notes

Eike Ritter

03 June 2015

Contents

1 Stateful injective agreement

1. Prove non-injective correspondence between $begin(e1(M_1))$ and $end(e2(M_2))$ as usual, it works! The use of sequence predicate chains correctly the two sessions, since the state produced by the begin event needs to be reachable in order for the end event to succeed.

2. Once non-injectivity is proven, treat $e2(M_2)$ both as a begin (now called end1) and as an end (called end2) event, inserting the clauses: H $\wedge end1(e2(M_2)) \Rightarrow mid$ $H \wedge mid \Rightarrow end2(e2(M_2))$ Intuitively this forces the resolution to produce clauses that contain $end1$ on the l.h.s. and $end2$ on the r.h.s.

3. Saturate the clauses, ensuring that Horn clauses of the form $H \wedge end1(e2(M_2)) \Rightarrow C$ are not eliminated as redundant because $H \Rightarrow C$ is present. It is important to note that this change in saturation does not influence termination, rather it *doubles* the amount of clauses produced in case both $H \wedge end1(e1(M_2)) \Rightarrow C$ and $H \Rightarrow C$ are present.

4. Look for rules in the saturation of the form:

$$H \wedge end1(e2(M_2)) \Rightarrow end2(e2(M_2'))$$

   Conjecture: Let $\sigma = mgu(M_2, M_2')$. If two occurrences of $begin(e1(M_1))$ are forced to unify in H then the protocol is secure, otherwise a potential attack is found. The intuitive reason why this should work is that if only one *begin* event is found, and both *end1* and *end2* events appear in the clause, then in the abstraction one begin event could have generated two end events, hence there is a potential attack. If two begin events are present in the hypothesis, then they are marked with different terms, representing potentially different sessions of the process that executed the begin event. If unifying the *end1* and *end2* events triggers also the unification of the two instances of *begin*, this means that the two occurrences represent one single session, hence the clause does not represent an attack.

   This doesn't hold as we were able to prove a counter example (protocol in test.pv that does not have injective agreement property).

2 Example:

GateCard.pv generates the following clauses, when testing non-injective agreement between $permit(x, y) \Rightarrow add(x)$: $s_1 \neq 0 \wedge att((s_1, 0), y) \wedge seq(s, (s_1, 0)) \Rightarrow end(permit(s_1, y))$
$s_1 \neq 0 \wedge att((s_1, 0), y) \wedge seq(s, (s_1, 0)) \Rightarrow seq((s_1, 0), (0, 0))$
$s_1 \neq 0 \wedge seq((s_1, 0), (0, 0)) \wedge att((s_1, 0), y) \wedge seq(s, (s_1, 0)) \Rightarrow att((0, 0), sign((s_1, y), K[]))$
$s_2 \neq 0 \wedge att((s_1, s_2), y) \wedge seq(s, (s_1, s_2)) \Rightarrow end(permit(s_2, y))$
$s_2 \neq 0 \wedge att((s_1, s_2), y) \wedge seq(s, (s_1, s_2)) \Rightarrow seq((s_1, s_2), (s_1, 0))$
$s_2 \neq 0 \wedge seq((s_1, s_2), (s_1, 0)) \wedge att((s_1, s_2), y) \wedge seq(s, (s_1, s_2)) \Rightarrow att((s_1, 0), sign((s_2, y), K[]))$
$seq(s, s') \Rightarrow att(s', m[i])$
$begin(add(m[i])) \wedge att((0, s_2), sign(m[i], L[])) \wedge seq(s, (0, s_2)) \Rightarrow seq((0, s_2), (m[i], 0))$
$s_1 \neq 0 \wedge begin(add(m[i])) \wedge att((s_1, 0), sign(m[i], L[])) \wedge seq(s, (s_1, 0)) \Rightarrow seq((s_1, 0), (s_1, m[i]))$
When testing injective agreement we produce the following clauses: $s_1 \neq 0 \wedge att((s_1, 0), y) \wedge seq(s, (s_1, 0)) \wedge mid \Rightarrow end2(permit(s_1, y))$
$s_1 \neq 0 \wedge att((s_1, 0), y) \wedge seq(s, (s_1, 0)) \wedge end1(permit(s_1, y)) \Rightarrow mid$
$s_1 \neq 0 \wedge att((s_1, 0), y) \wedge seq(s, (s_1, 0)) \wedge end1(permit(s_1, y)) \wedge mid \Rightarrow seq((s_1, 0), (0, 0))$
$s_1 \neq 0 \wedge seq((s_1, 0), (0, 0)) \wedge att((s_1, 0), y) \wedge seq(s, (s_1, 0)) \wedge end1(permit(s_1, y)) \wedge mid$
$\Rightarrow att((0, 0), sign((s_1, y), K[]))$
$s_2 \neq 0 \wedge att((s_1, s_2), y) \wedge seq(s, (s_1, s_2)) \wedge mid \Rightarrow end2(permit(s_2, y))$
$s_2 \neq 0 \wedge att((s_1, s_2), y) \wedge seq(s, (s_1, s_2)) \wedge end1(permit(s_2, y)) \Rightarrow mid$
$s_2 \neq 0 \wedge att((s_1, s_2), y) \wedge seq(s, (s_1, s_2)) \wedge end1(permit(s_2, y)) \wedge mid \Rightarrow seq((s_1, s_2), (s_1, 0))$
$s_2 \neq 0 \wedge seq((s_1, s_2), (s_1, 0)) \wedge att((s_1, s_2), y) \wedge seq(s, (s_1, s_2)) \wedge end1(permit(s_2, y)) \wedge mid$
$\Rightarrow att((s_1, 0), sign((s_2, y), K[]))$
$seq(s, s') \Rightarrow att(s', m[i])$

$begin(add(m[i])) \wedge att((0, s_2), sign(m[i], L[])) \wedge seq(s, (0, s_2)) \Rightarrow seq((0, s_2), (m[i], 0))$
$s_1 \neq 0 \wedge begin(add(m[i])) \wedge att((s_1, 0), sign(m[i], L[])) \wedge seq(s, (s_1, 0)) \Rightarrow seq((s_1, 0), (s_1, m[i]))$
Somehow we must record that the $m[i]$ generated by the card when adding a new token are *fresh*, hence different from any other token produced.

Saturation should produce the clause:

$$begin(add(m[i])) \wedge end1(permit(m[i], n[j])) \wedge begin(add(m[i'])) \Rightarrow end2(permit(m[i'], n[j']))$$

with the unification resulting in the following substitution:

$$\sigma = \{i/i', j/j'\}$$

3 Learned lesson:

1. The simple idea of checking $end1end2$ is sound but could give rise to false attacks. Instead of the *mid* predicate we should use correctly the *seq* predicate to capture the state dependency between *end1* and *end2*. As a solution for reaching $end1end2$ in stateless protocols it is possible to insert a dummy $seq(\phi, \phi)$ predicate so that the rules for *end1* and *end2* can be chained.

2. Using the *seq* predicate instead of *mid* is not good for non-termination.

3. What is not captured is the information on the state

$$(0,0) \rightarrow (m[i], 0) \rightarrow (0,0) \rightarrow (m[j], 0)$$

   it works because we know by freshness that $i \neq j$, hence we cannot reach the state $end1(x)end2(x)$, because it would force unification of i and j.

4. Idea:

1. the attacker knowledge does not increase in the second round of the protocol

2. the control flow in the second iteration is the same as in the first iteration, so it makes no sense to go on in this cycle.

4 Problems with Locks

4.1 Visible intermediate states

1. Locks are too coarse. For example the process:

$$lock(s_1, s_2, s_3); new\ a; out(a); set\ a \in s_1; set\ a \in s_2;$$

would generate:

$$att(a[0, 0, x3])$$
$$implies(a[0, 0, x3], a[1, 0, x3])$$
$$implies(a[1, 0, x3], a[1, 1, x3])$$

and hence the predicate

$$att(a[1, 0, x3])$$

is reachable, although one would like only the final state, namely

$$att(a[1, 1, x3])$$

to be visible from the outside.

4.2 Set-abstraction multi-value interleavings

1. For the set abstraction, one of the problems is the locality of the states in transitions

$$att(sign(xm[x1, x2, x3], xc[1], k[]))implies(xm[x1, x2, x3], xm[x1, 1, x3])$$
$$att(sign(xm[x1, x2, x3], xc[1], k[]))implies(xc[1], xc[2])$$

2. we get all the diamond of interleavings:

$$\begin{tikzcd}
 & \frac{m}{110}\,\frac{c}{0} \arrow[dr] & \\
\frac{m}{100}\,\frac{c}{0} \arrow[ur] \arrow[dr] & & \frac{m}{110}\,\frac{c}{1} \\
 & \frac{m}{100}\,\frac{c}{1} \arrow[ur] &
\end{tikzcd}$$

3. which is bad because the intermediate states should not be visible. We would like to see the following state transition system:

$$\frac{m}{100}\,\frac{c}{0} \longrightarrow \frac{m}{110}\,\frac{c}{1}$$

4. we could solve it by:

$$att(sign(xm[x1, x2, x3], xc[1], k[]))implies([xm[x1, x2, x3], xc[1]], [xm[x1, 1, x3], xc[2]])$$

$$att(sign(m[x1, x2, x3], xc[1], k[]))\wedge implies([xm[x1, x2, x3], xc[1]], [xm[x1, 1, x3], xc[2]])att(sign(m[x1, 1, x$$

[1] FOOTNOTE DEFINITION NOT FOUND: 0

[2] FOOTNOTE DEFINITION NOT FOUND: 1