Authenticity by Typing for Security Protocols

Andrew D. Gordon & Allan Jeffrey

presented by Vassilis Koutavas

Authenticity

Msg. 1 A → B : $\{M\}_K$

Authenticity

Msg. 1 A → B : $\{M\}_K$

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

Authenticity

Msg. 1 A → 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 → 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

Msg. 1 A → B : $\{M\}_K$

$\downarrow\uparrow$

C

Authenticity

Msg. 1 A → B : $\{M\}_K$

↓↑

C

Msg. 2 C → B: $\{M\}_K$

Msg. 3 C → B: $\{M\}_K$

…

Authenticity

Msg. 1 A → B : $\{M\}_K$

↓↑

C

Msg. 2 C → B: $\{M\}_K$

Msg. 3 C → B: $\{M\}_K$

...

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
- x It considers only **opponents that can be expressed in Spi-calculus**
- x 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 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 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 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 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 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 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 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 L; P | end L; P

| out M N | inp M (x:**T**); P

| new (x:**T**); P | decrypt M is $\{x:\mathbf{T}\}_K$; P

| check M is N; P

| repeat P | (P | P) | ...

L, M, N, K ::= x | $\{M\}_K$ | ...

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 → 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 → 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) | Op(net))
```

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)**
- ...

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 $\Rightarrow$ out M N is well-typed (WT)

M: Ch(T), P is WT $\Rightarrow$ inp M (x:T); P is WT

M: T, N: Key(T) $\Rightarrow$ $\{M\}_N$: Un

M: Un, N: Key(T), P is WT $\Rightarrow$ decrypt M is $\{x{:}T\}_N$; P is WT

Type and Effect System

The effects (the types of processes):

- Atomic end-effect: **end L**

  P: [end L_1, end L_2, ..., end L_n]

- ...

Type and Effect System

The effects (the types of processes):

- Atomic end-effect: **end L**

$$P: [\text{end } L_1, \text{end } L_2, \ldots, \text{end } L_n]$$

- ...

$P: es \;\Rightarrow\; \text{end } L; P : (es + [\text{end } L])$

$P: es \;\Rightarrow\; \text{begin } L; P : (es - [\text{end } L])$

$P: es_P, Q: es_Q \;\Rightarrow\; P|Q : (es_P + es_Q)$

Type and Effect System

The effects (the types of processes):

- Atomic end-effect: **end L**

$$P: [\text{end } L_1, \text{end } L_2, \ldots, \text{end } L_n]$$

- ...

$P\text{: es} \;\Rightarrow\; \text{end L; P : (es + [end L])}$

$P\text{: es} \;\Rightarrow\; \text{begin L; P : (es – [end L])}$

$P\text{: } es_P\text{, Q: } es_Q \;\Rightarrow\; P|Q : (es_P + es_Q)$

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

Typing the Sender

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

Typing the Sender

Sender(net:**Un**, key:**Key(T)**) =

  repeat

    new (msg:**MsgT**);

    begin msg;

    inp net (nonce:**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: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: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:**Un**, key:**Key(MsgT,Un)**) =

repeat

new (msg:**MsgT**);

begin msg; **: [ ] – [end msg] = [ ]**

inp net (nonce:**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 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:Un, key:Key(MsgT,Un)) =
  repeat
    new (nonce:Un);
    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:Un);
    out net nonce;
    inp net (ctext:Un);
    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: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 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 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;      : [end msg]
    check nonce is nc;
    end msg
```

Typing the Receiver

```
Receiver(net:Un, key:Key(MsgT,Un)) =
  repeat                                                : [end msg]
    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 System

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

Typing the System

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

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

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**

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}$_{key}$

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

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

causality

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: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: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;   : es + [check nonce]
      end msg
```

Typing the Sender and Receiver (again)

```
Receiver(net:Un,
  key:Key(MsgT,
          Nonce [...])) =
  repeat
    new (nonce:Un);          : es' - [check nonce]
    out net nonce;
    inp net (ctext:Un);
    decrypt ctext
      is {msg:MsgT,
          nc:Nonce [...]}key;
    check nonce is nc;       : es + [check nonce]
    end msg
```

Typing the Sender and Receiver (again)

```
Receiver(net:Un,
  key:Key(MsgT,
            Nonce [...])) =
  repeat
    new (nonce:Un);          : es' - [check nonce]
    out net nonce;
    inp net (ctext:Un);
    decrypt ctext
      is {msg:MsgT,
          nc:Nonce [...]}key;
    check nonce is nc;       : es + [check nonce] – [...]
    end msg
```

Typing the Sender and Receiver (again)

```
Receiver(net:Un,
  key:Key(MsgT,
              Nonce [...])) =
  repeat
    new (nonce:Un);          : es' - [check nonce]
    out net nonce;
    inp net (ctext:Un);
    decrypt ctext
      is {msg:MsgT,
          nc:Nonce [...]}key;
    check nonce is nc;       : es + [check nonce] – [...]
    end msg
```

nc: Nonce [...]

Typing the Sender and Receiver (again)

Receiver(net:**Un**,
key:**Key(MsgT,**
Nonce [msg])) =
repeat
new (nonce:**Un**); **: es' - [check nonce]**
out net nonce;
inp net (ctext:**Un**);
decrypt ctext
is {msg:**MsgT**,
nc:**Nonce [end msg]**}$_{key}$;
check nonce is nc; **: es + [check nonce] – [end msg]**
end msg

nc: 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: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] + [check nonce]
– [end msg] = [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
```

: [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
```

: [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: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
```

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                          : es + [end msg]
      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;                      : es' –  [end 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  : [ ]
```

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
```

: [ ] + [end msg]
= [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);         : [end msg]
    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])
    out net {msg, nc}key
```

: [end msg]
– [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 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 straight-forward way