Model checking secrecy

Sjouke Mauw
joint work with Cas Cremers

ECSS group
Eindhoven University of Technology
The Netherlands

TU/e
Outline

- Security protocols and secrecy.
- Example: Bilateral Key Exchange.
- Model checking algorithm.
- Comparison.
- Concluding remarks.
Motivation

“Security protocols are three-line programs that people still manage to get wrong”

(Roger Needham)

- Security protocol = set of interaction rules to guarantee security property (plus some intended functionality).
- Formal validation is imperative and feasible.
- Security properties: secrecy, authentication, non-repudiation, availability, ...
Model checking secrecy

- Well-understood property.
- Several tools available.
- Often: general purpose model checker instantiated for this problem.

Conjecture.
A model checker dedicated to verifying secrecy in security protocols will outperform general purpose model checkers applied to this problem.
Example

Bilateral Key Exchange (BKE):

Given a Public Key Infrastructure, two agents should agree upon the value of a freshly generated symmetric key. This key should remain secret.
$SK_i, PK_i, PK_r$

1. Nonce $n_i$

\[\{n_i, I\}_{PK_r}\]

2. Nonce $n_r$

\[\{h(n_i), n_r, kir\}_{PK_i}\]

\[\{h(n_r)\}_{kir}\]

Secret $kir$
Intruder model (Dolev-Yao)

- Intruder has complete control over network.
Intruder model (Dolev-Yao)

- Intruder has complete control over network.
- Intruder can pack/unpack messages as long as he knows the cryptographic key.

Possibly conspiring agents, i.e. intruder knows their secret keys.
Intruder model (Dolev-Yao)

- Intruder has complete control over network.
- Intruder can pack/unpack messages as long as he knows the cryptographic key.
- Possibly conspiring agents, i.e. intruder knows their secret keys.
Finite scenario

$SK_a, PK_a, PK_b$

$a : I(a, b)$

nonce $na$
var $U, K$

$\{na, a\}_{PK_b}$

$\{h(na), U, K\}_{PK_a}$

$\{h(U)\}_K$

secret $K$

$SK_a, PK_a, PK_e$

$a : I(a, e)$

nonce $na'$
var $V, L$

$\{na', a\}_{PK_e}$

$\{h(na'), V, L\}_{PK_a}$

$\{h(V)\}_L$

$SK_b, PK_b, PK_a$

$b : R(a, b)$

nonce $nb$
key $k_{ab}$
var $W$

$\{W, a\}_{PK_b}$

$\{h(W), nb, k_{ab}\}_{PK_a}$

$\{h(nb)\}_{k_{ab}}$

secret $k_{ab}$
State space

Initial intruder knowledge: $PKa, PKb, PKe, SKe$

1: $s(a, b, \{na, a\}_{PKb})$
State space

Initial intruder knowledge: \( PK_a, PK_b, PK_e, SK_e \)

1: \( s(a, b, \{na, a\}_{PK_b}) \)

2: \( r(a, b, \{na, a\}_{PK_b}) \)

3: \( r(a, b, \{na, a\}_{PK_b}) \)
State space

Initial intruder knowledge: $PK_a, PK_b, PK_e, SK_e$

1: $s(a, b, \{na, a\}_{PK_b})$

2: $r(a, b, \{na, a\}_{PK_b})$

3: $s(b, a, \{h(na), nb, kab\}_{PK_a})$

4: $\{h(na), nb, kab\}_{PK_a}$
State space

Initial intruder knowledge: $PK_a, PK_b, PK_e, SK_e$

1 : $r(b, a, \{h(na), nb, kab\}_{PK_a})$

3 : $s(b, a, \{h(na), nb, k_{ab}\}_{PK_a})$

3 : $r(a, b, \{na, a\}_{PK_b})$

1 : $s(a, b, \{na, a\}_{PK_b})$
State space

Initial intruder knowledge: $PK_a, PK_b, PK_e, SK_e$

1: $s(a, b, \{h(nb)\}_{kab})$

1: $r(b, a, \{h(na), nb, kab\}_{PK_a})$

$\{h(na), nb, kab\}_{PK_a}$

3: $s(b, a, \{h(na), nb, kab\}_{PK_a})$

3: $r(a, b, \{na, a\}_{PK_b})$

$\{na, a\}_{PK_b}$

1: $s(a, b, \{na, a\}_{PK_b})$
State space

Initial intruder knowledge: $PK_a, PK_b, PK_e, SK_e$

1. $s(a, b, \{h(nb)\}_{kab})$
2. $r(b, a, \{h(na), nb, k_{ab}\}_{PK_a})$
3. $s(b, a, \{h(na), nb, k_{ab}\}_{PK_a})$
4. $r(a, b, \{na, a\}_{PK_b})$
5. $s(a, b, \{na, a\}_{PK_b})$
6. $r(a, b, \{h(nb)\}_{kab})$
7. $\{h(nb)\}_{kab}$
State space

Initial intruder knowledge: $PK_a, PK_b, PK_e, SK_e$
State space

Initial intruder knowledge: $PKa, PKb, PKe, SKe$

1: $s(a, b, \{h(nb)\}_kab)$

2: $s(a, e, \{na', a\}_PKe)$

3: $r(a, b, \{h(nb)\}_kab)$

3: $r(b, a, \{h(na), nb, kab\}_PKa)$

3: $s(b, a, \{h(na), nb, kab\}_PKa)$

1: $s(a, b, \{na, a\}_PKb)$

1: $r(b, a, \{h(na), nb, kab\}_PKa)$

1: $s(a, b, \{h(na), nb, kab\}_PKa)$

3: $r(a, b, \{na, a\}_PKb)$
State space

Initial intruder knowledge: $PK_a, PK_b, PK_e, SK_e$
State space

Initial intruder knowledge: $PK_a, PK_b, PK_e, SK_e$

1: $s(a, b, \{h(nb)\}_{kab})$
2: $s(a, e, \{na', a\}_{PK_e})$
3: $r(a, b, \{na, a\}_{PK_b})$
3: $r(a, b, \{h(na'), nb, k_{ab}\}_{PK_a})$
3: $s(b, a, \{h(na'), nb, k_{ab}\}_{PK_a})$
3: $r(a, b, \{h(na'), nb, k_{ab}\}_{PK_a})$
1: $r(b, a, \{h(na), nb, k_{ab}\}_{PK_a})$
1: $s(a, b, \{h(nb)\}_{kab})$

TU/e
Correctness criterion

A security protocol is correct w.r.t. secrecy if

- for every finite scenario,
- for every possible trace of that scenario, under control of the intruder,
- whenever an agent reaches a secrecy claim,
- the claimed secret will never occur in the intruder knowledge.
**Auxiliary definitions**

**match**  The match function determines whether the intruder can satisfy the required message format.

**enabled**  An event is enabled if it is the first to be executed in a run and in case it is a read it must have a match with the intruder knowledge.

**after**  The after function returns the new system state after executing a run.
General model checking algorithm

$$\text{modelcheck}(\sigma) =$$

if $\sigma$ does not satisfy property then
  exit ("property fails")
else
  for all $ev \in \text{enabled}(\sigma)$ do
    modelcheck(after(\sigma, ev))
  end
end

Correct if state space forms directed acyclic graph.
traverseFull (runs, know, secrets)

if any secret in know then
    exit ("attack") ;
else
    for all ev ∈ enabled (runs, know) do
        if ev = secret (m) then
            traverseFull (after (runs, ev), know, secrets ∪ {m}) ;
        end
        if ev = send (m) then
            traverseFull (after (runs, ev), know ⊕ m, secrets) ;
        end
        if ev = read (m) then
            for all m' ∈ match (know, m) do
                traverseFull (after (runs, read (m'))), know, secrets) ;
            end
        end
    end
end
traverseFull (runs, know, secrets)

if any secret in know then
    exit ("attack") ;
else
    for all ev ∈ enabled(runs, know) do
        if ev = secret(m) then
            traverseFull(after(runs, ev), know, secrets ∪ {m}) ;
        end
        if ev = send(m) then
            traverseFull(after(runs, ev), know ⊕ m, secrets) ;
        end
        if ev = read(m) then
            for all m' ∈ match() do
                traverseFull(runs, read(m')), know, secrets) ;
            end
        end
    end
end
traverseFull \( (\text{runs}, \text{know}, \text{secrets}) \)

\[
\text{if any secret in know then}
\]
\[
\quad \text{exit ("attack") ;}
\]
\[
\text{else}
\]
\[
\quad \text{Choose } ev \in \text{enabled(\text{runs}, \text{know}) do}
\]
\[
\quad \quad \text{if } ev = \text{secret}(m) \text{ then}
\]
\[
\quad \quad \quad \text{traverseFull(after(\text{runs}, ev), know, secrets \cup \{m\}) ;}
\]
\[
\quad \text{end}
\]
\[
\quad \text{if } ev = \text{send}(m) \text{ then}
\]
\[
\quad \quad \text{traverseFull(after(\text{runs}, ev), know \oplus m, secrets) ;}
\]
\[
\quad \text{end}
\]
\[
\quad \text{if } ev = \text{read}(m) \text{ then}
\]
\[
\quad \quad \text{for all } m' \in \text{match(know, m) do}
\]
\[
\quad \quad \quad \text{traverseFull(after(\text{runs, read(m')}, \text{know, secrets}) ;}
\]
\[
\quad \text{end}
\]
\[
\quad \text{end}
\]
\[
\text{end}
\]
\textbf{traverseFull (runs, know, secrets)}

\begin{verbatim}
if any secret in know then
    exit ("attack") ;
else
    Choose \( ev \in enabled(runs, know) \) do
        if \( ev = secret(m) \) then
            traverseFull(after(runs, ev), know, secrets \( \cup \{m\} \) ) ;
        end
        if \( ev = send(m) \) then
            traverseFull(after(runs, \( m \oplus m \), secrets ) ;
        end
        if \( ev = read(m) \) then
            for all \( m' \in read(know, m) \) do
                traverseFull(after(runs, read(m')), know, secrets ) ;
            end
        end
    end
end
\end{verbatim}

Fast but incorrect:
General model checking algorithm with tail recursion

```
modelcheck (σ, except) =
    if σ does not satisfy property then
        exit ("property fails")
    else
        if enabled(σ) \ except ≠ ∅ then
            ev = choose(enabled(σ) \ except);
            modelcheck(after(σ, ev), ∅);
            modelcheck(σ, except ∪ {ev});
        end
    end
end
```
traverseFull2 \( \langle \text{runs}, \text{know}, \text{secrets}, \text{except} \rangle \)

if any secret in know then
- exit ("attack")
else
  if enabled2\( \langle \text{runs}, \text{know}, \text{except} \rangle \) \(\neq\) \(\emptyset\) then
    \(ev = \text{choose}(\text{enabled2}\langle \text{runs}, \text{know}, \text{except} \rangle)\)
    if \(ev = \text{secret}(m)\) then
      traverseFull2(after\(\langle \text{runs}, ev \rangle\), know, secrets \(\cup\) \{m\}, \emptyset)
      traverseFull2\(\langle \text{runs}, \text{know}, \text{secrets}, \text{except} \cup \{ev\} \rangle\)
    end
    if \(ev = \text{send}(m)\) then
      traverseFull2(after\(\langle \text{runs}, ev \rangle\), know \(\oplus\) m, secrets, \emptyset)
      traverseFull2\(\langle \text{runs}, \text{know}, \text{secrets}, \text{except} \cup \{ev\} \rangle\)
    end
    if \(ev = \text{read}(m)\) then
      for all \(m' \in \text{match}(\text{know}, m)\) do
        traverseFull2(after\(\langle \text{runs}, \text{read}(m') \rangle\), know, secrets, \emptyset)
      end
      traverseFull2\(\langle \text{runs}, \text{know}, \text{secrets}, \text{except} \cup \{ev\} \rangle\)
    end
  end
end

For-loop replaced by tail recursion
**traverseFull2** *(runs, know, secrets, except)*

if any secret in know then

    exit ("attack")

else

    if enabled2(runs, know, except) ≠ ∅ then

        ev = choose(enabled2(runs, know, except))

        if ev = secret(m) then

            traverseFull2(after(runs, ev), know, secrets ∪ {m}, ∅)

            traverseFull2(runs, know, secrets, except ∪ {ev})

        end

        if ev = send(m) then

            traverseFull2(after(runs, ev), know ⊕ m, secrets, ∅)

            traverseFull2(runs, know, secrets, except ∪ {ev})

        end

        if ev = read(m) then

            for all m' ∈ match(know, m) do

                traverseFull2(after(runs, read(m')), know, secrets, ∅)

            end

            traverseFull2(runs, know, secrets, except ∪ {ev})

        end

    end

end

enabled2(runs, know, except) = enabled(runs, know) \ except
Partial order reduction

Lemma.
If at a given state closed events $e$ and $f$ from different runs can be executed, then.

■ after executing event $e$, event $f$ can still be executed;
■ after executing event $f$, event $e$ can still be executed;
■ the states reached after $ef$ and $fe$ are both equal.

Example:

$e_1; e_2; e_3; send_1; send_2; e_4; e_5, \ldots$

$e_1; e_2; e_3; send_2; send_1; e_4; e_5, \ldots$

$e_1; e_2; e_3; send_2; e_4; send_1; e_5, \ldots$

All result in the same state, so we only have to traverse one of these.
traverseFull2 (runs, know, secrets, except)

if any secret in know then
    exit ("attack") ;
else
    if enabled2(runs, know, except) ≠ ∅ then
        ev = choose(enabled2(runs, know, except)) ;
        if ev = secret(m) then
            traverseFull2(after(runs, ev), know, secrets U {m}, ∅) ;
            traverseFull2(runs, know, secrets, except U {ev})
        end
        if ev = send(m) then
            traverseFull2(after(runs, ev), know ⊕ m, secrets, ∅) ;
            traverseFull2(runs, know, secrets, except U {ev})
        end
        if ev = read(m) then
            for all m' ∈ match(know, m) do
                traverseFull2(after(runs, read(m')), know, secrets, ∅) ;
            end
            traverseFull2(runs, know, secrets, except U {ev})
        end
    end
end
end
traverseFull2 \( (\text{runs}, \text{know}, \text{secrets}, \text{except}) \)

\[
\begin{array}{l}
\text{if any secret in know then} \\
\quad \text{exit ("attack");} \\
\text{else} \\
\quad \text{if enabled2(\text{runs, know, except})} \neq \emptyset \text{ then} \\
\qquad \text{ev = choose(enabled2(\text{runs, know, except}));} \\
\qquad \text{if ev = secret(m) then} \\
\qquad \quad \text{traverseFull2(after(\text{runs, ev}), know, secrets} \cup \{m\}, \emptyset); \\
\qquad \quad \text{traverseFull2(\text{runs, know, secrets, except} \cup \{ev\})}; \\
\quad \end{array}
\]

\[
\begin{array}{l}
\quad \text{if ev = send(m) then} \\
\qquad \text{traverseFull2(after(\text{runs, ev}), know} \oplus m, \text{secrets, } \emptyset); \\
\qquad \text{traverseFull2(\text{runs, know, secrets, except} \cup \{ev\})} \\
\qquad \text{end} \\
\quad \text{if ev = read(m) then} \\
\qquad \text{for all } m' \in \text{match(know, m) do} \\
\qquad \quad \text{traverseFull2(after(\text{runs, read(m')}, know, secrets, } \emptyset); \\
\qquad \text{end} \\
\text{end} \\
\text{end}
\end{array}
\]
\textbf{traverseFull2}(\texttt{runs, know, secrets, except})

\begin{verbatim}
if any secret in know then
  exit ("attack")
else
  if \texttt{enabled2(runs, know, except) \neq \emptyset} then
    \texttt{ev = choose(enabled2(runs, know, except))}
    if \texttt{ev = secret(m)} then
      traverseFull2(after(runs, ev), know, secrets \cup \{m\}, \emptyset)
      traverseFull2(runs, know, secrets, except \cup \{ev\})
    end
    if \texttt{ev = send(m)} then
      traverseFull2(after(runs, ev), know \oplus m, secrets, \emptyset)
      traverseFull2(runs, know, secrets, except \cup \{ev\})
    end
    if \texttt{ev = read(m)} then
      for all \texttt{m' \in match(know, m)} do
        traverseFull2(after(runs, read(m')), know, secrets, \emptyset)
      end
      traverseFull2(runs, know, secrets, except \cup \{ev\})
    end
  end
end
\end{verbatim}

Not needed
traverseFull2 (runs, know, secrets, except)

if any secret in know then
    exit ("attack") ;
else
    if enabled2(runs, know, except) ≠ ∅ then
        ev = choose(enabled2(runs, know, except)) ;
        if ev = secret(m) then
            traverseFull2(after(runs, ev), know, secrets ∪ {m}, ∅) ;
            traverseFull2(runs, know, secrets, except ∪ {ev})
        end
        if ev = send(m) then
            traverseFull2(after(runs, ev), know ⊕ m, secrets, ∅) ;
            traverseFull2(runs, know, secrets, except ∪ {ev})
        end
        if ev = read(m) then
            for all m' ∈ match(know, m) do
                traverseFull2(after(runs, read(m')), know, secrets, ∅) ;
            end
            traverseFull2(runs, know, secrets, except ∪ {ev}) ;
        end
    end
end
end
traverse \((\text{runs, know, secrets, forbidden})\)

if any secret in \(\text{know}\) then
    exit ("attack") ;
else
    if \(\text{enabled}(\text{runs, know, forbidden}) \neq \emptyset\) then
        \(ev = \text{choose}(\text{enabled}(\text{runs, know, forbidden}))\)
        if \(ev = \text{secret}(m)\) then
            traverse\((\text{after}(\text{runs, } ev), \text{know, secrets} \cup \{m\}, \text{forbidden})\) ;
        end
        if \(ev = \text{send}(m)\) then
            traverse\((\text{after}(\text{runs, } ev), \text{know} \oplus m, \text{secrets, forbidden})\) ;
        end
        if \(ev = \text{read}(m)\) then
            for all \(m' \in \text{match}(\text{know, m}) \land m' \notin \text{forbidden}(\text{read}(m))\) do
                traverse\((\text{after}(\text{runs, } \text{read}(m')), \text{know, secrets, forbidden})\) ;
            end
            traverse\((\text{runs, know, secrets, forbidden}[ev \rightarrow \text{know}])\) ;
        end
    end
end


```plaintext
traverse (runs, know, secrets, forbidden)

if any secret in know then
    exit ("attack")
else
    if enabled3(runs, know, forbidden) ≠ ∅ then
        ev = choose(enabled3(runs, know, forbidden))
        if ev = secret(m) then
            traverse(after(runs, ev), know, secrets ∪ {m}, forbidden)
        end
        if ev = send(m) then
            traverse(after(runs, ev), know ⊕ m, secrets, forbidden)
        end
        if ev = read(m) then
            for all m' ∈ match(know, m) ∧ m' ∉ forbidden(read(m)) do
                traverse(after(runs, read(m')), know, secrets, forbidden)
            end
            traverse(runs, know, secrets, forbidden[ev → know])
        end
    end
end
```

\[\text{enabled3}(\text{runs, know, forbidden}) = \{ev ∈ \text{enabled}(\text{runs, know}) \mid ev = \text{read}(m) ⇒ \exists m' ∈ \text{match}(\text{know}, m) m' ∉ \text{forbidden}(ev)\}\]
BKE

\[ SK_i, PK_i, PK_r \]

\[ I \]

nonce \( n_i \)

\[ \{ n_i, I \}_{PK_r} \]

nonce \( n_r \)

key \( k_{ir} \)

\[ \{ h(n_i), n_r, k_{ir} \}_{PK_i} \]

\[ \{ h(n_r) \}_{k_{ir}} \]

secret \( k_{ir} \)

\[ SK_r, PK_r, PK_i \]

\[ R \]
BKE without hash

$SK_i, PK_i, PK_r$

$SK_r, PK_r, PK_i$

nonce $ni$

nonce $nr$

key $kir$

secret $kir$

$\{ni, I\}_{PK_r}$

$\{ni, nr, kir\}_{PK_i}$

$\{nr\}_{kir}$
BKE without $r$

\[ SK_i, PK_i, PK_r \]

\[ SK_r, PK_r, PK_i \]

\[
\begin{align*}
&\text{nonce } n_i \\
&\{n_i, I\}^{PK_r} \\
&\{h(n_i), kir\}^{PK_i} \\
&\{0\}^{kir} \\
&\text{secret } kir
\end{align*}
\]
**BKE** $k_{ir}$ **within encryption**

![Diagram showing the process of BKE with $k_{ir}$](image)
Attack visualization

assumes $e : R$

creates $ni^0$

knows $ne, e, b, a, h, PK, SK(e)$

creates $nr^1, kir^1$

assumes $a : I$

Intruder

Intruder

Intruder

$\{ni^0, a\}_{PK(e)}$

$\{h(ni^0), nr^1, kir^1\}_{PK(a)}$

$\{h(nr^1), kir^1\}_{PK(e)}$

$\{h(nr^1), kir^1\}_{PK(b)}$

$\neg secret[kir^1]$
Tool comparison: number of states

The graph shows the number of states traversed against the number of runs for different tools. The axes are labeled as follows:

- Y-axis: number of states traversed
- X-axis: number of runs

The graph includes three lines:
- TraverseFull
- Brutus
- Traverse

The data points are marked with specific symbols, and the lines connect these points to illustrate the trend as the number of runs increases.
Tool comparison: execution time

![Graph showing execution time comparison for different tools.](image)
Conclusions

- Fastest algorithm that we know of (only basic type flaw attacks).
- Tool produces visual attack trees.
- Possible improvements:
  1. Exploit symmetry in scenario’s.
  2. Combine with Constraint Logic approach ⇒ hybrid model checker.
- Extend algorithm to different intruder models.
- Similar algorithm for authentication properties.