Dahl, Kobayashi, Sun, and Hüttel, ATVA 2011], a sound and automatic type-based verifier of correspondence assertions.

1 Introduction

Security protocols are often written in the so-called *narration* notation (e.g. [4, 9]). For instance, a "repaired" version of the Wide Mouthed Frog protocol [2, Section 3.2.4] can be written like:

$$\begin{array}{l}
1.\ A \rightarrow S : A \\
2.\ S \rightarrow A : N_S \\
3.\ A \rightarrow S : A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}} \\
4.\ S \rightarrow B : () \\
5.\ B \rightarrow S : N_B \\
6.\ S \rightarrow B : \{S, A, B, K_{AB}, N_B\}_{K_{BS}} \\
7.\ A \rightarrow B : A, \{M\}_{K_{AB}}
\end{array}$$

(In this paper, we write () for the "dummy" message. It was written $*$ in [2].) If literally read, the narration above says

1. Principal A sends message A to principal S.
2. Principal S sends message N_S to principal A.
3. Principal A sends message $A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}}$ to principal S.
4. ...

and so forth. However, this reading is rather superficial and describes only a small part of the actual behavior of each principal. For example:

⋆ Draft as of June 29, 2013

– In step 1, the "server" S should take the name A as a *parameter* to the rest of its actions.
– In step 2, S should *freshly generate* the "nonce" N_S.
– In step 3, S should *decrypt* the ciphertext $\{A, A, B, K_{AB}, N_S\}_{K_{AS}}$ and "check" the last element N_S (as well as the three elements A, A, and B).

To bridge this gap, translations into variants of (a subset of) the spi-calculus [2], based on the *knowledge* of each principal at each point of the protocol, have been proposed [3, 11]. The basic ideas of the translations are as follows:

– When a principal X receives a message M that X does not yet know (i.e., M is not in the knowledge of X at the point of the protocol), X learns M (i.e., M is added to the knowledge of X for the rest of the protocol).
– When a principal X receives a message M that X already knows, X checks whether M is equal to what X knows (if not, X stops).
– When a principal X sends a message M that X does *not* know, X freshly generates M and adds M to its knowledge.[1]
– When a principal X receives a ciphertext $\{M\}_{K^+}$ of which X knows the decryption key K^- (same as K^+ in the case of symmetric encryption), X decrypts the ciphertext and behaves as if it received the plaintext M.

For example, if the initial knowledge of A, B, and S is $\{A, B, S, K_{AS}, M\}$, $\{A, B, S, K_{AS}, K_{BS}\}$, and $\{A, B, S, K_{BS}\}$, respectively, and if K_{AS} and K_{BS} are secret (or "the initial knowledge of the attacker" is $\{A, B, S, M\}$), the narration above can be translated into the spi-calculus process

$$\nu K_{AS}.\ \nu K_{BS}.\ (A \mid B \mid S)$$

where

$$\begin{array}{rll}
A = & \texttt{net!}A \mid & \text{(i)}\\
& \texttt{net?}N_S. & \text{(ii)}\\
& \nu K_{AB}.\ \texttt{net!}(A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}}) \mid & \text{(iii)}\\
& \texttt{net!}(A, \{M\}_{K_{AB}}) \mid & \text{(iv)}\\
& 0 &
\end{array}$$

$$\begin{array}{rll}
B = & \texttt{net?}(). & \text{(i)}\\
& \nu N_B.\ \texttt{net!}N_B \mid & \text{(ii)}\\
& \texttt{net?}c_1. & \text{(iii)}\\
& \texttt{decrypt } c_1 \texttt{ is } \{S_1, A_1, B_1, K_{AB}, N_{B1}\}_{K_{BS}} \texttt{ in} & \text{(iv)}\\
& \texttt{check } (S_1, A_1, B_1, N_{B1}) \texttt{ is } (S, A, B, N_B) \texttt{ in} & \text{(v)}\\
& \texttt{net?}(A_2, c_2). & \text{(vi)}\\
& \texttt{check } A_2 \texttt{ is } A \texttt{ in} & \text{(vii)}\\
& \texttt{decrypt } c_2 \texttt{ is } \{M\}_{K_{AB}} \texttt{ in} & \text{(viii)}\\
& 0 &
\end{array}$$

[1] In [3], freshly generated messages are explicitly declared "for the sake of clarity"(p. 487).

$$
\begin{array}{lll}
S = & \texttt{net?}A_3. & \text{(i)} \\
& \texttt{check } A_3 \texttt{ is } A \texttt{ in} & \text{(ii)} \\
& \nu N_S.\ \texttt{net!}N_S \mid & \text{(iii)} \\
& \texttt{net?}(A_4, c_3). & \text{(iv)} \\
& \texttt{check } A_4 \texttt{ is } A \texttt{ in} & \text{(v)} \\
& \texttt{decrypt } c_3 \texttt{ is } \{A_5, A_6, B_1, K_{AB}, N_{S1}\}_{K_{AS}} \texttt{ in} & \text{(vi)} \\
& \texttt{check } (A_5, A_6, B_1, N_{S1}) \texttt{ is } (A, A, B, N_S) \texttt{ in} & \text{(vii)} \\
& \texttt{net!}() \mid & \text{(viii)} \\
& \texttt{net?}N_B. & \text{(ix)} \\
& \texttt{net!}\{S, A, B, K_{AB}, N_B\}_{K_{BS}} \mid & \text{(x)} \\
& 0 &
\end{array}
$$

(in this paper, we use `!` and `?` for output and input, respectively; for the sake of brevity, we also use pattern matching notations on tuples). The key points of the translation are as follows:

– In line (ii) of A and line (ix) of S, the received nonces N_S and N_B are respectively added to the knowledge of the receivers.
– In line (iv) and (viii) of B and line (vi) of S, the received ciphertexts c_1, c_2, and c_3 are decrypted with the known keys K_{BS}, K_{AB}, and K_{AS}, respectively.
– In line (v) and (vii) of B and line (ii), (v), and (vii) of S, the integrity of all the known messages are checked when received (or decrypted).
– In line (iii) of A, line (ii) of B, and line (iii) of S, the key K_{AB} and the nonces N_B and N_S, respectively, are freshly generated.

However, this interpretation still suffers from the following limitations:

– The principals B and S assumes a particular A and refuses to talk with other principals. This is especially problematic for the "server" S, which usually should process requests from multiple clients.
– Each principal plays only a single, fixed role (for once). Even if we replicate the translated processes A, B, and S, they still cannot play any other role.

To see a consequence of these limitations, consider the following (broken) variant of the protocol:

$$
\begin{array}{l}
1.\ A \to S : A \\
2.\ S \to A : N_S \\
3.\ A \to S : A, \{B, K_{AB}, N_S\}_{K_{AS}} \\
4.\ S \to B : () \\
5.\ B \to S : N_B \\
6.\ S \to B : \{A, K_{AB}, N_B\}_{K_{BS}} \\
7.\ A \to B : A, \{M\}_{K_{AB}}
\end{array}
$$

Note that the ciphertexts in step 3 and 6 are "simplified" from $\{A, A, B, K_{AB}, N_S\}_{K_{AS}}$ and $\{S, A, B, K_{AB}, N_B\}_{K_{BS}}$ to $\{B, K_{AB}, N_S\}_{K_{AS}}$ and $\{A, K_{AB}, N_B\}_{K_{BS}}$, respectively. As a result, the protocol becomes insecure when run *in parallel* with the following session of the *same protocol* in the *other direction* (i.e., the roles

Protocols π	$::=$	$\alpha_1; \ldots; \alpha_n$
Actions α	$::=$	$X \rightarrow Y : M \mid X \textit{ begins } M \mid X \textit{ ends } M$
Messages M	$::=$	$v \mid (M, N) \mid \texttt{inl}(M) \mid \texttt{inr}(M) \mid M^+ \mid M^- \mid \{M\}_N$
Variables v	$::=$	$X_{Y_1 \ldots Y_n}$
Atoms X	$::=$	$A, B, S, x, y, K, N, \ldots$

Fig. 1. Syntax of protocol narrations with correspondence assertions

of A and B are swapped).

$$\begin{aligned}
&1'.\ B \rightarrow S : B \\
&2'.\ S \rightarrow B : N'_S \\
&3'.\ B \rightarrow S : B, \{A, K_{BA}, N'_S\}_{K_{BS}} \\
&4'.\ S \rightarrow A : () \\
&5'.\ A \rightarrow S : N_A \\
&6'.\ S \rightarrow A : \{B, K_{BA}, N_A\}_{K_{AS}} \\
&7'.\ B \rightarrow A : B, \{M'\}_{K_{BA}}
\end{aligned}$$

Specifically, the attacker can substitute the nonce N'_S in step $2'$ with N_B in 5, and the ciphertext $\{A, K_{AB}, N_B\}_{K_{BS}}$ in step 6 with $\{A, K_{BA}, N_B\}_{K_{BS}}$ in $3'$, tricking B into using K_{BA} instead of K_{AB} in step 7. This flaw could be fixed by introducing "type tags" into the cihpertexts $\{B, K_{AB}, N_S\}_{K_{AS}}$ and $\{A, K_{AB}, N_B\}_{K_{BS}}$ of the protocol—like $\{\texttt{inl}(B, K_{AB}, N_S)\}_{K_{AS}}$ and $\{\texttt{inr}(A, K_{AB}, N_B)\}_{K_{BS}}$—but the problem here is that the previous translation cannot reflect this attack because of the one-to-one assignment of roles to principals.

In this paper, we propose an improved translation of protocol narrations into (a subset of) the spi-calculus, where *every* principal can play *every* role, getting rid of such limitations as above. We furthermore extend our translation to allow insider attacks (i.e., some of the principals may be malicious). We test the validity of our translations by type-checking the translated processes with SpiCA2 [5], a sound and automatic type-based verifier of correspondence assertions [7, 8, 12] in spi-calculus.

The rest of this paper is structured as follows. Section 2 gives the syntax of our protocol narrations and spi-calculus, both extended with correspondence assertions. Section 3 defines the translation, Section 4 gives an example, and Section 5 shows experimental results. Section 6 extends the translation with malicious participants and Section 7 concludes with discussions.

2 Syntax

The syntax of our protocol narrations—extended with correspondence assertions to be used by SpiCA2 after translation—is given in Figure 1. A protocol π is a sequence $\alpha_1; \ldots; \alpha_n$ of actions. An action α is either a transmission $X \rightarrow Y : M$ of message M from principal X to Y, or one of the correspondence assertions X *begins* M and X *ends* M. A message M is either a variable v, a pair (M, N)

$$\begin{array}{rl} \text{Processes } P ::= & 0 \;\mid\; \nu v.P \;\mid\; M\,\texttt{!}\,N \;\mid\; M\texttt{?}v.P \;\mid\; P \mid Q \;\mid\; *P \\ \mid & \texttt{check}\ M\ \texttt{is}\ N\ \texttt{in}\ P \;\mid\; \texttt{decrypt}\ M\ \texttt{is}\ \{v\}_N\ \texttt{in}\ P \\ \mid & \texttt{case}\ M\ \texttt{is}\ \texttt{inl}(v)\texttt{.}P \,[\!]\, \texttt{inr}(w)\texttt{.}Q \;\mid\; \texttt{split}\ M\ \texttt{is}\ (v,w)\ \texttt{in}\ P \\ \mid & \texttt{match}\ M\ \texttt{is}\ (N,v)\ \texttt{in}\ P \;\mid\; \texttt{begin}\ M.P \;\mid\; \texttt{end}\ M \end{array}$$

Fig. 2. Syntax of spi-calculus with correspondence assertions

of messages, a tagged message $\texttt{inl}(M)$ or $\texttt{inr}(M)$, one of the key pairs M^+ and M^-, or a ciphertext $\{M\}_N$. We assume that a ciphertext encrypted with v, v^+, and v^- can respectively be decrypted only with v, v^-, and v^+ (as in the standard Dolev-Yao model [6]). We often make v^+ public while keeping v^- private, and sometimes use encryption with v^- for signing (and decryption with v^+ for verification). A variable v has the form $X_{Y_1\ldots Y_n}$ for some $n \geq 0$, where $X, Y_1, \ldots, Y_n$ are a kind of "subvariables" called atoms. This will be useful for translating a parametrized variable (e.g., K_{AS} was parametrized over A in the protocols above) into a dynamic look-up.

The syntax of spi-calculus with correspondence assertions (input for SpiCA2 [5]) is given in Figure 2. Process 0 does nothing. $\nu v.P$ generates a fresh name, binds v to it, and executes P. $M\,\texttt{!}\,N$ sends message N to channel M, while $M\texttt{?}v.P$ receives a message from channel M, binds v to it, and executes P. $P \mid Q$ runs P and Q in parallel, and $*P$ spawns an infinite number of parallel P. `check` M `is` N `in` P compares M and N, and executes P if they are equal (or stops if not). `decrypt` M `is` $\{v\}_N$ `in` P decrypts the ciphertext M with N, binds v to the decrypted plaintext and executes P (or stops if the decryption fails). `case` and `split` processes destructs tagged and paired messages, respectively. `match` M `is` (N, v) `in` P compares N and the first element of the pair M, and if they are equal, binds v to the second element, and executes P (or stops if not). Although `match` can be implemented by using `split` and `check`, it is given a special typing rule in SpiCA2. Finally, `begin` M and `end` M are correspondence assertions. (The operational semantics of processes is straightforward [5] and omitted in this paper.)

3 The Translation

In this section, we present our translation of narrations in a "top-down" order according to the syntax in Figure 1.

3.1 Translation of protocols

Given the *initial knowledge* I of participants, which is a partial mapping to messages from names $A, B, S, \ldots$ of participants in the narration, a protocol

$\pi = \alpha_1; \ldots; \alpha_n$ is translated to the spi-calculus process $\mathcal{T}(\pi)$ as follows:

$$\begin{aligned}
\mathcal{T}(\pi) = {} & \mathtt{*}\nu p.\ \mathtt{*part!}p \mid \\
& \quad \nu\mathtt{db}.\ \nu\mathtt{dbplus}.\ \nu\mathtt{dbminus}. \\
& \qquad (\mathtt{*part?}p_1.\ \mathtt{part?}p_2.\ \nu K_{p_1p_2}.\ \mathtt{*db!}((p_1,p_2),K_{p_1p_2}) \mid \\
& \qquad \mathtt{*part?}p.\ \nu K_p.\ (\mathtt{*dbplus!}(p,K_p^+) \mid \mathtt{*dbminus!}(p,K_p^-) \mid \mathtt{*net!}K_p^+) \mid \\
& \qquad \textstyle\prod_{X \in dom(I)} \mathcal{T}_X(\pi))
\end{aligned}$$

The first line $\mathtt{*}\nu p.\ \mathtt{*part!}p$ generates an infinite number of names of participants and keeps sending them to the channel `part`. As emphasized in the introduction, our translation assigns multiple roles to each participant; thus, after the translation, the number of participants $p_1, p_2, \ldots$ (which is infinite!) does not match the number of roles $A, B, S, \ldots$.

The second line $\nu\mathtt{db}.\ \nu\mathtt{dbplus}.\ \nu\mathtt{dbminus}$ creates three secret channels `db`, `dbplus`, and `dbminus` for an ideal "key database," represented by the third and fourth lines. The third line then keeps receiving two names of participants ($\mathtt{*part?}p_1.\ \mathtt{part?}p_2$), freshly generates a symmetric key ($\nu K_{p_1p_2}$), and keeps sending it to `db` with the two participant names ($\mathtt{*db!}((p_1,p_2),K_{p_1p_2})$). This process is somewhat different from a realistic key database in that it generates an *infinite* number of $K_{p_1p_2}$ (instead of just one) even for the *same* p_1 and p_2. This discrepancy is okay as far as sound (but incomplete) verification of safety properties (such as no failure of correspondence assertions) is concerned, since *more* behavior is allowed, not less.

Similarly, the fourth line keeps receiving a participant name ($\mathtt{*part?}p$), generates a fresh name (νK_p), and keeps sending the asymmetric key pair to `dbplus` and `dbminus` with the participant name ($\mathtt{*dbplus!}(p,K_p^+) \mid \mathtt{*dbminus!}(p,K_p^-)$) as well as sending the public key to an open network ($\mathtt{*net!}K_p^+$). Again, it is fine for our purpose that the process generates an infinite number of key pairs for each principal.

The last line spawns the translations $\mathcal{T}_A(\pi), \mathcal{T}_B(\pi), \mathcal{T}_S(\pi), \ldots$ (defined below) of each role $A, B, S, \ldots$ (drawn from the domain of the initial knowledge I) in parallel.

3.2 Translation of roles

A role X in protocol $\pi = \alpha_1; \ldots; \alpha_n$ is translated as

$$\begin{aligned}
\mathcal{T}_X(\alpha_1; \ldots; \alpha_n) = {} & \mathtt{*part?}p_1.\ \ldots\ \mathtt{part?}p_m. \\
& \mathcal{T}_X(\rho_1, \alpha_1)(\lambda\rho_2. \\
& \quad \mathcal{T}_X(\rho_2, \alpha_2)(\lambda\rho_3. \\
& \qquad \ldots \\
& \qquad\quad \mathcal{T}_X(\rho_{n-1}, \alpha_{n-1})(\lambda\rho_n. \\
& \qquad\qquad \mathcal{T}_X(\rho_n, \alpha_n)(\lambda\rho_{n+1}. \\
& \qquad\qquad\quad \mathtt{0}))\ldots)) \\
& \text{where } \{Y_1, \ldots, Y_m\} = \{Y \mid Y \in I(X)\} \\
& \text{and } \rho_1 = \{Y_1 \mapsto p_1, \ldots, Y_m \mapsto p_m\}
\end{aligned}$$

where $\mathcal{T}_X(\rho_i, \alpha_i)$ is the translation of action α_i for role X with *knowledge* ρ_i, which is a partial mapping from messages in the narration to messages in the translated process. The translated process first receives the names $p_1, \ldots, p_m$ of principals of role $Y_1, \ldots, Y_m$ (drawn from the initial knowledge $I(X)$ of principals of role X), where the knowledge ρ_1 maps $Y_1, \ldots, Y_m$ to $p_1, \ldots, p_m$ in the rest of the translation. Since the knowledge may increase by each action, the translation $\mathcal{T}_X(\rho_i, \alpha_i)$ of action α_i in fact takes a continuation $\lambda\rho_{i+1}\ldots$ and applies it to the increased knowledge. (We adopt continuation passing style to simplify the definitions.)

3.3 Translation of actions

Action $Y \to Z : M$, Y *begins* M, and Y *ends* M of role X are translated by case analysis on whether Y or Z matches X. On one hand, if $Y = X$, the translated process $\mathcal{S}_X(\rho, M)(\lambda\sigma.(\ldots\sigma^*(M)\ldots c[\sigma]))$ looks up the key database and freshly generate names to compose the message $\sigma^*(M)$ to send, begin, or end (see Section 3.5). On the other hand, if $Z = X$, the process $\texttt{net?}x.\ \mathcal{R}_X(\rho, x, M)c$ checks the received message if it is known, or else adds it to the knowledge (see Section 3.7). In the other cases, the process does nothing, so the continuation c is just applied to the knowledge ρ without change. (We use square brackets [ ] for continuation application.)

$$\begin{array}{lll}
\mathcal{T}_X(\rho, X \to Y : M)c = \mathcal{S}_X(\rho, M)(\lambda\sigma.\ (\texttt{net!}\sigma^*(M) \mid c[\sigma])) & & \text{if } Y \neq X \\
\mathcal{T}_X(\rho, Y \to X : M)c = \texttt{net?}x.\ \mathcal{R}_X(\rho, x, M)c & \text{if } Y \neq X \text{ and } x \text{ fresh} & \\
\mathcal{T}_X(\rho, Y \to Z : M)c \ = c[\rho] & \text{if } Y \neq X \text{ and } Z \neq X & \\
\mathcal{T}_X(\rho, X\ \textit{begins}\ M)c = \mathcal{S}_X(\rho, M)(\lambda\sigma.\ \texttt{begin}\ \sigma^*(M).\ c[\sigma]) & & \\
\mathcal{T}_X(\rho, Y\ \textit{begins}\ M)c = c[\rho] & & \text{if } Y \neq X \\
\mathcal{T}_X(\rho, X\ \textit{ends}\ M)c \ \ = \mathcal{S}_X(\rho, M)(\lambda\sigma.\ (\texttt{end}\ \sigma^*(M) \mid c[\sigma])) & & \\
\mathcal{T}_X(\rho, Y\ \textit{ends}\ M)c \ \ = c[\rho] & & \text{if } Y \neq X
\end{array}$$

3.4 Message composition

The application $\rho^*(M)$ of knowledge ρ to message M is defined just along the structure of M.

$$\begin{array}{lll}
\rho^*(M) & = \rho(M) & \text{if } M \in \mathit{dom}(\rho) \\
\\
\rho^*((M_1, M_2)) & = (\rho^*(M_1), \rho^*(M_2)) & \text{otherwise} \\
\rho^*(\texttt{inl}(M)) & = \texttt{inl}(\rho^*(M)) & \\
\rho^*(\texttt{inr}(M)) & = \texttt{inr}(\rho^*(M)) & \\
\rho^*(M^+) & = (\rho^*(M))^+ & \\
\rho^*(M^-) & = (\rho^*(M))^- & \\
\rho^*(\{M\}_N) & = \{\rho^*(M)\}_{\rho^*(N)} &
\end{array}$$

3.5 Fresh name generation

The fresh name generation, required for output (and `begin`) of an unknown message, is defined below. In the first line, it tries to compose the message only by looking up the key database (see Section 3.6). If this look-up fails, the translation works along the structure of the composed message, as in line 2 to 7. In the last line, a fresh name w is generated for the unknown variable v, and the knowledge ρ is extended with the new mapping $v \mapsto w$.

$$\begin{array}{lll}
\mathcal{S}_X(\rho, M)c & = \mathit{lookup}_X(\rho, M)c & \text{if } \mathit{lookup}_X(\rho, M) \text{ is defined} \\
\\
\mathcal{S}_X(\rho, (M_1, M_2))c & = \mathcal{S}_X(\rho, M_1)(\lambda\sigma.\ \mathcal{S}_X(\sigma, M_2)c) & \text{otherwise} \\
\mathcal{S}_X(\rho, \mathtt{inl}(M))c & = \mathcal{S}_X(\rho, M)c & \\
\mathcal{S}_X(\rho, \mathtt{inr}(M))c & = \mathcal{S}_X(\rho, M)c & \\
\mathcal{S}_X(\rho, M^+)c & = \mathcal{S}_X(\rho, M)c & \\
\mathcal{S}_X(\rho, M^-)c & = \mathcal{S}_X(\rho, M)c & \\
\mathcal{S}_X(\rho, \{M\}_N)c & = \mathcal{S}_X(\rho, N)(\lambda\sigma.\ \mathcal{S}_X(\sigma, M)c) & \\
\mathcal{S}_X(\rho, v)c & = \nu w.\ c[\rho, v \mapsto w] & w \text{ fresh}
\end{array}$$

3.6 Key database look-up

When a parameterized variable $X_{Y_1 \ldots Y_n}$ in the initial knowledge $I(X)$ of principals of role X is needed, the translated process looks it up in the key database as follows. Again, the translation works along the structure of the message M to be composed, as in line 2 to 7 below. In the first line, if M is already composable (i.e., in the knowledge ρ), no look-up is necessary. Otherwise, the key k received from the database is extracted by using `match`, as in the last 6 lines.

$$\begin{array}{lll}
\mathit{lookup}_X(\rho, M)c & = c[\rho] & \text{if } M \in \mathit{dom}(\rho) \\
\\
\mathit{lookup}_X(\rho, (M_1, M_2))c & = \mathit{lookup}_X(\rho, M_1)(\lambda\sigma.\ \mathit{lookup}_X(\sigma, M_2)c) & \text{otherwise} \\
\mathit{lookup}_X(\rho, \mathtt{inl}(M))c & = \mathit{lookup}_X(\rho, M)c & \\
\mathit{lookup}_X(\rho, \mathtt{inr}(M))c & = \mathit{lookup}_X(\rho, M)c & \\
\mathit{lookup}_X(\rho, M^+)c & = \mathit{lookup}_X(\rho, M)c & \\
\mathit{lookup}_X(\rho, M^-)c & = \mathit{lookup}_X(\rho, M)c & \\
\mathit{lookup}_X(\rho, \{M\}_N)c & = \mathit{lookup}_X(\rho, N)(\lambda\sigma.\ \mathit{lookup}_X(\sigma, M)c) & \\
\mathit{lookup}_X(\rho, K_{YZ})c & = \mathtt{db?}x.\ \mathtt{match}\ x\ \mathtt{is}\ ((\rho(Y), \rho(Z)), k)\ \mathtt{in}\ c[\rho, K_{YZ} \mapsto k] & \\
 & & \text{if } K_{YZ} \in I(X) \text{ and } x, k \text{ fresh} \\
\mathit{lookup}_X(\rho, K_Z^+)c & = \mathtt{dbplus?}x.\ \mathtt{match}\ x\ \mathtt{is}\ (\rho(Z), k)\ \mathtt{in}\ c[\rho, K_Z^+ \mapsto k] & \\
 & & \text{if } K_Z^+ \in I(X) \text{ and } x, k \text{ fresh} \\
\mathit{lookup}_X(\rho, K_Z^-)c & = \mathtt{dbminus?}x.\ \mathtt{match}\ x\ \mathtt{is}\ (\rho(Z), k)\ \mathtt{in}\ c[\rho, K_Z^- \mapsto k] & \\
 & & \text{if } K_Z^- \in I(X) \text{ and } x, k \text{ fresh}
\end{array}$$

3.7 Equality checking and knowledge extension

When a message M is received, and if M can also be composed from the knowledge after key database look-ups, their equality with each other is checked (the first clause below). Otherwise, pairs and tagged messages—as well as ciphertexts with known keys—are destructed or decrypted, and the contents are checked (the second to fifth clauses, where $\widehat{N}$ is defined as $\widehat{N^+} = N^-$, $\widehat{N^-} = N^+$, and $\widehat{N} = N$ otherwise). Once the message cannot be checked or destructed any further, it is added to the knowledge of the receiver (the last clause).

$$
\begin{array}{lll}
\mathcal{R}_X(\rho, x, M)c & = \mathit{lookup}_X(\rho, M)(\lambda\sigma. & \\
 & \quad \texttt{check } \sigma^*(M) \texttt{ is } x \texttt{ in } c[\sigma]) & \\
 & & \text{if } \mathit{lookup}_X(\rho, M) \text{ is defined} \\
 & & \\
\mathcal{R}_X(\rho, x, (M_1, M_2))c & = \texttt{split } x \texttt{ is } (y_1, y_2) \texttt{ in} & \text{otherwise} \\
 & \quad \mathcal{R}_X(\rho, y_1, M_1)(\lambda\sigma. & \\
 & \quad\quad \mathcal{R}_X(\sigma, y_2, M_2)c) & y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\rho, x, \texttt{inl}(M))c & = \texttt{case } x \texttt{ is} & \\
 & \quad \texttt{inl}(y_1).\ \mathcal{R}_X(\rho, y_1, M)c\ [\!] & \\
 & \quad \texttt{inr}(y_2).\ \texttt{0} & y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\rho, x, \texttt{inr}(M))c & = \texttt{case } x \texttt{ is} & \\
 & \quad \texttt{inl}(y_1).\ \texttt{0}\ [\!] & \\
 & \quad \texttt{inr}(y_2).\ \mathcal{R}_X(\rho, y_2, M)c & y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\rho, x, \{M\}_N)c & = \mathit{lookup}_X(\rho, \widehat{N})(\lambda\sigma. & \\
 & \quad \texttt{decrypt } x \texttt{ is } \{y\}_{\sigma^*(\widehat{N})} \texttt{ in} & \\
 & \quad\quad \mathcal{R}_X(\sigma, y, M)c) & \\
 & \quad \text{if } \mathit{lookup}_X(\rho, \widehat{N}) \text{ is defined and } y \text{ fresh} & \\
\mathcal{R}_X(\rho, x, M)c & = c[\rho, M \mapsto x] & \text{otherwise}
\end{array}
$$

It is straightforward to add more checks into the translation above, for instance, when a decryption key K^- is received *after* a ciphertext $\{M\}_{K^+}$ or the corresponding encryption key K^+. We omitted such "extra" checks in favor of simplicity of the definition, as they were not necessary for our examples.

4 Example

For the sake of presentation, we use n-ary tuples for $n = 0, 3, \dots$ (in addition to pairs) and pattern matching on them. Let us assume the initial knowledge:

$$
\begin{aligned}
I(A) &= \{A, B, S, K_{AS}\} \\
I(B) &= \{B, S, K_{BS}\} \\
I(S) &= \{S, K_{AS}, K_{BS}\}
\end{aligned}
$$

Note that B does not a priori know A. Note also that S does not know A or B, even though it knows K_{AS} and K_{BS}! This is fine because K_{AS} and K_{BS} will be looked up from the key database by using the names of A and B received at runtime.

Then, the following (broken) version of the Wide Mouthed Frog protocol

1. $A \to S : A$
2. $S \to A : N_S$
3. A *begins* (A, B, K_{AB})
4. $A \to S : \{B, K_{AB}, N_S\}_{K_{AS}}$ [a]
5. $S \to B : ()$
6. $B \to S : N_B$
7. $S \to B : \{A, K_{AB}, N_B\}_{K_{BS}}$
8. B *ends* (A, B, K_{AB})

is translated into

```
*$\nu p$. *part!$p$ |
$\nu$db.
  (*part?$p_1$. part?$p_2$. $\nu K_{p_1 p_2}$. *db!$((p_1, p_2), K_{p_1 p_2})$ |
  $A$ | $B$ | $S$)
```

(for brevity, the database for asymmetric keys is omitted here), where

```
$A$ = *part?$A$. part?$B$. part?$S$.                          (*)
      net!$A$ |
      net?$N_S$.
      $\nu K_{AB}$. begin $(A, B, K_{AB})$.
      db?$x_1$. match $x_1$ is $((A, S), K_{AS})$ in          (***)
      net!$\{B, K_{AB}, N_S\}_{K_{AS}}$ |
      0
$B$ = *part?$B$. part?$S$.                                    (*)
      net?().
      $\nu N_B$. net!$N_B$ |
      net?$c_1$.
      db?$x_2$. match $x_2$ is $((B, S), K_{BS})$ in          (***)
      decrypt $c_1$ is $\{A, K_{AB}, N'_B\}_{K_{BS}}$ in      (**)
      check $N'_B$ is $N_B$ in
      end $(A, B, K_{AB})$ |
      0
$S$ = *part?$S$.                                              (*)
      net?$A$.                                                (**)
      $\nu N_S$. net!$N_S$ |
      net?$c_2$.
      db?$x_3$. match $x_3$ is $((A, S), K_{AS})$ in          (***)
      decrypt $c_2$ is $\{B, K_{AB}, N'_S\}_{K_{AS}}$ in      (**)
      check $N'_S$ is $N_S$ in
      net!() |
      net?$N_B$.
      db?$x_4$. match $x_4$ is $((B, S), K_{BS})$ in          (***)
      net!$\{A, K_{AB}, N_B\}_{K_{BS}}$ |
      0
```

The following are highlights of this translation:

(*) On one hand, the process A is parametrized by the names A, B, and S; similarly, process B is parameterized by names B and S, and process S by name S.

(**) On the other hand, process B learns name A *during* the run of the protocol; similarly, S learns A and B at runtime.

(***) Accordingly, the symmetric key K_{AS} (resp. K_{BS}) shared between A and S (resp. B and S) is looked up from the database at runtime.

5 Experiments

We tested the validity of our translation by verifying its results with SpiCA2 [5], a sound and automatic type-based verifier of correspondence assertions. From the WWW site of SpiCA2 (`http://www.kb.is.s.u-tokyo.ac.jp/~koba/spica2/`), we took 5 protocols using symmetric encryption and 12 using asymmetric.

The results are given in the Table 1 (at the end of the paper, for the sake of page breaks). The columns “expected” and “actual” show the expected and actual results, respectively. “Safe” means that type checking succeeded (i.e., the correspondence assertions would never fail), while “unsafe” means that it failed.

All the actual results match expected ones except for the two “not simply-typed.” They are due to the fact that our translation uses the same public key K^+ for both encryption and signature verification (and the same secret key K^- for both decryption and signing), which does not fit (the “simple” part of) the present type system of SpiCA2. It should be straightforward to adapt the latter to the former (or vice versa).

6 Extension with malicious participants

It is well known that some protocols such as (asymmetric-key version of) Needham-Schroeder [10] are vulnerable to an insider attack, i.e., unsafe when one of the principals is malicious. However, our translation above does not allow such attacks because the channels `db`, `dbplus`, and `dbminus` for the key database are private, i.e., the attacker cannot share any keys with the (good) principals, meaning that it cannot participate in the protocol at all.

To get rid of this limitation, we extend the translation with “bad” participants as follows. First, we separate the name set of bad participants from that of good ones, writing $\mathcal{N}^{bad}$ for the former and $\mathcal{N}^{good}$ for the latter. In the actual translation to SpiCA2, this separation is implemented just by adding an `inl` (for *bad*) or `inr` (for *good*) tag to each name.

The translation of a protocol π then becomes (the changes are underlined):

$$\begin{array}{l}
\mathcal{T}(\pi) = \underline{*\nu p \in \mathcal{N}^{good}.\ *\mathtt{part!}p} \mid \underline{*\nu p \in \mathcal{N}^{bad}.\ *\mathtt{part!}p} \mid \\
\quad \nu\mathtt{db}.\ \nu\mathtt{dbplus}.\ \nu\mathtt{dbminus}. \\
\qquad (*\mathtt{part?}p_1.\ \mathtt{part?}p_2.\ \nu K_{p_1p_2}.\ (*\mathtt{db!}((p_1,p_2),K_{p_1p_2}) \mid \\
\qquad\quad \underline{\mathtt{if}\ p_1 \in \mathcal{N}^{bad} \vee p_2 \in \mathcal{N}^{bad}\ \mathtt{then}\ *\mathtt{net!}K_{p_1p_2}}) \mid \\
\qquad *\mathtt{part?}p.\ \nu K_p.\ (*\mathtt{dbplus!}(p, K_p^+) \mid *\mathtt{dbminus!}(p, K_p^-) \mid *\mathtt{net!}K_p^+ \mid \\
\qquad\quad \underline{\mathtt{if}\ p \in \mathcal{N}^{bad}\ \mathtt{then}\ *\mathtt{net!}K_p^-}) \mid \\
\qquad \prod_{X \in dom(I)} \mathcal{T}_X(\pi))
\end{array}$$

The first line generates two kinds of participant names rather than just one. The fourth and sixth lines publish “private” keys if they belong to bad participants so that the attacker can use them.

Then, the translation of a protocol $\pi = \alpha_1; \ldots; \alpha_n$ for principals of role X is

$$\begin{array}{l}
\mathcal{T}_X(\alpha_1; \ldots; \alpha_n) = *\mathtt{part?}p_1.\ \ldots\ \mathtt{part?}p_m. \\
\qquad\qquad \mathcal{T}_X(\underline{b_1}, \rho_1, \alpha_1)(\lambda(\underline{b_2}, \rho_2). \\
\qquad\qquad\quad \mathcal{T}_X(\underline{b_2}, \rho_2, \alpha_2)(\lambda(\underline{b_3}, \rho_3). \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\quad \mathcal{T}_X(\underline{b_{n-1}}, \rho_{n-1}, \alpha_{n-1})(\underline{\lambda b_n}, \rho_n. \\
\qquad\qquad\qquad\qquad \mathcal{T}_X(\underline{b_n}, \rho_n, \alpha_n)(\lambda(\underline{b_{n+1}}, \rho_{n+1}). \\
\qquad\qquad\qquad\qquad\quad \mathtt{0})) \ldots)) \\
\qquad\qquad \text{where } \underline{b_1 = (\{p_1, \ldots, p_m\} \subseteq \mathcal{N}^{good})} \\
\qquad\qquad \text{and } \{Y_1, \ldots, Y_m\} = \{Y \mid Y \in I(X)\} \\
\qquad\qquad \text{and } \rho_1 = \{Y_1 \mapsto p_1, \ldots, Y_m \mapsto p_m\}
\end{array}$$

where the translation of each action α_i passes around a Boolean value b_i that represents whether *all* participants involved in the current session is good. This is necessary because, if any of the participants is bad, we will never execute any `end` assertion in this session since there is no hope that the bad participant executes the corresponding `begin` assertion. The rest of the changes are thus (the other definitions remain unchanged):

$$\begin{array}{lll}
\mathcal{T}_X(\underline{b}, \rho, X \to Y : M)c & = \mathcal{S}_X(\rho, M)(\lambda\sigma.\ (\mathtt{net!}\sigma^*(M) \mid c[(\underline{b}, \sigma)])) & \text{if } Y \neq X \\
\mathcal{T}_X(\underline{b}, \rho, Y \to X : M)c & = \mathtt{net?}x.\ \mathcal{R}_X(\underline{b}, \rho, x, M)c & \text{if } Y \neq X \text{ and } x \text{ fresh} \\
\mathcal{T}_X(\underline{b}, \rho, Y \to Z : M)c & = c[(\underline{b}, \rho)] & \text{if } Y \neq X \text{ and } Z \neq X \\
\mathcal{T}_X(\underline{b}, \rho, X\ \mathit{begins}\ M)c & = \mathcal{S}_X(\rho, M)(\lambda\sigma.\ \mathtt{begin}\ \sigma^*(M).\ c[(\underline{b}, \sigma)]) & \\
\mathcal{T}_X(\underline{b}, \rho, Y\ \mathit{begins}\ M)c & = c[(\underline{b}, \rho)] & \text{if } Y \neq X \\
\mathcal{T}_X(\underline{b}, \rho, X\ \mathit{ends}\ M)c & = \underline{\mathtt{if}\ b = \mathtt{false}\ \mathtt{then}\ c[(b, \rho)]\ \mathtt{else}} & \\
 & \quad \mathcal{S}_X(\rho, M)(\lambda\sigma.\ (\mathtt{end}\ \sigma^*(M) \mid c[(\underline{b}, \sigma)])) & \\
\mathcal{T}_X(\underline{b}, \rho, Y\ \mathit{ends}\ M)c & = c[(\underline{b}, \rho)] & \text{if } Y \neq X
\end{array}$$

$$
\begin{array}{lll}
\mathcal{R}_X(\underline{b}, \rho, x, M)c & = \mathit{lookup}_X(\rho, M)(\lambda\sigma. & \\
 & \quad \texttt{check}\ \sigma^*(M)\ \texttt{is}\ x\ \texttt{in}\ c[(\underline{b}, \sigma)]) & \\
 & \qquad\qquad \text{if } \mathit{lookup}_X(\rho, M) \text{ is defined} & \\
 & & \\
\mathcal{R}_X(\underline{b}, \rho, x, (M_1, M_2))c & = \texttt{split}\ x\ \texttt{is}\ (y_1, y_2)\ \texttt{in} & \text{otherwise} \\
 & \quad \mathcal{R}_X(\underline{b}, \rho, y_1, M_1)(\lambda(\underline{b'}, \sigma). & \\
 & \qquad \mathcal{R}_X(\underline{b'}, \sigma, y_2, M_2)c) & y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\underline{b}, \rho, x, \texttt{inl}(M))c & = \texttt{case}\ x\ \texttt{is} & \\
 & \quad \texttt{inl}(y_1).\ \mathcal{R}_X(\underline{b}, \rho, y_1, M)c\ [\!] & \\
 & \quad \texttt{inr}(y_2).\ \texttt{0} & y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\underline{b}, \rho, x, \texttt{inr}(M))c & = \texttt{case}\ x\ \texttt{is} & \\
 & \quad \texttt{inl}(y_1).\ \texttt{0}\ [\!] & \\
 & \quad \texttt{inr}(y_2).\ \mathcal{R}_X(\underline{b}, \rho, y_2, M)c & y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\underline{b}, \rho, x, \{M\}_N)c & = \mathit{lookup}_X(\rho, \widehat{N})(\lambda\sigma. & \\
 & \quad \texttt{decrypt}\ x\ \texttt{is}\ \{y\}_{\sigma^*(\widehat{N})}\ \texttt{in} & \\
 & \qquad \mathcal{R}_X(\underline{b}, \sigma, y, M)c) & \\
 & \qquad\qquad \text{if } \mathit{lookup}_X(\rho, \widehat{N}) \text{ is defined and } y \text{ fresh} & \\
\mathcal{R}_X(\underline{b}, \rho, x, \underline{A})c & = c[(\underline{(x \in \mathcal{N}^{good}) \wedge b}, (\rho, A \mapsto x))] & \underline{\text{if } A \in \mathit{dom}(I)} \\
\mathcal{R}_X(\underline{b}, \rho, x, M)c & = c[(\underline{b}, (\rho, M \mapsto x))] & \text{otherwise}
\end{array}
$$

It requires some trick to make SpiCA2 accept this translation: as mentioned above, the distinction of "bad" participant names $\mathcal{N}^{bad}$ and "good" ones $\mathcal{N}^{good}$ can be implemented by tagging, but then it often becomes the case that the type of an element of a tuple depends on the *tag* of another element of the same tuple; for instance, in the ciphertext $\{S, A, B, K_{AB}, N_B\}_{K_{BS}}$ of message 6 of the first protocol in Section 1, K_{AB} may be private or public, depending on whether A is good or bad, i.e., tagged by `inl` or `inr`. Such dependency is beyond the power of standard dependent type system as in SpiCA2. To address this problem, we move all `inl` and `inr` tags to the outside of tuples as far as possible (e.g., rewriting $\{\texttt{inl}(A), K_{AB}\}_{K_{BS}}$ to $\{\texttt{inl}(A, K_{AB})\}_{K_{BS}}$) and "normalize" (strange) dependent sums like $\Sigma x : \mathcal{N}^{bad} + \mathcal{N}^{good}.\ \texttt{if}\ x \in \mathcal{N}^{bad}\ \texttt{then}\ \mathit{public}\ \texttt{else}\ \mathit{private}$ to simple sums like $(\mathcal{N}^{bad} \times \mathit{public}) + (\mathcal{N}^{good} \times \mathit{private})$, roughly speaking.

With this trick, the results in Table 1 remain unchanged even under the presence of malicious participants. This is somewhat surprising because the extended translation allows more attacks. We conjecture that this is only a coincidence of the particular examples of protocols and the type system of SpiCA2, but further investigation is due.

7 Conclusions

We developed an interpretation of protocol narrations as a translation into the spi-calculus, and tested its validity by means of correspondence assertions and their verification.

From the translation, it is obvious that the full power of spi-calculus is not used. One may therefore argue that the target language of the translation can be simplified. While this is true, we believe that our translation into the spi-calculus (with correspondence assertions) is already simple enough. Moreover, the full power of spi-calculus would be useful for the *attacker* and the *environment* of a protocol.

Another natural question is whether our translation is "correct." Since there is no standard formal semantics of protocol narrations,[2] and since our translation is a *definition* of the meaning of protocol narrations, trying to *prove* its correctness seems pointless. However, a more direct semantics of protocol narrations is indeed desirable.

Security properties other than correspondence assertions (authenticity)—such as secrecy [1]—should also be considered in future work.

References

1. Abadi, M.: Secrecy by typing in security protocols. Journal of the ACM 46(5), 749–786 (1999), preliminary version appeared in *Theoretical Aspects of Computer Software*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1281, pp. 611–638, 1997
2. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1), 1–70 (1999), preliminary version appeared in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pp. 36–47, 1997
3. Briais, S., Nestmann, U.: A formal semantics for protocol narrations. Theoretical Computer Science 389(3), 484–511 (2007), preliminary version appeared in *Trustworthy Global Computing*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 3705, pp. 163–181, 2005
4. Clark, J., Jacob, J.: A survey of authentication protocol literature: Version 1.0. `http://www-users.cs.york.ac.uk/~jac/papers/drareview.ps.gz` (1997)
5. Dahl, M., Kobayashi, N., Sun, Y., Hüttel, H.: Type-based automated verification of authenticity in asymmetric cryptographic protocols. In: Proceedings of the 9th international conference on Automated technology for verification and analysis. Lecture Notes in Computer Science, vol. 6996, pp. 75–89. Springer-Verlag (2011)
6. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
7. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. Journal of Computer Security (2003), to appear. Extended abstract appeared in *14th IEEE Computer Security Foundations Workshop*, pp. 145–159, 2001.
8. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. Journal of Computer Security (2004), to appear. Extended abstract appeared in *15th IEEE Computer Security Foundations Workshop*, pp. 77–91, 2002.

[2] Briais and Nestmann [3] gave a formal semantics by translation into "executable narrations" which "closely correspond to terms in a quite restricted fragment of the spi-calculus" (p. 500). Independently, Sumii et al. [11] defined a (very) similar translation into an (again similar) subset of the spi-calculus, as outlined in Section 1. The former did not consider key databases (nor, in the first place, principals parametrized by the names of other principals, like the server S in Section 1), and neither of them considered multiple roles for a single principal.

9. Menezes, A.J., van Oorshot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1996)
10. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. Communications of the ACM 21(12), 993–999 (1978)
11. Sumii, E., Tatsuzawa, H., Yonezawa, A.: Translating security protocols from informal notation into spi calculus. IPSJ Transactions on Programming 45(SIG 12(PRO 23)), 1–10 (2004), abstract and figures in English, main text in Japanese
12. Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols. In: IEEE Symposium on Security and Privacy. pp. 178–194 (1993)

Table 1. Results of experiments with SpiCA2

Protocol	Expected	Actual
A simple handshake using a symmetric key	safe	safe
Woo and Lam's authentication protocol using a symmetric key	safe	safe
Otway and Ree's key exchange protocol using a symmetric key	safe	safe
Flawed wide mouth frog protocol	unsafe	unsafe
Fixed variant of wide mouth frog protocol	safe	safe
POSH (public out, secret home) protocol using an asymmetric key	safe	safe
SOPH (secret out, public home) protocol using an asymmetric key	safe	safe
SOSH (secret out, secret home) protocol using an asymmetric key	safe	safe
A three-party protocol that cannot be typed in Gordon and Jeffrey's type system	safe	not simply-typed
Cremers and Mauw's generalized Needham-Schroeder-Lowe protocol	safe	safe
ISO Public Key Two-Pass Unilateral Authentication Protocol	safe	safe
Needham-Schroeder protocol (flawed, hence untypable)	unsafe	unsafe
Needham-Schroeder-Lowe protocol (Lowe's fix, 3-message version)	safe	safe
Needham-Schroeder-Lowe protocol (Lowe's fix, 7-message version)	safe	not simply-typed
NSL protocol (optimized version)	safe	safe
NSL protocol (with secret)	safe	safe
NSL protocol (with secret and optimization)	safe	safe