Authenticity by Typing for Security Protocols

Andrew D. Gordon
Microsoft Research

Alan Jeffrey
DePaul University

*Presented for CMSC838z by*
*Saurabh Srivastava*
The way we (used to?) make protocols:

- Specify the toplevel functional requirements:
  - data transfer --> ftp
  - distributed data transfer --> bittorrent (!!)
  - medium access --> 802.x
  - authentication --> Needham-Schroeder, Otway-Ree’s etc.

- Specifications drawn up directly from functional requirements.
  - State machine diagrams as a side note.

- Proof of correctness was ad hoc
  - Errors!

- Security protocols are especially tricky.
Needham Schroeder authentication

- Celebrated! Or at least best known!
- Possibility of a replay attack (on Kab being compromised).
- B has no way of verifying that m3 is in fact new.
Amended Needham Schroeder authentication

m2 = \{A, Nb0\} Kbs

m3 = \{Kab, Nb0, A\} Kbs
Needham Schroeder ‘type-checked’:

$ java -jar ../cryptyc.jar ns.cry
Error, rooted at line 134:
In end-statement: Can't guarantee that a correspondence has begun.
Required correspondence:
  begun(providing session key kab to b for a)
Available correspondences:
  None

Modified Needham Schroeder:

[poseidon]$ java -jar ../cryptyc.jar ns-modified.cry
Type checked OK!
The overall view

Spi-calculus representation → Top level specifications → FSM? → Working code

Spi-calculus enriched with correspondence assertions → Type annotations for keys, nonces and msgs

Old

New!
Spi Calculus - by example

Names and Msgs:
- name: x
- pair: (M,N)
- empty: ()
- tagged union

Processes:
- out M N
- inp M (x:T);P
- P | Q
- others…

Public channel = net

new $k$; (A|B)

repeat

new $M$;
out net $\{M\}k$;

repeat

inp net $ctext$;
decrypt $ctext$ is $\{M\}k$;
Outline

Here

Spi-calculus representation

Top level specifications

Spi-calculus enriched with correspondence assertions

Type annotations for keys, nonces and msgs

Type checker

Ok?
Correspondence Assertions

- Augment the calculus:
  - with new events:
    - begin L
    - end L
  - with new processes:
    - begin L; P
    - end L; P

- Allows the correlation of events across processes

- Prevents *man-in-middle* and *replay* attacks

Eg:
- begin L; begin L; end L; end L
- begin L; end L
- begin L; begin L’; end L; end L’
  - end L
  - begin L; end L; end L

- (begin L | end L)
Safety and Robust safety

- **Safety:**
  - \( P \) is safe \( \iff \) each end \( L \) has a corresponding begin \( L \)

- **Opponents:** Arbitrary processes.
  - Untyped
  - Cannot assert events

- **Robust safety:**
  - \( P \) is safe \( \iff (P|O) \) is safe for any arbitrary \( O \).

*Note:*
- Might seen vacuous but in fact is not
- Consider the opponent who does a replay attack:
  \begin{verbatim}
  inp net msg; out net msg; out net msg;
  \end{verbatim}
Example

- Allows us to correlate across processes
- Opponent which breaks this:
  - inp net ctext; out net ctext; out net ctext
- Therefore safe but not robustly safe.

Solution: Add nonce!

Event 1: A begins M
Message 1: B->A: n
Message 2: A->B: {M,n}k
Event 2: B ends M

Now robustly safe!

But need to check that:
  - Type system!!!
Outline - status check

Spi-calculus representation

Top level specifications

Spi-calculus enriched with correspondence assertions

Type annotations for keys, nonces and msgs

Here

Type checker

Ok?
Typing protocols

- **Data types**
  - **Un**: The untrusted type.
    - Public data and channels: anything exposed to the adversary
  - Trusted types:
    - Ch(T): channel, Key(T): key for encrypting type T etc

- **Effects for processes**
  - Keep track of unmatched end assertions.
  - Something like: P: [end L, end L, end M, end N…]
  - Tracking sequential code is easy.
  - What about transferring effects over parallel processes?
    - The yellow arrow!
    - Some kind of temporal precedence needs to be established.
Nonce handshakes establish ordering!

- Paper’s most relevant contribution (in my opinion)
- The *only* way to create a nonce is to *cast* it.
  - Cast $N$ is $(x:\text{Nonce effects});P$ \hspace{1cm} \text{------- (cast is a new Process!!)}
- A check primitive
  - Exists
  - If typechecked - proves that a preceeding process called a cast
  - Processes now have effects of this form:
    - $P: [\text{end L, end M, … check } N, \text{ check } Q]$  
    - Allow the type checker to ensure that at `new’ nonce generation time , ie $(\text{new } N: \text{Un}; P)$, check $N$ occurs at most once in the effect of $P$
An example

- Scenario:
  - $n$ is a nonce that is somehow already shared between the processes. $n:Un$
  - $c$ is a private channel (so can have type other than Ch(Un)) - so that we do not have to bother about encryption and decryption of the nonce
  - Specifically it has type Ch(Nonce [end m])

- $P =$
  - begin m;
  - cast n is (n':Nonce [end m]);
  - out c n'

- $Q =$
  - inp c (x:Nonce [end m]);
  - check n is x;
  - end m

- $R =$ new (n:Un); (P|Q) typechecks?
An example

- **P =**
  - begin m;
  - cast n is (n’:Nonce [end m]);
  - out c n’

- **Q =**
  - inp c (x:Nonce [end m]);
  - check n is x;
  - end m

- **R =** new (n:Un); (P|Q)

**Rules (informally):**

- If **N:Un** and **P:ef**
  - cast N is (x:Nonce es); P: (es+ef)

- If **N:Un** and and **N’:Nonce es** and **Q:ef**
  - check N is N’; Q: ((ef - es) + [check N])

- If **P:ef**
  - new (N); P: (ef-[check N])

Red: Processes
Brown: types
Blue: Effects - end/check
Green: Data
Post mortem analysis!

- Notice what happened.
- Cast:
  - The cast ‘encapsulated’ all the begins into the effect that nonce ‘carried’
  - Since a cast is the only way that a nonce type can be created => nonces will always carry effects to other processes
- Check:
  - A check causes the reverse at the receiver process
  - In addition it adds to the effect multiset an effect which ensures that one and only one new matches the check. (single use of nonces)

Conceptually:

begin a;
begin b;
....
cast ___ es

check ___ es

....
end a;
end b;
Outline - status check

- Spi-calculus representation
- Spi-calculus enriched with correspondence assertions
- Top level specifications
- Type checker
- Type annotations for keys, nonces and msgs
- Ok?

Here
A few comments about the type checking

- Code already annotated with types
- Deterministic choice of judgement rules for types (msgs and names) \textit{and} effects (processes)
- For Un: everything type checks
- Interesting mention:
  - Msg Encrypt
  - Proc Decrypt
Summary

- Provides a way of validating the ‘model’ that will be implemented.
- Uses correspondence assertions to verify `control flow’ (Lam and Woo)
- Major contribution is the successful typing of nonce handshakes. Nonce handshakes allow processes to sync.
- Forms the basis of verifying correctness of public key based protocols
Criticism

- Model does not handle timestamp based protocols. Although it does not seem hard to incorporate.
- It is programming of the protocol! And that too with hard restrictions on the types.
  - Prone to error? - but probably less than manual spec.