Authenticity by Typing for Security Protocols

Andrew D. Gordon & Allan Jeffrey

presented by Vassilis Koutavas
Authenticity

Msg. 1  $A \rightarrow B : \{M\}_K$

• Assumptions of the protocol:
  – $A$ and $B$ share a key $k$
  – Messages are sent over a distributed network

• Authenticity Proper:
  $B$ can be sure that the message comes from $A$.

• Question: Can we prove that this protocol satisfies (or not) the authenticity property?
Authenticity

Msg. 1  A $\rightarrow$ B : $\{M\}_K$

- **Assumptions** of the protocol:
  - A and B share a key K
  - Messages are sent over an **distrusted network**
  - **Perfect encryption**

- Question: Can we prove that this protocol satisfies (or not) the authenticity property?
Authenticity

Msg. 1  $A \rightarrow B : \{M\}_K$

- **Assumptions** of the protocol:
  - $A$ and $B$ share a key $K$
  - Messages are sent over an **distrusted network**
  - **Perfect encryption**

- **Authenticity Property**: $B$ can be sure that the message comes from $A$. 
Authenticity

Msg. 1  $A \rightarrow B : \{M\}_K$

- **Assumptions** of the protocol:
  - $A$ and $B$ share a key $K$
  - Messages are sent over an **distrusted network**
  - **Perfect encryption**
- **Authenticity Property**: $B$ can be sure that the message comes from $A$.
- **Question**: How can we **prove** that this protocol satisfies (or not) the authenticity property?
Authenticity

\[\text{Msg. 1} \quad A \rightarrow B : \{M\}_K\]

\[\downarrow \uparrow\]

\[C\]

... Authenticity Property is not satisfied
Authenticity

Msg. 1  A → B : \{M\}_K

↓↑

C

Msg. 2  C → B: \{M\}_K
Msg. 3  C → B: \{M\}_K
...

• Authenticity Property is not satisfied
Authenticity

Msg. 1  A → B : \{M\}_K

↓↑

C

Msg. 2  C → B: \{M\}_K

Msg. 3  C → B: \{M\}_K

...

• Authenticity Property is not satisfied
Gordon & Jeffrey’s Idea

• Write protocols in a version of Spi-Calculus [Abadi & Gordon]
• Specify authenticity properties by annotating the code with correspondence assertions [Woo & Lam]
• Figure out types for the keys, nonces, and messages
• Check that the code is well-typed according to a type and effect system
• Theorem: Well-typed code is robustly safe
Benefits - Drawbacks

✓ Requires **little human effort**
✓ Type-checking is done **automatically**
✓ It is **decidable**
✓ It proves the desired properties in the presence of an **opponent of any size**
✓ **Doesn’t need to enumerate all the states** of the protocols
× It considers only **opponents that can be expressed in Spi-calculus**
× It gives **false-negatives**
Correspondence Assertions

• Express the protocol in some formal way
  – e.g. Spi-calculus

• Annotate the code with the right assertions
  – begin-events, end-events
  – Usually each valid run of the protocol should have a begin-event (in the initiator) and an end-event (in the responder)

• Prove that all runs of the protocol (in the presence of an adversary) satisfy the assertions
  – For each end-event there is a begin-event.
Correspondence Assertions in Spi-Calculus

\[ P ::= \begin{array}{c}
\text{begin } L; P \mid \text{end } L; P \\
\mid \text{out } M N \mid \text{inp } M (x:T); P \\
\mid \text{new } (x:T); P \mid \text{decrypt } M \text{ is } \{x:T\}_K; P \\
\mid \text{check } M \text{ is } N; P \\
\mid \text{repeat } P \mid (P \mid P) \mid \ldots
\end{array} \]

\[ L, M, N, K ::= x \mid \{M\}_K \mid \ldots \]
Correspondence Assertions in Spi-Calculus

\[
P ::= \begin{align*}
& \text{begin } L; \ P \ | \ \text{end } L; \ P \\
& \mid \ \text{out } M \ N \ | \ \text{inp } M \ (x:T); \ P \\
& \mid \ \text{new } (x:T); \ P \mid \ \text{decrypt } M \text{ is } \{x:T\}_K; \ P \\
& \mid \ \text{check } M \text{ is } N; \ P \\
& \mid \ \text{repeat } P \mid (P \mid P) \mid \ldots
\end{align*}
\]

L, M, N, K ::= x \mid \{M\}_K \mid \ldots
Correspondence Assertions in Spi-Calculus

P ::= begin L; P  |  end L; P
 |  out M N  |  inp M (x:T); P
 |  new (x:T); P  |  decrypt M is \{x:T\}_K; P
 |  check M is N; P
 |  repeat P  |  (P | P)  |  ...

L, M, N, K ::= x  |  \{M\}_K  |  ...
Correspondence Assertions in Spi-Calculus

\[ P ::= \begin{align*}
  &\text{begin } L; P \mid \text{end } L; P \\
  &\mid \text{out } M N \mid \text{inp } M \ (x:T); P \\
  &\mid \text{new } (x:T); P \mid \text{decrypt } M \text{ is } \{x:T\}_K; P \\
  &\mid \text{check } M \text{ is } N; P \\
  &\mid \text{repeat } P \mid (P \mid P) \mid \ldots
\end{align*} \]

\[ L, M, N, K ::= x \mid \{M\}_K \mid \ldots \]
Correspondence Assertions in Spi-Calculus

\[ P ::= \text{begin } L; P \mid \text{end } L; P \]
\[ \mid \text{out } M N \mid \text{inp } M (x:T); P \]
\[ \mid \text{new } (x:T); P \mid \text{decrypt } M \text{ is } \{x:T\}_K; P \]
\[ \mid \text{check } M \text{ is } N; P \]
\[ \mid \text{repeat } P \mid (P \mid P) \mid \ldots \]

\[ L, M, N, K ::= x \mid \{M\}_K \mid \ldots \]
Correspondence Assertions in Spi-Calculus

$$P ::= \text{begin } L; P \mid \text{end } L; P$$

$$\mid \text{out } M N \mid \text{inp } M (x:T); P$$

$$\mid \text{new } (x:T); P \mid \text{decrypt } M \text{ is } \{x:T\}^K; P$$

$$\mid \text{check } M \text{ is } N; P$$

$$\mid \text{repeat } P \mid (P \mid P) \mid \ldots$$

$$L, M, N, K ::= x \mid \{M\}^K \mid \ldots$$
Correspondence Assertions in Spi-Calculus

\[ P ::= \text{begin } L; \ P \ | \ \text{end } L; \ P \]
\[ \ | \ \text{out } M \ N \ | \ \text{inp } M \ (x:T); \ P \]
\[ \ | \ \text{new} \ (x:T); \ P \ | \ \text{decrypt} \ M \text{ is } \{x:T\}_K; \ P \]
\[ \ | \ \text{check} \ M \text{ is } N; \ P \]
\[ \ | \ \text{repeat} \ P \ | \ (P \ | \ P) \ | \ ... \]

\[ L, M, N, K ::= x \ | \ \{M\}_K \ | \ ... \]
Correspondence Assertions in Spi-Calculus

\[ P ::= \text{begin } L; P \mid \text{end } L; P \]
\[ \mid \text{out } M N \mid \text{inp } M (x: T); P \]
\[ \mid \text{new } (x: T); P \mid \text{decrypt } M \text{ is } \{x: T\}_K; P \]
\[ \mid \text{check } M \text{ is } N; P \]
\[ \mid \text{repeat } P \mid (P \mid P) \mid \ldots \]

\[ L, M, N, K ::= x \mid \{M\}_K \mid \ldots \]
Opponents and Safety

**Def:** P is **safe** iff
for every run of P
for every L
there is a distinct **begin L** for every **end L**

**Def:** An **opponent** O is an assertion-free (untyped) process

**Def:** P is **robustly safe** iff
for every opponent O
(P | O) is safe
The Simple Protocol, Revisited

Msg. 1  A → B : \{M\}_K

Sender(net, key) =
repeat
new (msg);
out net \{msg\}_key

Receiver(net, key) =
repeat
inp net (ctext);
decrypt ctext is \{msg\}_key

System(net) = new (key);
(Sender(net, key) | Receiver(net, key))
The Simple Protocol, Revisited

Event 1: A begins M
Msg. 1 \( A \rightarrow B : \{M\}_K \)
Event 2: B ends M

Sender(net, key) =
repeat
new (msg);
begin msg;
out net \{msg\}_key

Receiver(net, key) =
repeat
inp net (ctext);
decrypt ctext is \{msg\}_key;
end msg

System(net) = new (key);
(Sender(net, key) | Receiver(net, key))
The Simple Protocol, Revisited

**Event 1** : A begins M

**Msg. 1**  \( A \to B : \{M\}_K \)

**Event 2** : B ends M

Sender(net, key) =

repeat
    new (msg);
    begin msg;
    out net \{msg\}_key

Receiver(net, key) =

repeat
    inp net (ctext);
    decrypt ctext is \{msg\}_key;
    end msg

System(net) = new (key);

\( (\text{Sender(net, key)} | \text{Receiver(net, key)} | \text{Op(net)}) \)

\textbf{Op(net)} = inp net (ctext); out net ctext; out net ctext
The Simple Protocol, Revisited

Event 1: A begins M
Msg. 1  B → A: N_{fresh}
Msg. 2  A → B: \{M, N_{fresh}\}_K

Event 2: B ends M

Sender(net, key) =
repeat
  new (msg);
  begin msg;
  inp net (nonce);
  out net \{msg, nonce\}_key

Receiver(net, key) =
repeat
  new (nonce);
  out net nonce;
  inp net (ctext);
  decrypt ctext is \{msg, nc\}_key;
  check nonce is nc;
  end msg
Type and Effect System

The types:

- Untrusted type **Un**
  - type of adversaries, messages on public channels
- Shared-key types **Key(T)**
- Channel Types **Ch(T)**
- ...

M: Ch(Un)
N: Un

M: Ch(T), N: Key(T) {M}N: Un

M: T, N: Key(T) {M}N: Un

decrypt M is {x:T}N; P is WT
Type and Effect System

The types:

- **Untrusted type** `Un`  
  - type of adversaries, messages on public channels
- **Shared-key types** `Key(T)`
- **Channel Types** `Ch(T)`
- ...

\[
\begin{align*}
M: Ch(Un), & \quad N: Un \Rightarrow \text{out } M \text{ N is well-typed (WT)} \\
M: Ch(T), & \quad P \text{ is WT } \Rightarrow \text{inp } M (x:T); \quad P \text{ is WT} \\
M: T, & \quad N: Key(T) \Rightarrow \{M\}_N: Un \\
M: Un, & \quad N: Key(T), \quad P \text{ is WT } \Rightarrow \text{decrypt } M \text{ is } \{x:T\}_N; \quad P \text{ is WT}
\end{align*}
\]
The effects (the types of processes):

- Atomic end-effect: \texttt{end L}
  
  \[ P: [\text{end } L_1, \text{end } L_2, \ldots, \text{end } L_n] \]

- ...
The effects (the types of processes):

- Atomic end-effect: $\text{end L}$
  
  \[ P: [\text{end } L_1, \text{end } L_2, \ldots, \text{end } L_n] \]

- ...

\[ P: \text{es} \Rightarrow \text{end } L; P: (\text{es} + [\text{end } L]) \]
\[ P: \text{es} \Rightarrow \text{begin } L; P: (\text{es} - [\text{end } L]) \]
\[ P: \text{es}_P, Q: \text{es}_Q \Rightarrow P|Q: (\text{es}_P + \text{es}_Q) \]
Type and Effect System

The effects (the types of processes):
• Atomic end-effect: \texttt{end L}
  \[ P: [\texttt{end L}_1, \texttt{end L}_2, \ldots, \texttt{end L}_n] \]
• ...

\[ P: \texttt{es} \Rightarrow \texttt{end L}; \hspace{1em} P: (\texttt{es} + [\texttt{end L}]) \]
\[ P: \texttt{es} \Rightarrow \texttt{begin L}; \hspace{1em} P: (\texttt{es} - [\texttt{end L}]) \]
\[ P: \texttt{es}_P, Q: \texttt{es}_Q \Rightarrow P|Q : (\texttt{es}_P + \texttt{es}_Q) \]

**Theorem:** if \( P: [\ ] \) then \( P \) is robustly safe!
Typing the Sender

Sender(net, key) =
repeat
new (msg);
begin msg;
inp net (nonce);
out net {msg, nonce}key
Typing the Sender

Sender(net: \textcolor{red}{Un}, key: \textcolor{blue}{Key(T)}) =
repeat
    new (msg);
    begin msg;
    inp net (nonce);
    out net \{msg, nonce\}_key
Typing the Sender

Sender(net: \textbf{Un}, key: \textbf{Key(T)}) =
  repeat
    new (msg: \textbf{MsgT});
    begin msg;
    inp net (nonce);
    out net \{msg, nonce\}_{key}
Typing the Sender

Sender(net: \textbf{Un}, key: \textbf{Key(T)}) =
repeat
new (msg: \textbf{MsgT});
begin msg;
inp net (nonce: \textbf{Un});
out net \{msg, nonce\}_key
Typing the Sender

Sender(net: Un, key: Key(MsgT, Un)) =
repeat
new (msg: MsgT);
begin msg;
inp net (nonce: Un);
out net {msg, nonce}key
Typing the Sender

Sender(net: \textbf{Un}, key: \textbf{Key(MsgT,Un)}) =

repeat
  new (msg: \textbf{MsgT});
  begin msg;
  inp net (nonce: \textbf{Un});
  \textcolor{red}{\textbf{out net \{msg, nonce\}_key : [ ]}}
Typing the Sender

Sender(net: Un, key: Key(MsgT, Un)) =
repeat
  new (msg: MsgT);
  begin msg;
  inp net (nonce: Un);
  out net {msg, nonce}key
: [ ]
Typing the Sender

Sender(net: \textbf{Un}, key: \textbf{Key}(\textbf{MsgT}, \textbf{Un})) =

\begin{verbatim}
repeat
  new (msg: \textbf{MsgT});
  \textbf{begin msg;}
  inp net (nonce: \textbf{Un});
  out net \{msg, nonce\}\_key
\end{verbatim}

\textbf{: [ ]} – [end msg] = [ ]
Typing the Sender

Sender(net: \texttt{Un}, key: \texttt{Key(MsgT,Un)}) =

\begin{verbatim}
repeat
    new (msg: \texttt{MsgT});
    begin msg;
    inp net (nonce: \texttt{Un});
    out net \{msg, nonce\}_key
\end{verbatim}
Typing the Receiver

Receiver(net, key) =
  repeat
    new (nonce);
    out net nonce;
    inp net (ctext);
    decrypt ctext is \{msg, nc\}_key;
    check nonce is nc;
  end msg
Typing the Receiver

Receiver(net: **Un**, key: **Key(MsgT,Un)**) =

  repeat
    new (nonce);
    out net nonce;
    inp net (ctext);
    decrypt ctext is \{msg, nc\}_key;
    check nonce is nc;
  end msg
Typing the Receiver

Receiver(net:\textbf{Un}, key:\textbf{Key(MsgT,Un)}) = 
repeat
  new (nonce:\textbf{Un});
  out net nonce;
  inp net (ctext);
  decrypt ctext is \{msg, nc\}_key;
  check nonce is nc;
end msg
Typing the Receiver

Receiver(net: \texttt{Un}, key: \texttt{Key(MsgT,Un)}) =
repeat
  new (nonce: \texttt{Un});
  out net nonce;
  inp net (ctext: \texttt{Un});
  decrypt ctext is \{msg, nc\}_key;
  check nonce is nc;
end msg
Typing the Receiver

Receiver(net: \texttt{Un}, key: \texttt{Key(MsgT,Un)}) =
\begin{align*}
\text{repeat} \\
\text{new (nonce: Un);} \\
\text{out net nonce;} \\
\text{inp net (ctext: Un);} \\
\text{decrypt ctext is } \{\text{msg: MsgT, nc: Un}\}_\text{key}; \\
\text{check nonce is nc;} \\
\text{end msg}
\end{align*}
Typing the Receiver

Receiver(net: \textbf{Un}, key: \textbf{Key}(MsgT,Un)) =
repeat
new (nonce: \textbf{Un});
out net nonce;
inp net (ctext: \textbf{Un});
decrypt ctext is \{msg: \textbf{MsgT}, nc: \textbf{Un}\}_key;
check nonce is nc;
end msg : [end msg]
Typing the Receiver

Receiver(net: \texttt{Un}, key: \texttt{Key(MsgT,Un)}) =

repeat
    new (nonce: \texttt{Un});
    out net nonce;
    inp net (ctext: \texttt{Un});
    decrypt ctext is \{msg: \texttt{MsgT}, nc: \texttt{Un}\}_\texttt{key};
    check nonce is nc;
end msg

: [end msg]
Typing the Receiver

Receiver(net: **Un**, key: **Key(MsgT, Un)**) =

repeat

  new (nonce: **Un**);
  out net nonce;
  inp net (ctext: **Un**);
  decrypt ctext is {msg: **MsgT**, nc: **Un**}key;
  check nonce is nc;
end msg
Typing the Receiver

Receiver(net: $Un$, key: $Key(MsgT, Un)$) =

repeat
  new (nonce: $Un$);
  out net nonce;
  inp net (ctext: $Un$);
  decrypt ctext is \{msg: $MsgT$, nc: $Un$\}_{key};
  check nonce is nc;
end msg

: [end msg]
Typing the System

System(net) = new (key);
   (Sender(net, key) | Receiver(net, key))
Typing the System

System(net: Un) = new (key: \texttt{Key(MsgT,Un)});
(Sender(net, key) | Receiver(net, key))
Typing the System

System(net: Un) = new (key: Key(MsgT, Un));
(Sender(net, key) | Receiver(net, key))
: [ ] : [end msg]
Typing the System

System(net: \textbf{Un}) = new (key: \textbf{Key}(\textbf{MsgT, Un}));
\hspace{1cm}(\text{Sender}(\text{net, key}) \mid \text{Receiver}(\text{net, key}))

: [end msg]
Typing the System

System(net: **Un**) = new (key: **Key(MsgT,Un)**);
\[(Sender(net, key) \mid Receiver(net, key))\]

: [end msg]

**Problem:** We need to show **temporal precedences** between parallel processes
Typing the System

System(net:\textbf{Un}) = new (key:\textbf{Key(MsgT,Un)});
(Sender(net, key) | Receiver(net, key))

\textbf{: [end msg]}

\textbf{Problem:} We need to show \textbf{temporal precedences} between parallel processes
– These are guaranteed by the \textbf{nonce handshakes}
Typing the System

System(net: Un) = new (key: Key(MsgT, Un));
(Sender(net, key) | Receiver(net, key))

: [end msg]

Problem: We need to show temporal precedences between parallel processes
- These are guaranteed by the nonce handshakes
- This paper covers only a particular idiom of handshakes: incoming handshakes
Extending the Type System

They need:

- One more atomic effect: check L
- One more type: Nonce es
Typing the Sender and Receiver (again)

Sender(net: Un, 
   key: Key(MsgT, Un)) =
   repeat
      new (msg: MsgT);
   begin msg;
      inp net (nonce: Un);
      out net {msg, nonce}_key
   end msg

Receiver(net: Un, 
   key: Key(MsgT, Un)) =
   repeat
      new (nonce: Un);
      out net nonce;
      inp net (ctext: Un);
      decrypt ctext
      is {msg: MsgT, nc: Un}_key;
      check nonce is nc;
      end msg
Typing the Sender and Receiver (again)

Sender(net: \texttt{Un},
key: \texttt{Key(MsgT,Un)}) =
repeat
    new (msg: \texttt{MsgT});
    begin msg;
    inp net (nonce: \texttt{Un});
    out net \{msg, nonce\}_\texttt{key}

Receiver(net: \texttt{Un},
key: \texttt{Key(MsgT,Un)}) =
repeat
    new (nonce: \texttt{Un});
    out net nonce;
    inp net (ctext: \texttt{Un});
    decrypt ctext
    is \{msg: \texttt{MsgT}, nc: \texttt{Un}\}_\texttt{key};
    check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net: \textbf{Un},
key: Key(MsgT, Un)) =
repeat
    new (msg: \textbf{MsgT});
    begin msg;
    inp net (nonce: \textbf{Un});
    out net \{msg, nonce\}_{key}

Receiver(net: \textbf{Un},
key: Key(MsgT, Un)) =
repeat
    new (nonce: \textbf{Un});
    out net nonce;
    inp net (ctext: \textbf{Un});
    decrypt ctext
        is \{msg: \textbf{MsgT}, nc: \textbf{Un}\}_{key};
    check nonce is nc;
    end msg
Typing the Sender and Receiver (again)

Sender(net:Un, key:Key(Msg\text{T},\text{Un})) = repeat 
  new (msg:Msg\text{T});
  begin msg;
  inp net (nonce:Un);
  out net {msg, nonce}\_key
  end msg

Receiver(net:Un, key:Key(Msg\text{T},\text{Un})) = repeat
  new (nonce:Un);
  out net nonce;
  inp net (ctext:Un);
  decrypt ctext
    is {msg:Msg\text{T}, nc:Un}\_key;
  check nonce is nc;
  end msg
Typing the Sender and Receiver (again)

Sender(net: Un, 
  key: Key(MsgT, Un)) =
  repeat
    new (msg: MsgT);
    begin msg;
    inp net (nonce: Un);
    cast nonce
      is (nc: Nonce [...])
    out net {msg, nc}key

Receiver(net: Un, 
  key: Key(MsgT, Un)) =
  repeat
    new (nonce: Un);
    out net nonce;
    inp net (ctext: Un);
    decrypt ctext
      is {msg: MsgT, nc: Un}key;
    check nonce is nc;
  end msg
Typing the Sender and Receiver (again)

Sender(net: \textbf{Un},
key: \textbf{Key}(\textit{MsgT}, \textit{Un})) =
repeat
new (msg: \textit{MsgT});
begin msg;
inp net (nonce: \textbf{Un});
cast nonce
is (nc: \text{Nonce} [...])
out net \{msg, nc\}_key

Receiver(net: \textbf{Un},
key: \textbf{Key}(\textit{MsgT}, \textit{Un})) =
repeat
new (nonce: \textbf{Un});
out net nonce;
inp net (ctext: \textbf{Un});
decrypt ctext
is \{msg: \textit{MsgT}, nc: \textbf{Un}\}_key;
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net:Un, key:Key(MsgT, Nonce [...]))) =
repeat
new (msg:MsgT);
begin msg;
inp net (nonce:Un);
cast nonce
is (nc: Nonce [...]])
out net {msg, nc}_key

Receiver(net:Un, key:Key(MsgT, Nonce [...]))) =
repeat
new (nonce:Un);
out net nonce;
inp net (ctext:Un);
decrypt ctext
is {msg:MsgT, nc:Un}_key;
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(\text{net}: \text{Un},
\text{key}: \text{Key(MsgT, Nonce [...])}) =
repeat
  new (msg: MsgT);
  begin msg;
  inp net (nonce: \text{Un});
  \text{cast nonce is (nc: Nonce [...])}
  out net \{msg, nc\}_{\text{key}} : \text{Un}
Receiver(\text{net}: \text{Un},
\text{key}: \text{Key(MsgT, Nonce [...])}) =
repeat
  new (nonce: \text{Un});
  out net nonce;
  inp net (ctext: \text{Un});
  decrypt ctext is \{msg: MsgT, nc: \text{Un}\}_{\text{key}};
  check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net:\textbf{Un},
key:\textbf{Key}(\textbf{MsgT},
             \textbf{Nonce} [...])) =
repeat
    new (msg:\textbf{MsgT});
begin msg;
    inp net (nonce:\textbf{Un});
    \textbf{cast nonce}
    \begin{itemize}
        \item is (nc: \textbf{Nonce} [...])
    \end{itemize}
out net \{msg, nc\}_{key}

Receiver(net:\textbf{Un},
key:\textbf{Key}(\textbf{MsgT},
             \textbf{Nonce} [...])) =
repeat
    new (nonce:\textbf{Un});
out net nonce;
inp net (ctext:\textbf{Un});
decrypt ctext
    \begin{itemize}
        \item is \{msg:\textbf{MsgT}, nc:\textbf{Un}\}_{key};
    \end{itemize}
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net: \texttt{Un},
key: \texttt{Key(MsgT, Nonce [...])}) =
repeat
new (msg: \texttt{MsgT});
begin msg;
inp net (nonce: \texttt{Un});
cast nonce
  is (nc: Nonce [...])
out net \{msg, \texttt{nc}\}_\texttt{key}

Receiver(net: \texttt{Un},
key: \texttt{Key(MsgT, Nonce [...])}) =
repeat
new (nonce: \texttt{Un});
out net nonce;
inp net (ctxt: \texttt{Un});
decrypt ctext
  is \{msg: \texttt{MsgT}, nc: \texttt{Un}\}_\texttt{key};
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net: Un, key: Key(MsgT, Nonce [...] )) =
repeat
  new (msg: MsgT);
  begin msg;
  inp net (nonce: Un);
  cast nonce is (nc: Nonce [...] )
  out net {msg, nc}key
Receiver(net: Un, key: Key(MsgT, Nonce [...] )) =
repeat
  new (nonce: Un);
  out net nonce;
  inp net (ctxt: Un);
  decrypt ctxt
    is {msg: MsgT, nc: Nonce [...] }key;
  check nonce is nc;
  end msg
Typing the Sender and Receiver (again)

Sender(net: Un, 
key: \textbf{Key}(\textit{MsgT},
\textit{Nonce} [...])) = 
repeat
new (msg: \textit{MsgT});
begin msg;
inp net (nonce: \textbf{Un});
\textbf{cast nonce}
\hspace{1em} is (nc: \textit{Nonce} [...])
out net \{msg, nc\}_\text{key}

Receiver(net: Un, 
key: \textbf{Key}(\textit{MsgT},
\textit{Nonce} [...])) = 
repeat
new (nonce: \textbf{Un});
out net nonce;
inp net (ctext: \textbf{Un});
decrypt ctext
\hspace{1em} is \{msg: \textit{MsgT},
\hspace{1em} nc: \textit{Nonce} [...]\}_\text{key};
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net: **Un**, key: \texttt{Key(MsgT, Nonce [...]}) = repeat
new (msg: **MsgT**);
begin msg;
inp net (nonce: **Un**);
cast nonce
\texttt{is (nc: Nonce [...]})
out net \{msg, nc\}_key

Receiver(net: **Un**, key: \texttt{Key(MsgT, Nonce [...]}) = repeat
new (nonce: **Un**);
out net nonce;
inp net (ctext: **Un**);
decrypt ctext
\texttt{is \{msg:MsgT, nc:Nonce [...]\}}_key;
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net: Un, 
key: Key(MsgT,  
Nonce [...] )) =

repeat
new (msg: MsgT);
begin msg;
inp net (nonce: Un);

**cast nonce**

is (nc: Nonce [...] )

out net {msg, nc}_{key}

Receiver(net: Un,  
key: Key(MsgT,  
Nonce [...] )) =

repeat
new (nonce: Un);
out net nonce;
inp net (ctext: Un);
decrypt ctext
is {msg: MsgT, 
nc: Nonce [...] }_{key};
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Sender(net: \textbf{Un},
key:\textbf{Key}(\textbf{MsgT},
\textbf{Nonce} [...] )) =
repeat
new (msg: \textbf{MsgT});
begin msg;
inp net (nonce: \textbf{Un});
cast nonce is (nc: \textbf{Nonce} [...] )
out net \{msg, nc\}_key

Receiver(net: \textbf{Un},
key:\textbf{Key}(\textbf{MsgT},
\textbf{Nonce} [...] )) =
repeat
new (nonce: \textbf{Un});
out net nonce;
inp net (ctext: \textbf{Un});
decrypt ctext is \{msg: \textbf{MsgT},
\textbf{Nonce} [...]\}_key;
check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Receiver(net: Un,
key: Key(MsgT, 
    Nonce [...] )) =
repeat
    new (nonce: Un);
    out net nonce;
    inp net (ctext: Un);
    decrypt ctext
    is {msg: MsgT, 
        nc: Nonce [...] }key;
    check nonce is nc;
end msg
Typing the Sender and Receiver (again)

Receiver(net: \texttt{Un},
key: \texttt{Key(MsgT, Nonce [...])}) =

repeat
new (nonce: \texttt{Un});
out net nonce;
inp net (ctext: \texttt{Un});
decrypt ctext
is \{msg: \texttt{MsgT},
nc: \texttt{Nonce [...]}\}_{\texttt{key}},
check nonce is nc;
end msg
\textbf{: es + [check nonce]}

Typing the Sender and Receiver (again)

Receiver(net: \textbf{Un},
key: \textbf{Key}(\textbf{MsgT},
\textbf{Nonce} [...]))) =

repeat

\textbf{new} (nonce: \textbf{Un}); \quad : \text{es'} - [\text{check nonce}]

out net nonce;
inp net (ctext: \textbf{Un});
decrypt ctext
\quad is \{msg: \textbf{MsgT},
\quad \quad nc: \textbf{Nonce} [...]\}_{key};
\quad \textbf{check nonce} \text{ is nc}; \quad : \text{es} + [\text{check nonce}]
end msg
Typing the Sender and Receiver (again)

Receiver(net: \texttt{Un},
key: \texttt{Key(\texttt{MsgT},
    \texttt{Nonce [...])}}) =

repeat
  \texttt{new (nonce:Un);}
  \texttt{out net nonce;}
  \texttt{inp net (ctext:Un);}
  \texttt{decrypt ctext}
  \texttt{is \{msg:\texttt{MsgT},}
      \texttt{nc:\texttt{Nonce [...]}\}_{key};}
  \texttt{check nonce is nc;}
  \texttt{es’ - [check nonce]}
end msg

\texttt{es + [check nonce] – [...]
Typing the Sender and Receiver (again)

Receiver \( (\text{net: } \text{Un}, \text{key: } \text{Key}(\text{MsgT, Nonce } [...] )) = \)

\[
\text{repeat}
    \text{new (nonce: } \text{Un}; \text{ out net nonce;}
    \text{ inp net (ctext: } \text{Un);}
    \text{ decrypt ctext}
    \text{ is } \{\text{msg: } \text{MsgT, nc: } \text{Nonce } [...] \}_\text{key};
    \text{ check nonce is nc;}
\]

: \text{es' - [check nonce]}

\[
\text{nc: Nonce } [...]
\]

: \text{es + [check nonce] - [...]}

\text{end msg}
Typing the Sender and Receiver (again)

Receiver(net: Un,
key: Key(MsgT,
    Nonce [msg])) =
repeat
    new (nonce: Un);
end msg

: es' - [check nonce]

nc: Nonce [end msg]

: es + [check nonce] - [end msg]
Typing the Sender and Receiver (again)

Receiver(net: Un, 
key: Key(MsgT, 
   Nonce [msg])) = 
repeat
   new (nonce: Un);
   out net nonce;
   inp net (ctext: Un);
   decrypt ctext
   is {msg: MsgT, 
      nc: Nonce [end msg]}key;
   check nonce is nc;
end msg : [end msg]
Typing the Sender and Receiver (again)

Receiver(net: $\textbf{Un}$,
key: \textbf{Key}(MsgT,
\textbf{Nonce} [msg]) =

repeat
  new (nonce: $\textbf{Un}$);
  out net nonce;
  inp net (ctext: $\textbf{Un}$);
  decrypt ctext
  is \{msg: $\textbf{MsgT}$,
  nc: $\textbf{Nonce}$ [end msg]\}$_{\text{key}}$;
  check nonce is nc;
  end msg

: [end msg] + [check nonce]
  \[ = [end msg] = [check nonce] \]
Typing the Sender and Receiver (again)

Receiver(net: \texttt{Un},
key: \texttt{Key}(\texttt{MsgT},
\texttt{Nonce \{msg\}})) = 
repeat 
new (nonce: \texttt{Un});
out net nonce;
inp net (ctext: \texttt{Un});
decrypt ctext 
is \{msg: \texttt{MsgT},
nc: \texttt{Nonce \{end msg\}}\_{key};
check nonce is nc;
end msg

: [check nonce]
Typing the Sender and Receiver (again)

Receiver(net: \textbf{Un},
     key: \textbf{Key}(\textbf{MsgT,}
                  \textbf{Nonce }[\textbf{msg}])) =

repeat
\textbf{new } (nonce: \textbf{Un});
out net nonce;
inp net (ctext: \textbf{Un});
decrypt ctext
\textbf{is } \{msg: \textbf{MsgT},
          nc: \textbf{Nonce }[\textbf{end msg}]}_{\textbf{key}};
check nonce is nc;
end msg

: [check nonce]
– [check nonce] = [ ]
Typing the Sender and Receiver (again)

Receiver(net: Un, 
  key: Key(MsgT, 
           Nonce [msg]))) = 

  repeat
    new (nonce: Un);
    out net nonce;
    inp net (ctext: Un);
    decrypt ctext
    is {msg: MsgT,
       nc: Nonce [end msg]}key;
    check nonce is nc;
  end msg
Typing the Sender and Receiver (again)

Sender(net: \texttt{Un},
key: \texttt{Key}(\texttt{MsgT},
\texttt{Nonce} [...]))) =
repeat
new (msg: \texttt{MsgT});
begin msg;
inp net (nonce: \texttt{Un});
cast nonce
is (nc: \texttt{Nonce} [...])
out net \{msg, nc\}_key
Typing the Sender and Receiver (again)

Sender(net: Un, 
    key: Key(MsgT, 
                Nonce [end msg]))) =

repeat
  new (msg: MsgT);
  begin msg;
  inp net (nonce: Un);
  cast nonce
  is (nc: Nonce [end msg])
  out net {msg, nc}_key
Typing the Sender and Receiver (again)

Sender(net: Un, 
key: Key(MsgT, 
    Nonce [end msg])) =
repeat
    new (msg: MsgT);
    begin msg;
    inp net (nonce: Un);
    cast nonce 
    is (nc: Nonce [end msg])
    : es + [end msg]
    out net {msg, nc}key
Typing the Sender and Receiver (again)

Sender(net:Un, key:Key(MsgT, Nonce [end msg])) =
repeat
  new (msg:MsgT);
  begin msg;
  inp net (nonce:Un);
  cast nonce is (nc: Nonce [end msg])
  out net {msg, nc}_{key} : es’ – [end msg]
  : es + [end msg]

Typing the Sender and Receiver (again)

Sender(net: Un, key: Key(MsgT, Nonce [end msg])) =
repeat
  new (msg: MsgT);
  begin msg;
  inp net (nonce: Un);
  cast nonce
  is (nc: Nonce [end msg])
  out net {msg, nc}_{key} : [ ]
Typing the Sender and Receiver (again)

Sender(net: \textbf{Un},  
key: \textbf{Key}({\textbf{MsgT},  
\textbf{Nonce} [end msg]})) =

repeat

new (msg: \textbf{MsgT}); 
begin msg; 
inp net (nonce: \textbf{Un}); 
\textbf{cast nonce} is (nc: \textbf{Nonce} [end msg])
out net \{msg, nc\}_{\text{key}} 
\textbf{: [ ] + [end msg] = [end msg]}
Typing the Sender and Receiver
(again)

\[
\text{Sender}(\text{net:} \textbf{Un}, \text{key:} \textbf{Key}((\text{MsgT}, \text{Nonce} [\text{end msg}]))) = \\
\text{repeat} \\
\text{new (msg:} \textbf{MsgT}); \\
\text{begin msg;} \\
\text{inp net (nonce:} \textbf{Un}); \\
\text{cast nonce} \\
\text{is (nc: Nonce [end msg])} \\
\text{out net \{msg, nc\}_key} : [\text{end msg}]
\]
Typing the Sender and Receiver (again)

Sender(net: \textbf{Un}, 
\text{key}: \textbf{Key} (\textbf{MsgT,} 
\textbf{Nonce [end msg]}))) =

repeat

new (msg: \textbf{MsgT});

\text{begin msg;}
\text{inp} \text{ net (nonce:Un);} 
\textbf{cast nonce}
\text{ is (nc: Nonce [end msg])}
\text{out net \{msg, nc\}_key}

\textbf{: [end msg]}
\textbf{– [end msg]} = [ ]
Typing the Sender and Receiver (again)

Sender(net: \textbf{Un},
\textbf{key}: Key(MsgT,\n\textbf{Nonce [end msg]}))) =
\begin{align*}
\text{repeat} & : [ ] \\
\text{new (msg: MsgT);} & \\
\text{begin msg;} & \\
\text{inp net (nonce: Un);} & \\
\textbf{cast nonce} & \\
\text{is (nc: Nonce [end msg])} & \\
\text{out net \{msg, nc\}_{key}} &
\end{align*}
Typing the System (again)

System(net: Un) = new (key: Key(MsgT, Un));
  (Sender(net, key) | Receiver(net, key))
Typing the System (again)

System(net: Un) = new (key: Key(MsgT, Un));
(Sender(net, key) | Receiver(net, key))
: [] : []
Typing the System (again)

System(net: Un) = new (key: Key(MsgT, Un));
(Sender(net, key) | Receiver(net, key))

: [ ]

And thus the protocol is proven to be robustly safe!
Conclusions

• This paper presents a system for **automatically checking correspondence assertions**, using a novel type and effect system
• Doesn’t set a bound on the **size of the opponents**
• **Conservative system** (false negatives)
• Requires **some human intervention** to achieve decidability
• Restricted to only **one pattern** of nonce handshakes, but probably can be extended to others in a straightforward way