Computationally Sound Mechanized Proofs of Correspondence Assertions

Bruno Blanchet
CNRS, Ecole Normale Supérieure
blanchet@di.ens.fr

July 2007

Introduction

Our goal: implement an automatic, computationally sound prover for security protocols.

We have already implemented a prover for secrecy properties.

In this talk, we show how to extend it to correspondence assertions, that is, properties of the style:

> *If some event has been executed, then some other events have been executed.*

Basic application: authentication.

Proofs by sequences of games

The prover produces the proof is a sequence of games, as in Shoup's or Bellare and Rogaway's method:

- In the first game, the adversary plays against the real protocol.
- The prover transforms each game into the next by syntactic transformations or by applying security assumptions on cryptographic primitives.

  Consecutive games are computationally indistinguishable.
- The desired security property can be proved directly on the last game.

Games are formalized in a process calculus.

Differences with secrecy

For the proof of correspondences:

- The language for games in the same as for secrecy, except for the addition of events.
- The game transformations and the proof strategy are the same as for secrecy. (The events are left unchanged.)
- One needs a new algorithm for checking correspondences on the last game.

Example: a nonce challenge

Simple example inspired by the corrected Woo-Lam public-key protocol (1997)

$$B \rightarrow A : (N, B)$$
$$A \rightarrow B : \{pk_A, B, N\}_{sk_A}$$

In our language:

$$
\begin{array}{l}
c0(); \textbf{new } rk_A : keyseed; \\
\textbf{let } pk_A = pkgen(rk_A) \textbf{ in let } sk_A = skgen(rk_A) \textbf{ in } \overline{c1}\langle pk_A \rangle; \\
\quad !^{i_A \leq n} c2[i_A](xN : nonce, xB : host); \textbf{event } e_A(pk_A, xB, xN); \\
\qquad \textbf{new } r : seed; \overline{c3[i_A]}\langle sign(concat(pk_A, xB, xN), sk_A, r) \rangle \\
| \; !^{i_B \leq n} c4[i_B](xpk_A : pkey); \textbf{new } N : nonce; \overline{c5[i_B]}\langle N, B \rangle; \\
\qquad c6[i_B](s : signature); \textbf{if } verify(concat(xpk_A, B, N), xpk_A, s) \textbf{ then} \\
\qquad \textbf{if } xpk_A = pk_A \textbf{ then event } e_B(xpk_A, B, N)
\end{array}
$$

Arrays

All variables defined under replications are implicitly arrays.
This allows us to store all values that occur during the executions of the game.
This replaces lists used by cryptographers and is key to cryptographic proofs.

$c0(); \textbf{new } rk_A : keyseed;$

$\textbf{let } pk_A = pkgen(rk_A) \textbf{ in let } sk_A = skgen(rk_A) \textbf{ in } \overline{c1}\langle pk_A \rangle;$

$\quad !^{i_A \leq n} c2[i_A](xN[i_A] : nonce, xB[i_A] : host);$

$\qquad \textbf{event } e_A(pk_A, xB[i_A], xN[i_A]); \textbf{new } r[i_A] : seed;$

$\qquad \overline{c3[i_A]}\langle sign(concat(pk_A, xB[i_A], xN[i_A]), sk_A, r[i_A])\rangle$

$\mid !^{i_B \leq n} c4[i_B](xpk_A[i_B] : pkey); \textbf{new } N[i_B] : nonce; \overline{c5[i_B]}\langle N[i_B], B\rangle;$

$\qquad c6[i_B](s[i_B] : signature);$

$\qquad \textbf{if } verify(concat(xpk_A[i_B], B, N[i_B]), xpk_A[i_B], s[i_B]) \textbf{ then}$

$\qquad \textbf{if } xpk_A[i_B] = pk_A \textbf{ then event } e_B(xpk_A[i_B], B, N[i_B])$

After game transformations

Using the unforgeability of signatures, the signature verification with pk_A succeeds only for signatures generated with sk_A.
After game transformations, we obtain the last game:

$$
\begin{array}{l}
c0(); \textbf{new } rk_A : keyseed; \textbf{let } pk_A = pkgen'(rk_A) \textbf{ in } \overline{c1}\langle pk_A\rangle;\\
\quad !^{i_A \leq n} c2[i_A](xN : nonce, xB : host);\\
\qquad \textbf{event } e_A(pk_A, xB, xN); \textbf{let } m = concat(pk_A, xB, xN) \textbf{ in}\\
\qquad \textbf{new } r : seed; \overline{c3[i_A]}\langle sign'(m, skgen'(rk_A), r)\rangle\\
\mid\ !^{i_B \leq n} c4[i_B](xpk_A : pkey); \textbf{new } N : nonce; \overline{c5[i_B]}\langle N, B\rangle;\\
\qquad c6[i_B](s : signature); \textbf{find } u \leq n \textbf{ suchthat}\\
\qquad\quad \textbf{defined}(m[u], xB[u], xN[u]) \wedge (xpk_A = pk_A) \wedge (B = xB[u])\\
\qquad\quad \wedge (N = xN[u]) \wedge verify'(concat(xpk_A, B, N), xpk_A, s) \textbf{ then}\\
\qquad \textbf{event } e_B(xpk_A, B, N)
\end{array}
$$

Non-injective correspondences

A non-injective correspondence is a formula of the form $\psi \Rightarrow \phi$ where

$\phi ::=$	formula
M	term (without arrays)
$\mathit{event}(e(M_1, \ldots, M_m))$	event
$\phi_1 \wedge \phi_2$	conjunction
$\phi_1 \vee \phi_2$	disjunction

and ψ is a formula that contains only events and conjunctions.

Example

$$\mathit{event}(e_B(x, y, z)) \Rightarrow \mathit{event}(e_A(x, y, z))$$

means that, if $e_B(x, y, z)$ is executed, then $e_A(x, y, z)$ has also been executed (except in cases of negligible probability).

Non-injective correspondences: formal semantics

Let ρ be an environment that maps variables to bitstrings.
Let $\mathcal{E}$ be a sequence of events.

Definition

$\rho, \mathcal{E} \vdash M$ if and only if M evaluates to *true* in environment ρ
$\rho, \mathcal{E} \vdash \mathit{event}(e(M_1, \dots, M_m))$ if and only if
for all $j \leq m$, M_j evaluates to a_j in ρ and $e(a_1, \dots, a_m) \in \mathcal{E}$

Definition

$\mathcal{E} \vdash \psi \Rightarrow \phi$ if and only if
for all ρ defined on $\mathrm{var}(\psi)$ such that $\rho, \mathcal{E} \vdash \psi$,
there exists an extension ρ' of ρ to $\mathrm{var}(\phi)$ such that $\rho', \mathcal{E} \vdash \phi$.

Definition

Q_0 satisfies $\psi \Rightarrow \phi$ with public variables V if and only if
for all evaluation contexts C accessing only variables of V in Q_0,
$\Pr[C[Q_0]$ executes $\mathcal{E}$ and $\mathcal{E} \not\vdash \psi \Rightarrow \phi]$ is negligible.

Injective correspondences

An injective correspondence also allows injective events $\mathit{inj\text{-}event}(e(M_1, \ldots, M_m))$.

Each execution of the injective events in ψ corresponds to distinct injective events in ϕ.

Example

$$\mathit{inj\text{-}event}(e_B(x, y, z)) \Rightarrow \mathit{inj\text{-}event}(e_A(x, y, z))$$

means that each execution of $e_B(x, y, z)$ corresponds to a distinct execution of $e_A(x, y, z)$.

Intuition for the proof: non-injective correspondences (1)

Prove the correspondence $event(e_B(x, y, z)) \Rightarrow event(e_A(x, y, z))$ in the game

$$\ldots !^{i_A \leq n} \ldots \textbf{event } e_A(pk_A, xB, xN); \textbf{let } m = \ldots \textbf{ in } \ldots$$
$$\mid\ !^{i_B \leq n} \ldots \textbf{find } u \leq n \textbf{ suchthat } \textbf{defined}(m[u], xB[u], xN[u]) \wedge$$
$$(xpk_A = pk_A) \wedge (B = xB[u]) \wedge (N = xN[u]) \wedge$$
$$verify'(concat(xpk_A, B, N), xpk_A, s) \textbf{ then } \textbf{event } e_B(xpk_A, B, N)$$

Intuition for the proof: non-injective correspondences (1)

Prove the correspondence $event(e_B(x,y,z)) \Rightarrow event(e_A(x,y,z))$ in the game

$$\ldots !^{i_A \le n} \ldots \textbf{event } e_A(pk_A, xB, xN); \textbf{let } m = \ldots \textbf{ in } \ldots$$

$$| \ !^{i_B \le n} \ldots \textbf{find } u \le n \textbf{ suchthat defined}(m[u], xB[u], xN[u]) \wedge$$
$$(xpk_A = pk_A) \wedge (B = xB[u]) \wedge (N = xN[u]) \wedge$$
$$verify'(concat(xpk_A, B, N), xpk_A, s) \textbf{ then event } e_B(xpk_A, B, N)$$

If $event(e_B(x,y,z))$ has been executed, the program point **event** $e_B(xpk_A, B, N)$ has been reached for some $i_B = i'_B$, and $e_B(x,y,z) = e_B(xpk_A[i'_B], B, N[i'_B])$.

So $m[u[i'_B]]$, $x_B[u[i'_B]]$, and $xN[u[i'_B]]$ are defined, $xpk_A[i'_B] = pk_A$, $B = xB[u[i'_B]]$, and $N[i'_B] = xN[u[i'_B]]$.

Since $m[u[i'_B]]$ is defined, the definition of $m[i_A]$ has been executed for $i_A = u[i'_B]$, so **event** $e_A(pk_A, xB[i_A], xN[i_A])$ has been executed.

If $event(e_B(x, y, z))$ has been executed, the program point **event** $e_B(xpk_A, B, N)$ has been reached for some $i_B = i'_B$, and $e_B(x, y, z) = e_B(xpk_A[i'_B], B, N[i'_B])$.

So $m[u[i'_B]]$, $x_B[u[i'_B]]$, and $xN[u[i'_B]]$ are defined, $xpk_A[i'_B] = pk_A$, $B = xB[u[i'_B]]$, and $N[i'_B] = xN[u[i'_B]]$.

Since $m[u[i'_B]]$ is defined, the definition of $m[i_A]$ has been executed for $i_A = u[i'_B]$, so **event** $e_A(pk_A, xB[i_A], xN[i_A])$ has been executed.

We have

- $x = xpk_A[i'_B] = pk_A$
- $y = B = xB[u[i'_B]] = xB[i_A]$
- $z = N[i'_B] = xN[u[i'_B]] = xN[i_A]$

so $e_A(pk_A, xB[i_A], xN[i_A]) = e_A(x, y, z)$ has been executed.

Intuition for the proof: injective correspondences (1)

Prove the correspondence
$\mathit{inj\text{-}event}(e_B(x, y, z)) \Rightarrow \mathit{inj\text{-}event}(e_A(x, y, z))$ in the game

$\ldots !^{i_A \leq n} \ldots \textbf{event } e_A(pk_A, xB, xN); \textbf{let } m = \ldots \textbf{ in } \ldots$

$|\ !^{i_B \leq n} \ldots \textbf{find } u \leq n \textbf{ suchthat } \textbf{defined}(m[u], xB[u], xN[u]) \wedge$
$\quad (xpk_A = pk_A) \wedge (B = xB[u]) \wedge (N = xN[u]) \wedge$
$\quad \mathit{verify}'(\mathit{concat}(xpk_A, B, N), xpk_A, s) \textbf{ then } \textbf{event } e_B(xpk_A, B, N)$

Intuition for the proof: injective correspondences (2)

Prove the correspondence
$inj\text{-}event(e_B(i, x, y, z)) \Rightarrow inj\text{-}event(e_A(i', x, y, z))$ in the game

$$\ldots !^{i_A \leq n} \ldots \textbf{event } e_A(i_A, pk_A, xB, xN); \textbf{let } m = \ldots \textbf{ in } \ldots$$
$$| \; !^{i_B \leq n} \ldots \textbf{find } u \leq n \textbf{ suchthat } \textbf{defined}(m[u], xB[u], xN[u]) \wedge$$
$$(xpk_A = pk_A) \wedge (B = xB[u]) \wedge (N = xN[u]) \wedge$$
$$verify'(concat(xpk_A, B, N), xpk_A, s) \textbf{ then } \textbf{event } e_B(i_B, xpk_A, B, N)$$

In order to record in which session each event is executed, we add replication indices to events.

Intuition for the proof: injective correspondences (3)

Prove the correspondence
$\mathit{inj\text{-}event}(e_B(i,x,y,z)) \Rightarrow \mathit{inj\text{-}event}(e_A(i',x,y,z))$ in the game

$\ldots !^{i_A \leq n} \ldots$ **event** $e_A(i_A, pk_A, xB, xN)$; **let** $m = \ldots$ **in** $\ldots$
$|\ !^{i_B \leq n} \ldots$ **find** $u \leq n$ **suchthat** **defined**$(m[u], xB[u], xN[u]) \wedge$
$(xpk_A = pk_A) \wedge (B = xB[u]) \wedge (N = xN[u]) \wedge$
$\mathit{verify}'(\mathit{concat}(xpk_A, B, N), xpk_A, s)$ **then** **event** $e_B(i_B, xpk_A, B, N)$

If $\mathit{event}(e_B(i,x,y,z))$ has been executed, the program point **event** $e_B(i_B, xpk_A, B, N)$ has been reached for some $i_B = i'_B$, and $e_B(i,x,y,z) = e_B(i'_B, xpk_A[i'_B], B, N[i'_B])$.

So $m[u[i'_B]]$, $x_B[u[i'_B]]$, and $xN[u[i'_B]]$ are defined, $xpk_A[i'_B] = pk_A$, $B = xB[u[i'_B]]$, and $N[i'_B] = xN[u[i'_B]]$.

Since $m[u[i'_B]]$ is defined, the definition of $m[i_A]$ has been executed for $i_A = u[i'_B]$, so **event** $e_A(i_A, pk_A, xB[i_A], xN[i_A])$ has been executed.

Intuition for the proof: injective correspondences (4)

As before, $e_A(i_A, pk_A, xB[i_A], xN[i_A]) = e_A(i', x, y, z)$ for some i'.

In order to show injectivity, we show that

e_B executed twice, for $i_B = i'_B$ and $i_B = i''_B$, with $i'_B \neq i''_B$

$\Rightarrow e_A$ executed twice, for $i_A = u[i'_B]$ and $i_A = u[i''_B]$, with $u[i'_B] \neq u[i''_B]$

By contraposition, we show $u[i'_B] = u[i''_B] \Rightarrow i'_B = i''_B$.

$$
\begin{array}{lll}
u[i'_B] = u[i''_B] & \Rightarrow & xN[u[i'_B]] = xN[u[i''_B]] \\
 & \Rightarrow & N[i'_B] = N[i''_B] \qquad \text{since } xN[u[i'_B]] = N[i'_B] \\
 & & \qquad\qquad\qquad\quad \text{and } xN[u[i''_B]] = N[i''_B] \\
 & \Rightarrow & i'_B = i''_B \text{ up to negligible probability} \\
 & & \qquad\qquad\qquad\quad \text{by eliminating collisions.}
\end{array}
$$

Proof technique

In order to prove $\psi \Rightarrow \phi$, two main steps:

1. Collect the facts that hold when the events in ψ are executed.
2. Reason on these facts using an equational prover in order to show that the events in ϕ have been executed (and show injectivity when needed).

We shall now detail these points.

Collecting true facts

For each program point P, we collect a set of true facts at that point $\mathcal{F}_P$.

- We take into account assignments and tests above P.

Example

In **if** M **then** P, $M \in \mathcal{F}_P$.

- We take into account facts that hold at all definitions of variables.

Example

If $\mathbf{defined}(x[\widetilde{M}]) \in \mathcal{F}_P$ and M holds at all definitions of $x[\widetilde{i}]$, then $M\{\widetilde{M}/\widetilde{i}\} \in \mathcal{F}_P$.

- We take into account that code is always executed up to the next output before switching to another thread.

Collecting true facts: example (1)

$c0(); \textbf{new } rk_A : keyseed; \textbf{let } pk_A = pkgen'(rk_A) \textbf{ in } \overline{c1}\langle pk_A \rangle;$

$\quad !^{i_A \leq n} c2[i_A](xN : nonce, xB : host);$

$\qquad \textbf{event } e_A(pk_A, xB, xN); \textbf{let } m = concat(pk_A, xB, xN) \textbf{ in}$

$\qquad \textbf{new } r : seed; \overline{c3[i_A]}\langle sign'(m, skgen'(rk_A), r) \rangle$

$\mid !^{i_B \leq n} c4[i_B](xpk_A : pkey); \textbf{new } N : nonce; \overline{c5[i_B]}\langle N, B \rangle;$

$\qquad c6[i_B](s : signature); \textbf{find } u \leq n \textbf{ suchthat}$

$\qquad\quad \textbf{defined}(m[u], xB[u], xN[u]) \wedge (xpk_A = pk_A) \wedge (B = xB[u])$

$\qquad\quad \wedge (N = xN[u]) \wedge verify'(concat(xpk_A, B, N), xpk_A, s) \textbf{ then}$

$\qquad \textbf{event } e_B(xpk_A, B, N)$

At program point $P = \textbf{event } e_B(xpk_A, B, N)$,

$\mathcal{F}_P = \{\textbf{defined}(m[u[i_B]]), \textbf{defined}(xB[u[i_B]]), \textbf{defined}(xN[u[i_B]]),$

$\qquad xpk_A[i_B] = pk_A, B = xB[u[i_B]], N[i_B] = xN[u[i_B]], \ldots$

Collecting true facts: example (2)

$$
\begin{aligned}
& c0(); \textbf{new } rk_A : \mathit{keyseed}; \textbf{let } pk_A = \mathit{pkgen}'(rk_A) \textbf{ in } \overline{c1}\langle pk_A \rangle; \\
& \quad !^{i_A \le n} c2[i_A](xN : \mathit{nonce}, xB : \mathit{host}); \\
& \qquad \textbf{event } e_A(pk_A, xB, xN); \textbf{let } m = \mathit{concat}(pk_A, xB, xN) \textbf{ in} \\
& \qquad \textbf{new } r : \mathit{seed}; \overline{c3[i_A]}\langle \mathit{sign}'(m, \mathit{skgen}'(rk_A), r) \rangle \\
& \mid \ldots
\end{aligned}
$$

At program point $P = \textbf{event } e_B(xpk_A, B, N)$,

$$
\begin{aligned}
\mathcal{F}_P = \{ & \textbf{defined}(m[u[i_B]]), \textbf{defined}(xB[u[i_B]]), \textbf{defined}(xN[u[i_B]]), \\
& xpk_A[i_B] = pk_A, B = xB[u[i_B]], N[i_B] = xN[u[i_B]], \\
& \mathit{event}(e_A(pk_A, x_B[u[i_B]], xN[u[i_B]])), \ldots \}
\end{aligned}
$$

because $\textbf{defined}(m[u[i_B]]) \in \mathcal{F}_P$.

Equational prover

We use an algorithm inspired by the Knuth-Bendix completion algorithm to derive new equalities from known equalities.

The equational prover also eliminates collisions when they have negligible probability.

Details of this prover can be found in the paper:
Blanchet, A Computationally Sound Mechanized Prover for Security Protocols, TDSC, to appear.

We say that $\mathcal{F}$ yields a contradiction when the prover starting from $\mathcal{F}$ derives *false*.

Proof of non-injective correspondences (1)

Let $\psi \Rightarrow \phi = F_1 \wedge \ldots \wedge F_m \Rightarrow \phi$ be a non-injective correspondence, with fresh variables.

Example

$event(e_B(x, y, z)) \Rightarrow event(e_A(x, y, z))$.

If $F_1, \ldots, F_m$ have been executed,
then there exist $P_1, \ldots, P_m$ such that, for all $j \leq m$,

- $F_j = event(e_j(M_{j1}, \ldots, M_{jm_j}))$,
- **event** $e_j(M'_{j1}, \ldots, M'_{jm_j}); P_j$ occurs in Q_0, and
- **event** $e_j(M'_{j1}, \ldots, M'_{jm_j})$ has been executed with $F_j = \theta'_j event(e_j(M'_{j1}, \ldots, M'_{jm_j}))$, where θ'_j renames the replication indices at P_j to fresh replication indices.

Example

event $e_B(xpk_A, B, N)$ has been executed, with
$event(e_B(x, y, z)) = event(e_B(xpk_A[i'_B], B, N[i'_B]))$, $\theta' = \{i'_B / i_B\}$.

Proof of non-injective correspondences (2)

Then the facts $\mathcal{F}_j = \theta'_j \mathcal{F}_{P_j} \cup \{\theta'_j M'_{j1} = M_{j1}, \dots, \theta'_j M'_{jm_j} = M_{jm_j}\}$ hold.

Example

$\mathcal{F}_P\{i'_B/i_B\} \cup \{x = xpk_A[i'_B], y = B, z = N[i'_B]\}$ hold, where $\mathcal{F}_P$ has been described previously.

For each such $P_1, \dots, P_m$, we show that

$$\mathcal{F} = \mathcal{F}_1 \cup \dots \cup \mathcal{F}_m \text{ implies } \theta\phi$$

for some θ equal to the identity on $\mathrm{var}(\psi)$, by the equational prover.

(For this proof, we prove atomic facts contained in $\theta\phi$, we choose θ by matching facts to prove with elements of $\mathcal{F}$, and we show that $\mathcal{F}$ implies M by showing that $\mathcal{F} \cup \{\neg M\}$ yields a contradiction.)

Example

$x = xpk_A[i'_B], y = B, z = N[i'_B], xpk_A[i'_B] = pk_A, B = xB[u[i'_B]]$, $N[i'_B] = xN[u[i'_B]]$, $event(e_A(pk_A, x_B[u[i'_B]], xN[u[i'_B]]))$ imply $event(e_A(x, y, z))$.

Authentication and key exchange

We have also given:

- a proof of mutual authentication from correspondence assertions;
- a proof of authenticated key exchange from a combination of correspondence assertions and secrecy (also shown by our prover).

  Difficulty: the secrecy is the secrecy of a single variable, the shared key is stored in two variables, k_A in participant A, k_B in participant B.

  Intuitively, we show by correspondences that each key k_B of B with A is an element of array k_A, so secrecy of k_A is sufficient.

Details in the paper.

Experimental results

We have tested our prover on the following protocols:

- Woo-Lam shared-key original and corrected versions
- Woo-Lam public-key original and corrected versions
- Needham-Schroeder public-key original and corrected versions
- Denning-Sacco public-key original and corrected versions
- Needham-Schroeder shared-key original and corrected versions, with and without key confirmation
- Yahalom, with and without key confirmation
- Otway-Rees

We try to prove authentication and authenticated key exchange, as appropriate for each protocol.

The prover obviously fails proving false properties.

It succeeds proving true ones, except in one case:
the original version of the Needham-Schroeder shared-key protocol.
(It does not see that $N_B[i] \neq N_B[i'] - 1$ except in cases of negligible probability.)

Conclusion and future work

The prover succeeds proving most desired correspondences, but some points not directly related to correspondences could still be improved:

- automatic proof strategy: the prover sometimes needs indications from the user.
- equational theories, e.g. handle the equations of XOR.
- handle more primitives, e.g. Diffie-Hellman key agreements.

Tool available at
`http://www.di.ens.fr/~blanchet/cryptoc.html`