Bab 25

Analisa Protokol Kriptografi

Merancang protokol kriptografi bukan sesuatu yang mudah, bahkan untuk para ahli. Sebagai contoh, Needham-Schroeder *public key protocol* rentan terhadap *man-in-the-middle attack*. Needham-Schroeder *symmetic-key protocol*, protokol yang digunakan oleh Kerberos versi pertama, juga rentan terhadap *replay attack* jika kunci sesi bocor, sehingga perlu dimodifikasi dengan mekanisme *timestamp*. Kelemahan dari suatu protokol kriptografi bisa digolongkan:

- kelemahan algoritma enkripsi/*hashing* atau
- kelemahan logika dari protokol.

Kelemahan algoritma enkripsi/*hashing* dapat dianalisa menggunakan *cryptanalysis*. Analisa protokol kriptografi fokus pada analisa logika dari protokol dengan mengasumsi (untuk sementara) bahwa tidak ada kelemahan dalam algoritma enkripsi/*hashing*.

Analisa protokol kriptografi memerlukan, sebagai dasar, suatu teori logika mengenai berbagai jenis komponen informasi seperti kunci dan data, berbagai operasi terhadap komponen informasi seperti enkripsi data menggunakan kunci, dan informasi apa yang bisa didapat oleh seseorang dari suatu himpunan komponen informasi. Protokol kemudian dirumuskan terdiri dari berbagai operasi dengan urutan tertentu dan melibatkan berbagai aktor dan komponen informasi. Analisa dilakukan untuk melihat efek dari urutan operasi terhadap pengetahuan berbagai aktor mengenai informasi.

Untuk analisa protokol kriptografi, ada dua jenis logika yang kerap digunakan oleh para peneliti, yaitu:

- *modal logic* seperti BAN (Burrows-Abadi-Needham) *logic*, atau

- *classical logic*.

Analisa biasanya dilakukan dengan bantuan alat:

- Untuk *modal logic*, analisa biasanya dibantu dengan *model checker*.
- Untuk *classical logic*, analisa biasanya dibantu dengan *theorem prover*.

Cara *model checker* bekerja adalah dengan mengecek semua kemungkinan *state*, sedangkan *theorem prover* lebih bersifat manipulasi simbol.

Suatu hal yang kurang memuaskan dengan BAN *logic* (lihat [bur90]) adalah *formal semantics* untuk konsep *freshness* yang menjadi bagian dari logika, tidak jelas. Jadi tidak jelas apa arti sesungguhnya dari konsep tersebut. Menggunakan *classical logic*, konsep *freshness* didefinisikan secara langsung, jadi bisa lebih jelas apa yang dimaksud. Oleh sebab itu kita akan fokus pada penggunaan *classical logic* dalam pembahasan lebih lanjut. Penggunaan *classical logic* juga dilakukan oleh Paulson (lihat [pau98]) dan Bolignano (lihat [bol96]).

Pertama, kita akan bahas bagaimana kita dapat membuat suatu teori menggunakan *classical logic* yang meliputi:

- komponen informasi,
- konstruksi menggunakan komponen informasi yang menghasilkan komponen informasi yang lebih besar, dan
- informasi apa yang bisa didapat dari suatu himpunan komponen informasi.

Kita namakan teori tersebut *message theory* dimana suatu *message* merupakan informasi yang terstruktur berdasarkan komponen, atau singkatnya, *message* merupakan komponen informasi. Jadi *message* merupakan apa yang dinamakan *abstract data type*, tepatnya suatu *recursive disjoint union*. Suatu *recursive disjoint union* adalah suatu *type* yang merupakan *union* dari beberapa *subtype* yang *disjoint*. Diantaranya, ada *subtype* yang bersifat *recursive*, yaitu ada nilai *subtype* yang merupakan konstruksi dimana ada subkomponen langsung atau tidak langsung yang mempunyai *subtype* yang sama, bahkan nilai *subtype* bisa mempunyai subkomponen dengan *type message*. Beberapa *subtype* dari *message* bersifat atomik (tidak terdiri dari subkomponen). Ada 6 *subtype* yang bersifat atomik yaitu:

- *text*, untuk naskah,
- *principal*, untuk identitas,
- *nonce*, untuk nilai acak, dimana setiap nilai hanya digunakan sekali dalam aplikasi protokol kriptografi,

- *symmetric key* (*symkey*),
- *private key* (*privkey*), dan
- *public key* (*pubkey*).

Kita definisikan *key* sebagai:

$$key = symkey \cup privkey \cup pubkey$$

dan *atomic* sebagai:

$$atomic = text \cup principal \cup nonce \cup key.$$

Ada 3 *subtype* non-atomik yaitu:

- *encryption*, dengan konstruktor $encrypt(m, k)$, dimana $m \in message$ dan $k \in key$,
- *aggregation*, konstruktornya $combine(m_1, m_2)$, dimana $m_1 \in message$ dan $m_2 \in message$ (karena *combine* bersifat *associative*, kita dapat gunakan $combine(a, b, c)$ untuk $combine(combine(a, b), c)$), dan
- *hashed*, dengan konstruktor $hash(m)$, dimana $m \in message$.

Jadi *message* adalah:

$$message = atomic \cup encryption \cup aggregation \cup hashed.$$

Konstruktor *hash* bersifat *injective*, jadi kita melakukan idealisasi dimana *hash* bersifat *collision-free*. Dalam teori ini terdapat juga fungsi

$$inversekey : key \longrightarrow key$$

dimana

$$\begin{array}{ll} inversekey(k) = k & \text{jika } k \in symkey, \\ inversekey(k) \in pubkey & \text{jika } k \in privkey, \\ inversekey(k) \in privkey & \text{jika } k \in pubkey. \end{array}$$

Lagi kita lakukan idealisasi dengan membuat *inversekey* bersifat *injective*. Konsep penting dalam teori *message* menggunakan fungsi atau predikat *known*:

$$known : message \times messageSet \longrightarrow boolean$$

dimana

$$messageSet = \{m | m \in message\}.$$

Predikat $known(m, s)$ dapat diinterpretasikan sebagai: *message* m dapat diketahui jika setiap *message* dalam s diketahui. Untuk singkatnya kita katakan

bahwa *message* m diketahui dari himpunan s. Ada beberapa *axiom* mengenai *known* dalam teori *message*. *Axiom* pertama adalah mengenai pengetahuan langsung:

$$\begin{aligned} &\forall m \in message, s \in messageSet : \\ &\quad m \in s \Longrightarrow known(m, s) \end{aligned} \tag{25.1}$$

Axiom kedua adalah mengenai *transitivity*:

$$\begin{aligned} &\forall m \in message, s_1, s_2 \in messageSet : \\ &\quad (known(m, s_1) \wedge \forall c \in s_1 : known(c, s_2)) \Longrightarrow known(m, s_2) \end{aligned} \tag{25.2}$$

Axiom 25.2 mengatakan bahwa jika *message* m diketahui dari himpunan s_1 dan setiap elemen dari s_1 diketahui dari himpunan s_2, maka m diketahui dari himpunan s_2. *Axiom* berikut mengatakan *known* bersifat monotonik berdasarkan himpunan.

$$\begin{aligned} &\forall m \in message, s_1, s_2 \in messageSet : \\ &\quad (known(m, s_1) \wedge s_1 \subseteq s_2) \Longrightarrow known(m, s_2) \end{aligned} \tag{25.3}$$

Tiga *axiom* berikutnya mengatakan bahwa *message* yang merupakan hasil konstruksi komponen-komponen yang diketahui juga diketahui.

$$\begin{aligned} &\forall m \in message, k \in key, s \in messageSet : \\ &\quad (known(m, s) \wedge known(k, s)) \Longrightarrow known(encrypt(m, k), s) \end{aligned} \tag{25.4}$$

$$\begin{aligned} &\forall m_1, m_2 \in message, s \in messageSet : \\ &\quad (known(m_1, s) \wedge known(m_2, s)) \Longrightarrow known(combine(m_1, m_2), s) \end{aligned} \tag{25.5}$$

$$\begin{aligned} &\forall m \in message, s \in messageSet : \\ &\quad known(m, s) \Longrightarrow known(hash(m), s) \end{aligned} \tag{25.6}$$

Dua *axiom* berikutnya masing-masing menunjukkan bagaimana caranya untuk mendapatkan komponen *message* dari *encrypt* dan kedua komponen dari *combine*.

$$\begin{aligned} &\forall m \in message, k \in key, s \in messageSet : \\ &\quad (known(encrypt(m, k), s) \wedge known(inversekey(k), s)) \Longrightarrow \\ &\qquad known(m, s) \end{aligned} \tag{25.7}$$

$$\begin{aligned} &\forall m_1, m_2 \in message, s \in messageSet : \\ &\quad known(combine(m_1, m_2), s) \Longrightarrow (known(m_1, s) \wedge known(m_2, s)) \end{aligned} \tag{25.8}$$

Untuk mengetahui komponen *message* dari *encrypt*, *inversekey* dari kunci perlu diketahui. Untuk *combine*, kedua komponen bisa didapat langsung.

Selanjutnya kita definisikan fungsi *parts* yang akan digunakan dalam definisi konsep *fresh*.

$$parts : messageSet \longrightarrow messageSet.$$

Fungsi *parts* menambahkan ke himpunan, semua komponen dari setiap *message* dalam himpunan, semua subkomponen dari setiap komponen, dan seterusnya. *Message* yang atomik tidak mempunyai komponen:

$$\begin{aligned} &\forall m \in atomic, s \in messageSet : \\ &\quad parts(\{m\} \cup s) = \{m\} \cup parts(s). \end{aligned} \tag{25.9}$$

Untuk *encrypt*, *combine* dan *hash*, kita tambahkan komponen ke *parts*.

$$\begin{aligned} &\forall m \in message, k \in key, s \in messageSet : \\ &\quad parts(\{encrypt(m,k)\} \cup s) = \\ &\qquad \{encrypt(m,k)\} \cup parts(\{m,k\} \cup s). \end{aligned} \tag{25.10}$$

$$\begin{aligned} &\forall m_1, m_2 \in message, s \in messageSet : \\ &\quad parts(\{combine(m_1,m_2)\} \cup s) = \\ &\qquad \{combine(m_1,m_2)\} \cup parts(\{m_1,m_2\} \cup s). \end{aligned} \tag{25.11}$$

$$\begin{aligned} &\forall m \in message, s \in messageSet : \\ &\quad parts(\{hash(m)\} \cup s) = \{hash(m)\} \cup parts(\{m\} \cup s). \end{aligned} \tag{25.12}$$

Selesailah pembuatan *message theory*.

Berikutnya, kita kembangkan mekanisme untuk melakukan simulasi protokol berupa *state machine*. *State* dari *state machine* terdiri dari:

- himpunan semua *message* yang telah terlihat, kita namakan *seen*,
- himpunan semua *message* yang telah diterima atau dibuat oleh setiap *principal*, kita namakan *storage*, dan
- sejarah dari *events* yang telah terjadi, kita namakan *history*.

Komponen *seen* merupakan himpunan *message* dan merepresentasikan *medium* komunikasi yang terbuka (dapat disadap dan dapat diinjeksi). Komponen *storage* merupakan kumpulan dari himpunan *message* yang diindeks dengan *principal*. Jadi $storage[a]$ adalah himpunan *message* yang telah dibuat atau diterima oleh *principal* a. Untuk memudahkan analisa, kita melakukan idealisasi dengan mengumpamakan bahwa setiap kunci publik dan setiap *principal* tidak perlu dirahasiakan, tetapi merupakan informasi publik.

$$public = pubKey \cup principal.$$

Kita definisikan konsep $forgeable(p,m)$ (*principal* p dapat membuat *message* m) sebagai berikut:

$$forgeable(p,m) = known(m, storage[p] \cup seen \cup public).$$

Selain *forgeable* kita perlu definisikan konsep *freshness*:

$$fresh(m) = m \notin parts((\bigcup_p storage[p]) \cup seen \cup public).$$

Komponen *history* adalah deretan *event* yang telah terjadi. Suatu langkah protokol adalah suatu *event* yang bisa berupa:

- $send(s, m)$, pengiriman *message* m oleh *principal* s,
- $receive(r, m)$, penerimaan *message* m oleh *principal* r,
- $outOfBand(s, r, m)$, *transfer message* m dari *principal* s ke *principal* r, melalui jalur khusus yang aman,
- $generate(p, m)$, pembuatan *message* m yang *atomic* oleh *principal* p,
- $construct(p, m)$, konstruksi *message* m oleh *principal* p, dan
- $intruder(p, m)$, injeksi *message* m oleh *principal* p.

Suatu simulasi protokol adalah evolusi dari *state machine* dimana setiap langkah mempunyai dua macam persyaratan:

- persyaratan *state machine*, contohnya untuk *send*, *message* yang dikirimkan harus ada dalam *storage principal*, dan
- persyaratan protokol, biasanya berupa persyaratan bahwa *event* dengan atribut tertentu telah ada dalam *history*.

Untuk setiap langkah, *event* untuk langkah tersebut ditambahkan ke deretan *event* dalam *history*. Persyaratan *state machine* dan efek dari suatu langkah terhadap *state* (selain penambahan *event* ke *history*) adalah sebagai berikut:

Langkah	**Persyaratan**	**Efek**
$send(s, m)$	$m \in storage[s]$	$seen \leftarrow seen \cup \{m\}$
$receive(r, m)$	$m \in seen$	$storage[r] \leftarrow storage[r] \cup \{m\}$
$outOfBand(s, r, m)$	$m \in storage[s]$	$storage[r] \leftarrow storage[r] \cup \{m\}$
$generate(p, m)$	$m \in atomic$ $fresh(m)$	$storage[p] \leftarrow storage[p] \cup \{m\}$
$construct(p, m)$	$forgeable(p, m)$	$storage[p] \leftarrow storage[p] \cup \{m\}$
$intruder(p, m)$	$m \in storage[s]$	$seen \leftarrow seen \cup \{m\}$

Beberapa hal yang perlu diperhatikan adalah:

- Langkah *generate* dimaksudkan untuk membuat *nonce*, *text*, *symKey* atau *privKey* (*atomic message* yang bisa dirahasiakan). Karena *pubKey* dan *principal* keduanya tidak *fresh*, *generate* tidak bisa digunakan untuk membuat kunci publik atau *principal*.
- Langkah *intruder* sebetulnya sama dengan langkah *send*. Perbedaan hanya pada penggunaan, langkah *intruder* tidak mempunyai persyaratan protokol, jadi dapat terjadi kapan saja, asalkan persyaratan *state machine* terpenuhi.
- Langkah *construct* dimaksudkan untuk membuat *message* yang akan dikirim menggunakan *send*, *outOfBand* atau *intruder*.
- Untuk keperluan analisa protokol, setiap langkah dapat diberi *label* dan *id* agar persyaratan protokol dapat dirumuskan.

Mari kita gunakan Needham-Schroeder *symmetric key protocol* sebagai contoh untuk melihat bagaimana mekanisme yang telah dibuat dapat digunakan untuk menganalisa protokol. Secara garis besar, menggunakan protokol ini, A ingin berkomunikasi dengan B secara aman dengan bantuan *server* S. Pada awalnya, A, B dan S mengetahui beberapa hal:

- A dan S mengetahui kunci simetris K_{AS}. Selain A dan S, tidak ada yang mengetahui K_{AS}.
- B dan S mengetahui kunci simetris K_{BS}. Selain B dan S, tidak ada yang mengetahui K_{BS}.

Langkah pertama dalam protokol adalah

$$A \longrightarrow S : A, B, N_A$$

dimana A mengidentifikasi dirinya dan B, dan N_A adalah suatu *nonce* yang *fresh*. Langkah kedua adalah

$$S \longrightarrow A : \{N_A, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$$

dimana K_{AB} adalah kunci simetris *fresh* yang dibuat oleh S untuk A dan B. A mendekripsi $\{N_A, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$ untuk mendapatkan K_{AB} dan $\{K_{AB}, A\}_{K_{BS}}$. Langkah ketiga adalah

$$A \longrightarrow B : \{K_{AB}, A\}_{K_{BS}}$$

dimana $\{K_{AB}, A\}_{K_{BS}}$ dapat didekripsi oleh B untuk mendapatkan K_{AB} dan A. Langkah keempat adalah

$$B \longrightarrow A : \{N_B\}_{K_{AB}}$$

dimana N_B adalah *nonce* yang *fresh*. Langkah terahir adalah

$$A \longrightarrow B : \{N_B - 1\}_{K_{AB}}$$

sebagai konfirmasi dari A. Selanjutnya A dan B dapat berkomunikasi menggunakan kunci simetris K_{AB} untuk mengenkripsi komunikasi.

Mari kita coba formalisasikan Needham-Schroeder *symmetric key protocol* menggunakan mekanisme yang telah kita buat. Untuk *initial state*, pengetahuan mengenai kunci dapat kita formalkan sebagai berikut:

$$\begin{array}{l} K_{AS}, K_{BS} \in symKey, \\ K_{AS} \in storage[A], \\ K_{BS} \in storage[B], \\ K_{AS}, K_{BS} \in storage[S], \\ K_{AS}, K_{BS} \notin seen, \\ \forall p \in principal : K_{AS} \in storage[p] \Longrightarrow p = A \vee p = S, \\ \forall p \in principal : K_{AB} \in storage[p] \Longrightarrow p = B \vee p = S. \end{array}$$

Langkah pertama kita bagi menjadi empat langkah: 1_a, 1_b, 1_c dan 1_d. Langkah 1_a adalah

$$generate(a, n_a, 1_a, id)$$

dengan syarat belum ada sesi dengan identifikasi id, dimana n_a adalah *nonce* yang dibuat, 1_a adalah *label*, dan id adalah identifikasi untuk sesi (unik untuk setiap sesi). Langkah 1_b adalah

$$construct(a, combine(a, b, n_a), 1_b, id)$$

dengan persyaratan:

- $generate(a, n_a, 1_a, id)$ ada dalam *history*.

Langkah 1_c adalah

$$send(a, combine(a, b, n_a), 1_c, id)$$

dengan persyaratan:

- $construct(a, combine(a, b, n_a), 1_b, id)$ ada dalam *history*.
- $send(a, combine(a, b, n_a), 1_c, id)$ belum ada dalam *history*.

Langkah 1_d adalah

$$receive(S, combine(a, b, n_a), 1_d, id)$$

dengan persyaratan:

- $send(a, combine(a, b, n_a), 1_c, id)$ ada dalam *history*.
- $receive(S, combine(a, b, n_a), 1_d, id)$ belum ada dalam *history*.

Langkah kedua juga kita bagi menjadi empat langkah: 2_a, 2_b, 2_c dan 2_d. Langkah 2_a adalah

$$generate(S, k_{ab}, 2_a, id)$$

dengan persyaratan:

- $receive(S, combine(a, b, n), 1_c, id)$ ada dalam *history* untuk *nonce* n yang sembarang.
- $generate(S, k, 2_a, id)$ belum ada dalam *history* untuk sembarang kunci k.

Langkah 2_b adalah

$$construct(S, m, 2_b, id)$$

dengan persyaratan:

- $generate(S, k_{ab}, 2_a, id)$ ada dalam *history*.
- n didapat dari $receive(S, combine(a, b, n), 1_c, id)$,

dimana

$$m = encrypt(combine(n, k_{ab}, b, encrypt(combine(k_{ab}, a), K_{bS})), K_{aS}).$$

Langkah 2_c adalah

$$send(S, m, 2_c, id)$$

dengan persyaratan:

- $construct(S, m, 2_b, id)$ ada dalam *history*.
- $send(S, m, 2_c, id)$ belum ada dalam *history*,

dimana

$$m = encrypt(combine(n, k_{ab}, b, encrypt(combine(k_{ab}, a), K_{bS})), K_{aS})$$

Langkah 2_d adalah

$$receive(a, m, 2_d, id)$$

dengan persyaratan:

- $send(S, m, 2_c, id)$ ada dalam *history*.
- $receive(a, m, 2_d, id)$ belum ada dalam *history*,

dimana

$$m = encrypt(combine(n, k_{ab}, b, encrypt(combine(k_{ab}, a), K_{bS})), K_{aS}).$$

Langkah ketiga kita bagi menjadi tiga langkah: 3_a, 3_b dan 3_c. Langkah 3_a adalah

$$construct(a, encrypt(combine(k_{ab}, a), K_{bS}), 3_a, id)$$

dengan persyaratan $receive(a, m, 2_d, id)$ ada dalam *history* dimana

$$m = encrypt(combine(n, k_{ab}, b, encrypt(combine(k_{ab}, a), K_{bS})), K_{aS}).$$

Langkah 3_b adalah

$$send(a, encrypt(combine(k_{ab}, a), K_{bS}), 3_b, id)$$

dengan persyaratan:

- $construct(a, encrypt(combine(k_{ab}, a), K_{bS}), 3_a, id)$ ada dalam *history*.
- $send(a, encrypt(combine(k_{ab}, a), K_{bS}), 3_b, id)$ belum ada dalam *history*.

Langkah 3_c adalah

$$receive(b, encrypt(combine(k_{ab}, a), K_{bS}), 3_c, id)$$

dengan persyaratan:

- $send(a, encrypt(combine(k_{ab}, a), K_{bS}), 3_b, id)$ ada dalam *history*.
- $receive(b, encrypt(combine(k_{ab}, a), K_{bS}), 3_c, id)$ belum ada dalam *history*.

Langkah keempat kita bagi menjadi empat langkah: 4_a, 4_b, 4_c dan 4_d. Langkah 4_a adalah

$$generate(b, n_b, 4_a, id)$$

dengan persyaratan:

- $receive(b, encrypt(combine(k_{ab}, a), K_{bS}), 3_c, id)$ ada dalam *history*.
- $generate(b, n, 4_a, id)$ belum ada dalam *history* untuk sembarang *nonce* n.

Langkah 4_b adalah

$$construct(b, encrypt(n_b, k_{ab}), 4_b, id)$$

dengan persyaratan $generate(b, n_b, 4_a, id)$ ada dalam *history*. Langkah 4_c adalah

$$send(b, encrypt(n_b, k_{ab}), 4_c, id)$$

dengan persyaratan:

- $construct(b, encrypt(n_b, k_{ab}), 4_b, id)$ ada dalam *history*.
- $send(b, encrypt(n_b, k_{ab}), 4_c, id)$ belum ada dalam *history*.

Langkah 4_d adalah

$$receive(a, encrypt(n_b, k_{ab}), 4_d, id)$$

dengan persyaratan:

- $send(b, encrypt(n_b, k_{ab}), 4_c, id)$ ada dalam *history*.
- $receive(a, encrypt(n_b, k_{ab}), 4_d, id)$ belum ada dalam *history*.

Langkah terahir kita bagi menjadi tiga langkah: 5_a, 5_b dan 5_c. Langkah 5_a adalah

$$construct(a, encrypt(combine(n_b, 1), k_{ab}), 5_a, id)$$

dengan persyaratan $receive(a, encrypt(n_b, k_{ab}), 4_d, id)$ ada dalam *history*. Kita gunakan $combine(n_b, 1)$ untuk $n_b - 1$ karena efeknya serupa. Langkah 5_b adalah

$$send(a, encrypt(combine(n_b, 1), k_{ab}), 5_b, id)$$

dengan persyaratan:

- $construct(a, encrypt(combine(n_b, 1), k_{ab}), 5_a, id)$ ada dalam *history*.
- $send(a, encrypt(combine(n_b, 1), k_{ab}), 5_b, id)$ belum ada dalam *history*.

Langkah 5_c adalah

$$receive(b, encrypt(combine(n_b, 1), k_{ab}), 5_c, id)$$

dengan persyaratan:

- $send(a, encrypt(combine(n_b, 1), k_{ab}), 5_b, id)$ ada dalam *history*.
- $receive(b, encrypt(combine(n_b, 1), k_{ab}), 5_c, id)$ belum ada dalam *history*.

Perumusan protokol telah dibuat *general* sehingga a dapat diperankan oleh A atau B, demikian juga dengan b. Jika $a = A$ dan $b = B$ maka $K_{aS} = K_{AS}$ dan $K_{bS} = K_{BS}$. Sebaliknya, jika $a = B$ dan $b = A$ maka $K_{aS} = K_{BS}$ dan $K_{bS} = K_{AS}$. Secara umum, huruf besar digunakan untuk nilai konstan, sedangkan huruf kecil digunakan untuk nilai yang tidak konstan. Huruf k kecil digunakan untuk k_{ab} karena nilai k_{ab} berbeda untuk sesi yang berbeda.

Langkah-langkah protokol yang telah dirumuskan dapat dilakukan kapan saja asalkan persyaratan *state machine* dan persyaratan protokol dipenuhi. Selain langkah protokol, langkah *principal* yang bersifat *intruder* dapat dilakukan kapan saja, asalkan persyaratan *state machine* dipenuhi. *Intruder* dapat melakukan berbagai langkah termasuk:

- langkah *generate*,
- langkah *construct*, dan
- langkah *intruder*.

Bermula dengan *initial state* dimana *history* masih kosong, berbagai langkah protokol dan *intruder* dapat dilakukan untuk membuat suatu evolusi. Contoh dari analisa adalah mencoba membuktikan bahwa suatu rahasia tidak bocor untuk semua kemungkinan evolusi. Satu cara untuk membuktikan bahwa rahasia tidak bocor adalah dengan menggunakan induksi. Untuk setiap langkah protokol dan setiap langkah *intruder* dicoba buktikan bahwa jika rahasia belum bocor sebelum langkah, maka rahasia tetap tidak bocor setelah langkah.

Kita coba buktikan bahwa K_{AS} dan K_{BS} tidak dibocorkan oleh langkah 1_a. Jadi, sebelum langkah, asumsi mengenai kunci pada *initial state* tetap berlaku. Bocor bisa dirumuskan sebagai

$$forgeable(I, K_{AS}) \vee forgeable(I, K_{BS})$$

dimana *principal* I adalah intruder dan $I \neq A$, $I \neq B$ dan $I \neq S$. Mari kita lihat efek dari langkah 1_a:

$$generate(a, n_a, 1_a, id)$$

yaitu

$$storage[a] \leftarrow storage[a] \cup \{n_a\}.$$

Jika $a \neq I$ maka pembuktian mudah karena

$$storage[I] \cup seen \cup public$$

tidak berubah, jadi

$$forgeable(I, m) = known(m, storage[I] \cup seen \cup public)$$

juga tidak berubah untuk semua $m \in message$. Jadi tidak ada *message* baru yang diketahui *intruder* I, dengan kata lain langkah 1_a tidak membocorkan informasi. Jika $a = I$, maka pembuktiannya agak lebih rumit. Pada dasarnya kita harus buktikan

$$\begin{aligned} &\forall k \in key, s \in messageSet, n \in nonce : \\ &\quad \neg known(k, s) \Longrightarrow \neg known(k, s \cup \{n\}). \end{aligned}$$

Kita tidak akan membuktikannya disini. Jelas bahwa untuk membuktikan bahwa protokol tidak membocorkan rahasia, teori perlu diperkuat dengan pembuktian berbagai teorema.

Percobaan analisa Needham-Schroeder *symmetric key protocol* menggunakan mekanisme yang telah kita bangun menunjukkan bahwa pembuktian keamanan protokol secara formal memang tidak mudah. Akan tetapi mekanisme yang telah kita bangun memberi kerangka yang cukup baik, dan apa yang diperlukan secara garis besar telah ditunjukkan. Untuk menyelesaikan pembuktian diperlukan ketekunan yang luar biasa dan kerja keras. Tentunya alat seperti *theorem prover* dapat digunakan untuk membantu pembuktian.

25.1 Ringkasan

Di bab ini kita telah bahas cara melakukan analisa protokol kriptografi. Analisa protokol kriptografi berfokus pada logika dari protokol, bukan kekuatan algoritma enkripsi. Analisa diperlukan karena logika protokol yang bermasalah dapat membocorkan rahasia, meskipun algoritma enkripsi yang digunakan sangat kuat. Ada dua jenis logika yang dapat digunakan sebagai dasar untuk analisa, yaitu *modal logic* dan *classical logic*. Biasanya analisa menggunakan *modal logic* dibantu dengan *model checker*, sedangkan analisa menggunakan *classical logic* dibantu dengan *theorem prover*. Cara analisa yang dikembangkan dan dibahas di bab ini menggunakan *classical logic* sebagai dasar. Analisa Needham-Schroeder *symmetric key protocol* digunakan sebagai contoh.