

# A Security Analysis of a Cryptographic Handshake Protocol

Yi Ting Lo  
Advisor: Liang Ting Chen

August 27, 2025

## 1 Motivation / Introduction

Cryptographic protocols that lack formal correctness verification are prone to unforeseen attacks. For example, the key exchange protocol proposed by Needham and Schroeder in 1978 was later shown—17 years after its publication—to be vulnerable to a man-in-the-middle attack. This underscores the necessity of rigorous formal verification to ensure protocol security.

In this work, I formalize a handshake protocol to analyze its security properties—with a focus on confidentiality. Then, I introduce **ProVerif** to verify these properties, and finally provide a security proof.

## 2 Handshake protocol and potential issues

Consider a scenario where a client needs to transmit a secret  $s$  to the server. Since messages sent by the client may be exposed to eavesdroppers and there is a risk of a malicious attacker impersonating the server, additional safeguards are necessary. To address these concerns, the client and server follow the protocol illustrated in Figure 1 to ensure both the confidentiality and integrity of the transmitted message.

### Protocol description:

1. Server and client both generate a key pair  $(pk_A, sk_A)$  and  $(pk_B, sk_B)$ .
2. Client  $A$  sends its public key  $pk_A$  to Server  $B$ .
3. Server  $B$  generates a session key  $k$ , signs  $(pk_B, k)$ , encrypts it with  $pk_A$ , and sends it to  $A$ .
4. Client  $A$  decrypts, verifies the signature, and uses  $k$  to send  $Enc_k(s)$  securely to  $B$ .

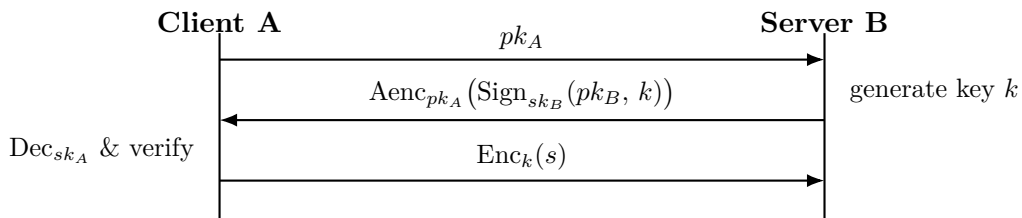


Figure 1: Handshake protocol sequence diagram

An eavesdropper observing only the transmitted messages seems unable to learn  $s$ , and the presence of signatures appears to prevent an adversary from impersonating the server. However, the protocol is still insecure because it remains vulnerable to a man-in-the-middle (MitM) attack, where an attacker intercepts and alters the exchanged messages to establish separate sessions with the client and server.

### 3 Pi-Calculus, Spi-Calculus

The *pi-calculus* is a process algebra designed to describe concurrent systems with dynamic communication structures. Its core operations—name passing, input/output on channels, replication, and name restriction—enable precise modeling of mobile processes and message exchange.

The *spi-calculus* extends the pi-calculus with cryptographic primitives such as encryption, decryption, pairing, and nonce generation. This extension facilitates formal reasoning about security properties, including secrecy and authentication, within cryptographic protocols. (See appendix for full syntax)

To make the concept concrete, here's the handshake protocol written in spi-calculus:

Server B (spi-calculus)	Client A (spi-calculus)
$  \begin{aligned}  B &\triangleq c(x_{pkX}). \\  &(\nu k) \\  &\bar{c}\langle aenc(sign((pk_B, k), sk_B), x_{pkX}) \rangle. \\  &c(x). \\  &\text{let } z = sdec(x, k) \text{ in } 0  \end{aligned}  $	$  \begin{aligned}  A &\triangleq \bar{c}\langle pk_A \rangle. \\  &c(x). \\  &\text{let } y = adec(x, sk_A) \text{ in} \\  &\text{let } (pk_{B'}, k) = \text{checksign}(y, pk_B) \text{ in} \\  &\text{if } pk_{B'} = pk_B \text{ then} \\  &\quad \bar{c}\langle senc(s, k) \rangle. 0 \\  &\text{else } 0  \end{aligned}  $

### 4 ProVerif and Protocol Security properties

First, we define  $\text{attacker}(\cdot)$  such as attacker knows  $s$  if and only if  $\text{attacker}(s)$  is satisfied:

**Formal Definition of  $\text{attacker}(\cdot)$ :** A trace

$$T = (E_0, \mathcal{P}_0) \longrightarrow^* (E', \mathcal{P}')$$

satisfies  $\text{attacker}(M)$  if and only if  $T$  contains a reduction of the form

$$(E, \mathcal{P} \cup \{\bar{c}\langle M \rangle.Q, c(x).P\}) \longrightarrow (E, \mathcal{P} \cup \{Q, P\{M/x\}\})$$

for some  $E, \mathcal{P}, x, P, Q$  and  $c \in \text{Init}$

To declare a protocol insecure, one must find an attacker trace that satisfies the above definition a task that can be complex to perform manually. **ProVerif** is an automated security verification tool to automates this process: it determines whether  $\text{attacker}(s)$  holds and, if so, outputs an attacker trace. Using this trace and the reduction rules, we can formally confirm that the protocol is insecure (see Appendix for details). For the handshake protocol discussed earlier, the query  $\text{attacker}(s)$  does indeed succeed, proving the protocol insecure.

After analyzing the attacker trace, I discovered that the handshake protocol contains a flaw because Server B does not include the public key of the message originator in the transmitted message. This omission allows a Man-in-the-Middle (MitM) attack. By incorporating the originator's public key into the transmitted tuple, the query  $\text{attacker}(s)$  no longer succeeds. Since **ProVerif** guarantees soundness, we can conclude that the modified protocol is secure.

### 5 Future Work

- Use proof assistant (such as Agda) to prove the correctness.
- Understand why the soundness is ensure.

# A Syntax of Pi-Calculus and Spi-Calculus

## Pi-Calculus Syntax

The *pi-calculus* defines processes that interact through named channels. Its syntax can be expressed as follows:

$$\begin{aligned} P, Q ::= & 0 \quad (\text{nil process}) \\ & | \bar{a}\langle b \rangle.P \quad (\text{output of name } b \text{ on channel } a) \\ & | a(x).P \quad (\text{input of } x \text{ on channel } a) \\ & | P \mid Q \quad (\text{parallel composition}) \\ & | (\nu a)P \quad (\text{restriction, generate fresh name } a) \\ & | !P \quad (\text{replication, infinite copies of } P) \end{aligned}$$

## Spi-Calculus Syntax

The *spi-calculus* extends the pi-calculus by adding cryptographic constructs for modeling security protocols:

$$M, N ::= x \mid n \mid (M_1, M_2) \mid \{M\}_K \quad (\text{terms: names, pairs, encryption})$$

$$\begin{aligned} P, Q ::= & 0 \quad (\text{nil}) \\ & | \bar{a}\langle M \rangle.P \quad (\text{send term } M \text{ on } a) \\ & | a(x).P \quad (\text{receive term into } x) \\ & | P \mid Q \quad (\text{parallel}) \\ & | (\nu n)P \quad (\text{new name or nonce}) \\ & | !P \quad (\text{replication}) \\ & | \text{case } M \text{ of } \{x\}_K \text{ in } P \text{ else } Q \quad (\text{decryption}) \\ & | \text{case } M \text{ of } (x, y) \text{ in } P \text{ else } Q \quad (\text{pattern match on pairs}) \end{aligned}$$

Here:

- $a, b$  are channel names,  $x, y$  are variables, and  $n$  is a nonce or key.
- $\{M\}_K$  denotes encryption of message  $M$  under key  $K$ .
- $(M_1, M_2)$  denotes a pair of messages.

These constructs allow modeling of secrecy and authentication properties in cryptographic protocols.

## B Full handshake protocol

```
1 free c:channel.
2
3 free s:bitstring [private].
4 query attacker(s).
5
6 type key.
7
8 fun senc(bitstring, key): bitstring.
9
10 reduc forall m: bitstring, k: key; sdec(senc(m,k), k) = m.
```

```

11
12 type skey.
13 type pkey.
14
15 fun pk(skey): pkey.
16 fun aenc(bitstring, pkey): bitstring.
17
18 reduc forall m: bitstring, k: skey; adec(aenc(m, pk(k)), k) = m.
19
20 type sskey.
21 type spkey.
22
23 fun spk(sskey): spkey.
24 fun sign(bitstring, sskey): bitstring.
25
26 reduc forall m: bitstring, k: sskey; getmess(sign(m,k)) = m.
27 reduc forall m: bitstring, k: sskey; checksign(sign(m,k),spk(k)) = m.
28
29 let clientA(pkA:pkey ,skA:skey ,pkB:spkey) =
30   out(c, pkA);
31   in(c,x:bitstring);
32   let y = adec(x,skA) in
33   let (=pkB,k:key) = checksign(y,pkB) in
34   out(c,senc(s,k)).
35
36 let serverB(pkB:spkey ,skB:sskey) =
37   in(c,pkX:pkey);
38   new k:key;
39   out(c,aenc(sign((pkB,k),skB),pkX));
40   in(c,x:bitstring);
41   let z = sdec(x,k) in
42   0.
43
44 process
45   new skA:skey;
46   new skB:sskey;
47   let pkA = pk(skA) in out(c,pkA);
48   let pkB = spk(skB) in out(c,pkB);
49   ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB)) )

```

Listing 1: Handshake protocol in ProVerif syntax

## C Attacker trace derivation

Figure 2 presents the reduction rules for the operational semantics of the calculus. The rules describe how a system configuration, represented as  $(E, \mathcal{P})$ , evolves according to the process structure and the current frame  $E$ .

### Processes

#### Client $A$ :

$$A \equiv \bar{c}\langle \text{pk}(sk_A) \rangle. c(x). \text{let } y = \text{adec}(x, sk_A) \text{ in} \\ \text{let } (= \text{spk}(sk_B), k) = \text{checksign}(y, \text{spk}(sk_B)) \text{ in } \bar{c}\langle \text{senc}(s, k) \rangle. 0$$

#### Server $B$ :

$$B \equiv c(pk_X). (\nu k) \bar{c}\langle \text{aenc}(\text{sign}((\text{spk}(sk_B), k), sk_B), pk_X) \rangle. \\ c(z). \text{let } \_ = \text{sdec}(z, k) \text{ in } 0.$$

$E, \mathcal{P} \cup \{0\} \longrightarrow E, \mathcal{P}$	(Red Nil)
$E, \mathcal{P} \cup \{!P\} \longrightarrow E, \mathcal{P} \cup \{P, !P\}$	(Red Repl)
$E, \mathcal{P} \cup \{(\nu a)P\} \longrightarrow E \cup \{a'\}, \mathcal{P} \cup \{P\{a'/a\}\}$	(Red Res) where $a' \notin E$ .
$E, \mathcal{P} \cup \{\bar{N}\langle M \rangle.Q, N(x).P\} \longrightarrow E, \mathcal{P} \cup \{Q, P\{M/x\}\}$	(Red I/O)
$E, \mathcal{P} \cup \{\text{let } x = g(M_1, \dots, M_n) \text{ in } P \text{ else } Q\} \longrightarrow E, \mathcal{P} \cup \{P\{M'/x\}\}$	(Red Destr 1) if $g(M_1, \dots, M_n) \longrightarrow M'$ .
$E, \mathcal{P} \cup \{\text{if } M = M \text{ then } P \text{ else } Q\} \longrightarrow E, \mathcal{P} \cup \{P\}$	(Red Cond 1)
$E, \mathcal{P} \cup \{\text{event}(M).P\} \longrightarrow E, \mathcal{P} \cup \{P\}$	(Red Event)

Figure 2: Operational semantics

**Attacker  $E$ :**

$$\begin{aligned}
E &\equiv c(p_A).c(p_B).\bar{c}\langle \text{pk}(sk_E) \rangle.c(m_1). \\
&\quad \text{let } y = \text{adec}(m_1, sk_E) \text{ in let } (\_, k) = \text{checksign}(y, p_B) \text{ in} \\
&\quad \bar{c}\langle \text{aenc}(y, p_A) \rangle.c(m_2).\text{let } s' = \text{sdec}(m_2, k) \text{ in } 0.
\end{aligned}$$

**Initial System:**

$$\text{Sys}_0 \equiv (\nu sk_A)(\nu sk_B)(\bar{c}\langle \text{pk}(sk_A) \rangle.0 \mid \bar{c}\langle \text{spk}(sk_B) \rangle.0 \mid !A \mid !B \mid E)$$

### Operational Semantics Derivation

1. **Expand Replication (Red Repl)**: add one instance of  $A$  and one instance of  $B$ .
2. **Public Key Output (Red I/O)**: client sends  $\text{pk}(sk_A)$ , server sends  $\text{spk}(sk_B)$ ; attacker records both.
3. **Attacker to Server (Red I/O)**:  $E$  sends  $\text{pk}(sk_E)$  to  $B$ .
4. **Fresh Key (Red Res)**: server generates fresh  $k$ .
5. **Server Reply (Red I/O)**: server outputs  $\text{aenc}(\text{sign}((\text{spk}(sk_B), k), sk_B), \text{pk}(sk_E)))$ ; attacker receives it.
6. **Decrypt and Verify (Red Destr 1)**: attacker computes  $y := \text{sign}((\text{spk}(sk_B), k), sk_B)$  and extracts  $k$  via RED COND 1.
7. **Forward to Client (Red I/O)**: attacker sends  $\text{aenc}(y, \text{pk}(sk_A))$  to  $A$ .
8. **Client Decrypt (Red Destr 1)**:  $A$  decrypts and verifies signature; binds  $k$ .
9. **Client Sends Secret (Red I/O)**:  $A$  outputs  $\text{senc}(s, k)$ ; attacker intercepts.
10. **Attacker Learns  $s$  (Red Destr 1)**: compute  $\text{sdec}(\text{senc}(s, k), k) \rightarrow s$ .

## References

- [1] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. *ProVerif 2.05: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. INRIA and CNRS, École Normale Supérieure, University of Birmingham, 2023. Available at: <https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf>.
- [2] Martín Abadi and Andrew D. Gordon, “A Calculus for Cryptographic Protocols: The Spi Calculus,” *Information and Computation*, vol. 148, no. 1, pp. 1–70, 1999. Available: <https://doi.org/10.1006/inco.1998.2740>.
- [3] Bruno Blanchet, “Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif,” in *Foundations of Security Analysis and Design VII (FOSAD 2012/2014)*, Lecture Notes in Computer Science, vol. 8604, pp. 1–38, Springer, 2014. Available: [https://doi.org/10.1007/978-3-319-10082-1\\_1](https://doi.org/10.1007/978-3-319-10082-1_1).