UNIVERSITÀ CA' FOSCARI – VENEZIA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Scienze dell'Informazione

Tesi di Laurea

Laureando: Francesco Placella

ProtOK: un tool per l'analisi statica di protocolli crittografici

Relatore: Chiar.mo prof. Riccado Focardi
Correlatore: Dott. Matteo Maffei

Anno Accademico 2003-04

Indice

Ringraziamenti

Un ringraziamento particolare va al prof. Umberto Eco, per avermi consentito, grazie al suo libro[1], d'ignorare consapevolmente un buon numero di nozioni utili ai fini della realizzazione di questa tesi.

[1] Umberto Eco, *Come si fa una tesi di laurea*, I ed., Milano, Bompiani, 1977 (XIII ed., Strumenti Bompiani, 1998)

Capitolo 1

Protocolli e crittografia

1.1 Comunicazione

L'argomento fondamentale alla base di questo progetto è la *comunicazione*, in particolare quella che avviene tra due entità alla volta. Saranno presi in esame anche casi di comunicazione tra più di due entità, ma la comunicazione vera e propria avverrà sempre tra due di esse alla volta.

Una possibile definizione di comunicazione è "scambio di messaggi di qualunque natura tra due o più entità", che non fornisce alcuna indicazione sull'effettiva ricezione del contenuto informativo del messaggio, la relativa *comprensione*. È bene infatti tenere distinti i due concetti.

Per evitare pericolosi fraintendimenti sono stati ideati, nel corso della storia della comunicazione, degli strumenti chiamati *protocolli* utili a schematizzare lo scambio di messaggi e a renderne univoco il significato.

L'informatica fa un uso massiccio di protocolli per consentire la comunicazione tra macchine differenti o tra varie componenti della stessa macchina, nell'ambito di questo progetto si farà riferimento al primo scenario, con le limitazioni già menzionate.

1.2 Protocolli e sicurezza

Per garantire la *sicurezza* di determinate comunicazioni vanno utilizzati protocolli che sfruttano particolari accorgimenti per assicurare il rispetto di alcune proprietà a cui gli operatori della comunicazione potrebbero essere interessati: è la combinazione di queste proprietà che definisce in concreto e contestualizza il concetto di sicurezza.

- *autenticità*, il mittente ed il destinatario dei singoli messaggi devono essere correttamente identificabili perché la comunicazione abbia valore;

- *integrità*, dev'essere impossibile modificare il contenuto informativo dei messaggi, o almeno questa anomalia dev'essere rilevabile dal destinatario;
- *segretezza*, il contenuto informativo dei messaggi dev'essere accessibile ai soli operatori della comunicazione;
- *non ripudiabilità*, una volta spedito un messaggio, il mittente non può rifiutarne la paternità;
- *disponiblità*, la capacità comunicativa di un dato operatore non può essere limitata dall'intervento di altri operatori.

Una comunicazione può non richiedere tutte queste proprietà, ma è chiaro che alcune possono implicare le altre: normalmente ha infatti poco senso richiedere, ad esempio, l'autenticità in uno scambio di messaggi senza che ne sia garantita l'integrità. Nella progettazione dei protocolli va quindi considerato l'insieme di proprietà che si desidera proteggere contro la gamma di attacchi che possono essere portati al fine di compromettere la sicurezza della comunicazione.

1.2.1 Possibili attacchi

Considerando lo scenario classico di due operatori onesti A e B che comunicano e che assumono il ruolo di mittente e destinatario rispettivamente (o alternativamente), si possono individuare una serie di attacchi che minano una o più delle proprieta elencate in §.1.2 e che implicitamente definiscono un modello di avversario E in base alle sue capacità:

- *falsificazione*, l'avversario E crea un nuovo messaggio e lo invia al destinatario B spacciandosi per il mittente A;
- *modifica*, l'avversario E altera il contenuto del messaggio iniviato dal mittente A e lo inoltra al destinatario B;
- *intercettazione*, l'avversario E intercetta il messaggio di A destinato a B e ne apprende il contenuto;
- *rifiuto*, E rinnega un messaggio da lui precedentemente inviato a B in quanto potrebbe essere stato ottenuto tramite *falsificazione*;
- *interruzione*, E elimina sistematicamente i messaggi inviati da A (o in arrivo a B).

È chiaro che tutti gli attacchi tranne l'intercettazione configurano un avversario *attivo*, in grado cioè di apportare modifiche al contenuto del canale di comunicazione; al contrario, nel caso dell'intercettazione si ha un avversario *passivo* con capacità di sola lettura nei confronti del contenuto del canale. In ogni caso si ha un canale *non sicuro*.

1.2.2 Il modello di Dolev-Yao

Il modello di Dolev-Yao [DY83] si basa su due assunzioni fondamentali:

- l'avversario è *onnipotente*: è in grado di inserirsi in ogni comunicazione e leggere o modificare il contenuto informativo di qualsiasi messaggio; è inoltre in grado di avviare una comunicazione nei panni di sé stesso o impersonando qualsiasi altro operatore;

- il sistema di crittografia (→ §.1.3) su cui si basano i protocolli utilizzati è considerato *perfetto*: l'avversario sfrutta solo difetti progettuali dei protocolli per compromettere la comunicazione.

La prima assunzione può sembrare irrealistica, ma si è rivelata l'unica in grado di garantire un approccio sicuro alla progettazione di protocolli.

1.3 Crittografia

Fin da quando si è sviluppata una sensibilità nei confronti di queste problematiche, la soluzione più affidabile è stata individuata nella *crittografia*, ovvero una trasformazione reversibile del contenuto informativo del messaggio, tale da renderlo completamente inaccessibile a chi non conosca la trasformata inversa. Le due trasformazioni prendono rispettivamente il nome di *cifratura* e *decifratura*.

Storicamente sono due gli approcci agli algoritmi di cifratura (e decifratura): algoritmo *segreto*, e algoritmo pubblico *parametrico*. Il primo approccio ha lo svantaggio enorme che, se per caso viene compromesso l'algoritmo, anche l'intero sistema basato su di esso e tutte le comunicazioni passate possono essere tranquillamente considerati compromessi; il secondo approccio ha il duplice vantaggio di essere sottoposto a maggiori controlli e verifiche di robustezza e di essere vulnerabile solo nel parametro (la chiave), che può essere tranquillamente sostituito se compromesso. Tutti i maggiori sistemi di crittografia attuali sfruttano il secondo approccio.

Quest'ultimo introduce il problema del *key management*, la gestione delle chiavi (illustrata più dettagliatamente in §.1.4.2): chi è autorizzato a conoscerle? Anche in questo caso sono due gli approcci individuati:

- *crittografia a chiave simmetrica*, in cui gli operatori A e B condividono una chiave k_{AB} nota soltanto a loro:

$$x = D_{k_{AB}}(E_{k_{AB}}(x))$$

- *crittografia a chiave pubblica*, in cui un operatore A possiede una coppia di chiavi (pk_A, sk_A), pubblica e privata: la seconda, conosciuta solo da A, permette di decifrare tutto ciò che è cifrato con la prima, che dev'essere nota a chiunque voglia comunicare con A, cioè:

$$x = D_{sk_A}(E_{pk_A}(x))$$

normalmente non è richiesto l'inverso ($\rightarrow$ §.1.3.3, *chiave pubblica*):

$$x = D_{pk_A}(E_{sk_A}(x))$$

1.3.1 Tipi di cifrari

Storicamente i *cifrari* (algoritmi di cifratura) possono essere classificati in due categorie fondamentali: *block cipher* e *stream cipher*; in entrambi i casi il testo del messaggio è suddiviso in blocchi, normalmente di uguale lunghezza, ma mentre i primi cifrano ognuno con la stessa chiave, i secondi a partire dalla chiave generano un *flusso* di chiavi differenti con cui cifrano i rispettivi blocchi.

Cifrari a blocchi

Gli esempi più semplici di cifrari di questo tipo sono quelli *a spostamento*, in cui, dato un alfabeto di lunghezza d, ogni lettera viene spostata di k posizioni:

$$E_k(x) = x + k \pmod d$$
$$D_k(y) = y - k \pmod d$$

e quelli *a sostituzione*, in cui la chiave è costituita da una permutazione π dell'alfabeto originale:

$$E_k(x) = \pi(x)$$
$$D_k(y) = \pi^{-1}(y)$$

in entrambi i casi ad ogni lettera in chiaro corrisponde sempre la stessa lettera cifrata (cifrari *monoalfabetici*), il che li rende facilmente attaccabili.

I cifrari *polialfabetici* sono invece caratterizzati dal far corrispondere ad ogni lettera in chiaro lettere di volta in volta differenti. Spesso sono evoluzioni di quelli descritti in precedenza, come nel caso del cifrario *di Vigenére*, in cui si utilizza una chiave k di lunghezza n per determinare la lunghezza del blocco; ogni lettera k_i della chiave determina uno *spostamento* della corrispondente lettera del blocco in chiaro x:

$$k = k_1 k_2 \cdots k_n$$

$$E_k(x) = E_k(x_1, x_2, \ldots, x_n) = x_1 + k_1, x_2 + k_2, \ldots, x_n + k_n \pmod d$$
$$D_k(y) = D_k(y_1, y_2, \ldots, y_n) = y_1 - k_1, y_2 - k_2, \ldots, y_n - k_n \pmod d$$

risulta chiaro che la stessa lettera potenzialmente è cifrata ogni volta in modo differente.

Cifrari a flusso
La caratteristca principale di questi cifrari è l'utilizzo di una chiave differente per ogni blocco di testo in chiaro; il flusso di chiavi è generato tramite una funzione che dipende dalla chiave k: un fusso *sincrono* genera tutte le chiavi in modo indipendente dal testo in chiaro X, mentre in uno *asincrono* la funzione generatrice dipende anche dal blocco di testo in chiaro x_i appena cifrato:

$$z_i = f(z_{i-1}), \text{ con } z_0 = k, \text{ flusso sincrono}$$

$$z_i = f(x_{i-1}), \text{ con } x_0 = k, \text{ flusso asincrono}$$

$$Y = y_1 y_2 \cdots y_n = E_{z_1}(x_1) E_{z_2}(x_2) \cdots E_{z_n}(x_n)$$

Il vantaggio del primo approccio sta nella possibilità di generare il flusso di chiavi senza bisogno del testo in chiaro, quindi l'eventuale perdita di un blocco non influenzerebbe la capacità di decifrare quelli successivi. Al contrario la modalità asincrona obbliga a generare le chiavi un blocco alla volta, in fase di decifratura; consente però al contempo di distribuire l'informazione di ogni blocco su tutti i blocchi successivi: questo costituisce un problema in caso di perdita di un blocco, ma è utile per generare un *Message Authentication Code (MAC)*, ovvero un'informazione altamente ridondante che, allegata al messaggio originale, permette di verificarne l'integerità.

Cifrari composti
Il grande limite degli esempi citati sinora è la facile attaccabilità. Un metodo molto efficace per aumentare l'efficienza dei cifrari è la *composizione*; ovviamente, dati due cifrari S_1 e S_2, il codominio del primo dev'essere un sottoinsieme del dominio del secondo, $C_1 \subseteq D_2$, allora si avrà $S_1 \times S_2$:

$$E_k(x) = E_{(k_1,k_2)}(x) = E^2_{k_2}(E^1_{k_1}(x))$$

$$D_k(y) = D_{(k_1,k_2)}(y) = D^1_{k_1}(D^2_{k_2}(x))$$

con $k \in K = K_1 \times K_2$.

Occorre precisare che i due cifrari non devono necessariamente essere differenti, ma in questo caso non devono essere *idempotenti*, cioè posto $S = S_1 = S_2$ deve valere la proposizione:

$$S \neq S \times S$$

in pratica ogni iterazione della cifratura deve fornire un risultato differente.

Bisogna inoltre fare attenzione nella composizione di cifrari idempotenti, perché in caso *commutino*, se vale cioè:

$$S_1 \times S_2 = S_2 \times S_1$$

allora anche il cifrario composto è idempotente.

1.3.2 Caratteristiche di sicurezza e cifrari

Prendendo in considerazione alcune delle proprietà elencate in §.1.2, si possono tracciare le caratteristiche che deve vantare un cifrario per raggiungere lo scopo prefisso.

Integrità
Un cifrario può essere progettato in modo tale che una modifica localizzata dell'informazione cifrata si ripercuota su tutta o gran parte dell'informazione decifrata, in questo modo la modifica è immediatamente rilevabile da parte del destinatario, a meno che il contenuto non abbia una struttura arbitraria, il che può accadere; è quindi buona norma aggiungere dell'informazione ridondante nel messaggio per ovviare a questo inconveniente (→ §.1.3.1, *MAC*).

Segretezza
Ovviamente un cifario deve rendere inaccessibile il contenuto informativo del messaggio originale, altrimenti verrà a mancare uno dei requisiti fondamentali per cui i cifrari sono stati ideati: un cifrario che inverte la stringa corrispondente all'informazione originale avrà ben poca fortuna di questi tempi.

Quanto è stato introdotto sopra in modo piuttosto intuitivo è trattato in modo esaustivo dalla *teoria di Shannon*, brevemente descritta in §.1.3.4. Le altre proprietà vanno affrontate a livello più alto, a prescindere dalle caratteristiche del sistema di cifratura.

Autenticazione
È in genere ottenuta tramite la dimostrazione da parte del *claimant*, l'entità che vuole provare la propria identità, della conoscenza di un segreto che è sua prerogativa.

Non ripudiabilità
La soluzione adottata universalmente in questo ambito è la *firma digitale* (discussa in maggior dettaglio in §.1.3.3), un meccanismo crittografico che associa indissolubilmente il proprietario della firma al messaggio firmato.

Disponibilità
Questo problema va affrontato in maniera più drastica, limitando praticamente le capacità dell'avversario o aumentando le risorse a disposzione dell'operatore.

1.3.3 Firma digitale

La firma digitale è stata pensata per costituire l'equivalente digitale di una firma: questa ha delle proprietà e ne conferisce altre ai documenti su cui è apposta. Le peculiarità principali che non possono essere replicate banalmente in ambito digitale, sono fondamentalmente due:

- una firma è *apposta* su un documento, è quindi indissolublimente legata ad esso;
- una firma *non è replicabile* da chiunque non ne sia il proprietario: esistono verifiche che permettono di stabilire con estrema precisione se la firma sia o meno autentica; la firma ed il suo proprietario sono un binomio inscindibile.

Uno *schema di firma* per affrontare e risolvere questi problemi, deve presentare caratteristiche precise:

- $Sign_k(X)$, un meccanismo di firma;
- $Ver_k(X, Y)$, un meccanismo pubblico per verificare che Y sia effettivamente la firma di X;
- k dev'essere un segreto prerogativa del firmatario;
- Y può essere ottenuto da X solo conoscendo k.

Firma digitale tramite crittografia a chiave pubblica
Nel caso in cui per lo schema a chiave pubblica utilizzato valga la relazione

$$X = D_{pk_A}(E_{sk_A}(X))$$

si dispone automaticamente di uno schema di firma digitale; difatti chiunque possieda la chiave pubblica pk_A può verificare che dalla firma Y si ottiene il messaggio X:

$$X = D_{pk_A}(Y)$$

il che implica la cifratura di X tramite sk_A, segreto appartenente ad A. Ovviamente lo schema dev'essere sufficientemente robusto da soddisfare le altre richieste, ma normalmente se lo schema è utilizzabile come cifrario è altrettanto efficiente come schema di firma.

Un problema indipendente dall'implementazione dello schema di firma è costituito dalla dimensione del messaggio: da quanto detto finora, è facile ipotizzare che questa coincida la dimensione della firma, difatti quest'ultima deve contenere un'informazione pari a quella del messaggio, per potervi essere legata. La soluzione adottata in questi casi è quella delle *funzioni hash*;

si tratta di funzioni *one-way* (non invertibili) che permettono di creare un *riassunto* di lunghezza arbitraria e che, per lunghezze ragionevoli, riducono quasi a 0 la probabilità di collisione[1]. In questo modo è possibile firmare soltanto il riassunto del messaggio ed ottenere un'affidabilità del sistema peressoché identica:

$$M = \{X, Sign_k(h(X))\}$$

I due algoritmi di hashing più utilizzati correntemente sono:

- *MD5*, Rivest, 1995
- *Secure Hash Algoritm, (SHA)*, National Institute for Standard & Technology, 1993

1.3.4 La teoria di Shannon

La teoria elaborata da Shannon nel 1949, permette di valutare la sicurezza di un cifrario, cioè misurare e classificare la sua *attaccabilità*, tramite delle precise categorie.

- *sicurezza incondizionata*, stabilisce che un cifrario è inattaccabile pur avendo a disposizione una capacità di calcolo illimitata; un cifrario incondizionatamente sicuro è chiamato *cifrario perfetto*, poiché è caratterizzato dal fatto che l'incertezza che si ha sull'informazione in chiaro non è minimamente influenzata dallo studio dell'informazione cifrata; questa è una classificazione teorica perché qualsiasi cifrario è attaccabile per *forza bruta* (provando tutte le cifrature possibili), se si dispone di illimitata capacità di calcolo[2];
- *sicurezza computazionale*, un approccio più realistico: si determinano le richieste in termini di capacità di calcolo dell'algoritmo, assumendo che quella dell'avversario sia *polinomiale*, e si verifica che l'attacco per forza bruta sia irrealizzabile; alternativamente è possibilie dimostrare che rompere l'algoritmo equivale a risolvere un problema notoriamente non polinomiale; la "perfezione" del cifrario rimane comunque una caratteristica altamente desiderabile.

Modello probabilistico

Per prima cosa Shannon distingue i seguenti elementi: l'insieme dei testi in chiaro P, l'insieme dei testi cifrati C, l'insieme delle chiavi K; definisce inoltre le funzioni di cifratura e decifratura E_k e D_k; $p_P(x)$ è la probabilità di un testo $x \in P$; $p_K(k)$ è la probabilità di una chiave $k \in K$.

[1] Trovare o costruire due testi che abbiano lo stesso hash.

[2] Fortunatamente anche questa è un'ipotesi puramente teorica.

Definisce quindi l'insieme dei testi cifrati ottenibili con k

$$C(k) = \{E_k(x)|x \in P\}$$

la probabilità di un testo cifrato $y \in C$

$$p_C(y) = \sum_{k \in K|y \in C(k)} p_K(k) p_P(D_k(y)), \text{ con } x = D_k(y)$$

e ed infine quelle complementari

$$p_C(y|x) = \sum_{k \in K, x = D_k(y))} p_K(k) \qquad p_P(x|y) = \frac{p_C(y|x) p_P(x)}{p_C(y)}$$

a questo punto è possibile enunciare i risultati principali[3]:

Definizione 1.3.1 *Un cifrario si dice* perfetto *o* incondizionatamente stabile, *se l'incertezza sul testo in chiaro non è influenzata dall'osservazione del testo cifrato, ovvero se:*

$$p_P(x|y) = p_P(x)$$

Teorema 1.3.1 *In un cifrario perfetto la cardinalità dell'insieme delle chiavi è maggiore di quella dell'insieme dei testi in chiaro ed in particolare si ha*

$$|K| \geq |C| \geq |P|$$

Teorema 1.3.2 *Dato un cifrario tale che:*

$$|K| = |C| = |P|$$

allora il cifrario è perfetto se e solo se:

1. *per ogni coppia (x,y) esiste una sola chiave tale che $E_k(x) = y$;*

2. *la probabilità della chiave k è*

$$p_K(k) = \frac{1}{|K|} \qquad \forall k \in K$$

1.3.5 Cenni di crittanalisi

Con *crittanalisi* s'intende l'insieme delle tecniche utilizzabili per aggirare un sistema di cifratura; gli scopi della crittanalisi spaziano dal recuperare il testo in chiaro a partire da quello cifrato, passando per l'individuazione di una chiave segreta, fino alla scoperta di una vera e propria falla nel cifrario preso in esame, ovvero l'individuazione di un "controalgoritmo" che renda il primo inutilizzabile. Un crittanalista ha a disposizione quattro scenari principali:

[3] Per i dettagli si veda [MOV97].

- *ciphertext*, è disponibile il solo crittogramma;

- *known plaintext*, sono disponibili una o più coppie di testo in chiaro M e del corrispondente testo cifrato C;

- *chosen plaintext*, è disponibile un meccanismo per ottenere la cifratura C di un testo in chiaro a scelta M; la chiave segreta corrente k rimane ignota;

- *chosen ciphertext*, è disponibile un meccanismo di decifratura che fornisce il testo in chiaro M a partire dal quello cifrato C; la chiave k rimane ignota.

Un tipo di crittanalisi che si è rivelata molto efficace con cifrari non molto sofisticati (→ §.1.3.1), è quella *probabilistica*: si sfruttano le statistiche basate sulle caratteristiche morfologiche delle parole appartenenti alla lingua del messaggio (bisogna conoscerla), per risalire alla mappatura da testo in chiaro a testo cifrato ed ottenere la chiave.

Per molti cifrari più elaborati sono stati proposti degli attacchi ad hoc; quelli che sono usciti indenni da tutti i tentativi di rottura, sono tuttora in uso.

1.3.6 Principali cifrari in uso

I sistemi di crittografia più importanti al giorno d'oggi sono tre: i primi due si basano sullo schema a chiave simmetrica, il terzo è lo schema a chiave pubblica per eccellenza.

DES

Il *Data Encryption Standard* è il più popolare tra i cifrari a chiave condivisa; è stato sviluppato nel 1975 da IBM per conto dell'esercito americano e nella sua versione originale è ormai attaccabile per forza bruta; tuttavia non è mai stato trovato un attacco alternativo che abbia avuto successo.

Il DES cifra blocchi di stringhe di 64 bit utilizzando una chiave di 56 bit; sfrutta la composizione e l'iterazione di varie sottocomponenti per ottenere un output dall'apparenza completamente casuale, la cui dipendenza dall'input è complicatissima da rilevare. Dispone inoltre di varie modalità di utilizzo che gli consentono di operare sia come block cipher che come stream cipher, potendo così adattarsi alle più disparate esigenze di prestazioni, affidabilità e integrità.

La versione attuale è il *Triple-DES* (DES3), una composizione tripla dello schema originale (non idempotente), per la precisione si ha:

$$Y = E_k(X) = E_{k_1}(D_{k_2}(E_{k_3}(X))) \text{ con } k \in K = (K_1 \times K_2 \times K_3)$$

ne risulta una chiave k di 168 bit; inoltre ponendo $k_1 = k_2 = k_3$, si ottiene lo schema originale, garantendo così la compatibilità nei confronti delle precedenti versioni. Il nuovo schema così ottenuto richiede una capacità quantificabile in $O(2^{112})$, in termini di spazio o di tempo: l'attacco per forza bruta non è attuabile (al momento).

IDEA
L'*International Data Encryption Algorithm* è stato presentato nel 1990 come alternativa a DES; cifra anch'esso blocchi di 64 bit, ma è caratterizzato da una chiave di 128 bit. Anche IDEA può vantare una notevole capacità di confondere la dipendenza dell'output dall'input: ogni singolo bit di testo in chiaro e chiave influiscono sul valore di tutti i bit del crittogramma, in modo da nascondere la struttura statistica del testo.

RSA
Come detto, l'RSA è *lo* schema a chiave pubblica: nasce nel 1977 ad opera di Rivest, Shamir e Adleman come risposta ad una serie di linee guida proposte da Diffie e Hellman nel 1976. L'algoritmo fondamentale consiste in un elevamento a potenza modulo n del testo in chiaro x, con $x < n$; valgono le seguenti relazioni:

$n = pq$ con p, q primi

$ab \equiv 1 \pmod{\phi(n)}$

$pk = (n, b) \quad sk = (p, q, a)$

con ϕ funzione di Eulero[4]; le funzioni di cifratura sono:

$$E_{pk}(x) = x^b \pmod{n} \quad \text{e} \quad D_{sk}(y) = y^a \pmod{n}$$

RSA è sicuro perché l'unico modo di risalire ad a a partire da b è calcolare $\phi(n)$, ma l'unico modo efficiente per farlo è conoscere p e q, dato che vale

$$\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$$

ciò significa che per rompere RSA, bisogna fattorizzare n che, per valori sull'ordine di 1024 bit o superiori, è infattibile.

A questo punto ci potrebbe domandare quale sia la necessità di utilizzare due schemi di cifratura così differenti. Il motivo fondamentale è che i cifrari a chiave simmetrica sono molto veloci ed efficienti, anche su input di grandi dimensioni, ma comportano una gestione della chiavi decisamente complicata,

[4] $\phi(n)$ è il numero di $x < n$ tali che $MCD(x, n) = 1$; se n è primo, ovviamente $\phi(n) = n - 1$, altrimenti non esiste un metodo efficiente per calcolare ϕ.

come si vedrà in §.1.4.2; al contrario gli schemi a chiave pubblica sono molto impegnativi dal punto di vista delle risorse, vantano però una gestione della chiave più vantaggiosa e forniscono un meccanismo per la firma digitale.

Normalmente i due schemi sono utilizzati in combinazione: si sfrutta quello a chiave pubblica per negoziare una chiave di *sessione* condivisa tra i due operatori della comunicazione; da quel momento in poi si usa lo schema a chiave simmetrica.

1.4 Protocolli crittografici

Un *protocollo crittografico*, come è facile intuire, non è altro che un normale protocollo che si basa su un sistema di crittografia per ottenere una qualche forma di sicurezza (→ §.1.2). Le questioni principali che si trovano ad affrontare i protocolli crittografici riguardano la gestione delle chiavi (descritta in §.1.4.2) e l'autenticazione (descritta in §.1.4.4).

Considerando la situazione modellata da Dolev-Yao (→ §.1.2.2), l'avversario E può sfruttare gli eventuali difetti nella logica dei protocolli per apprendere il contenuto dei messaggi scambiati durante la comunicazione, nonostante questi possano essere cifrati, o comunque compromettere una o più delle caratteristiche descritte in §.1.2.

Sebbene esistano attacchi ad hoc che sfruttano debolezze particolari di certi protocolli, si possono comunque individuare alcuni tipi d'attacco più generalizzati:

- *known-key*, in cui si sfrutta la conoscenza della chiave utilizzata nella comunicazione; come si è già accennato, normalmente i protocolli crittografici utilizzano *chiavi di sessione* (a breve termine): se l'avversario in qualche modo riesce ad apprenderla, può avviare una nuova sessione di comunicazione, riciclando un vecchio messaggio d'avvio, e impersonare uno dei due operatori:

  1. $E(A) \rightarrow B: \ \{start(k_s)\}_{k_A B}$
  2. $B \rightarrow E(A): \ \{start(k_s)\}_{k_A B}$
  3. $E(A) \rightarrow B: \ \{\ldots\}_{k_s}$
  4. $B \rightarrow E(A): \ \{\ldots\}_{k_s}$

- *free-decryption*, in cui l'avversario può sfruttare delle ambiguità nel protocollo per farsi decifrare uno o più messaggi dagli stessi operatori della comunicazione:

  1. $E(A) \rightarrow B: \ \{msg\}_{k_{AB}}$
  2. $B \rightarrow E(A): \ msg$

- *man-in-the-middle*, in cui l'avversario si frappone tra gli operatori all'avvio della comunicazione, negoziandone in prima persona, pur senza rivelarsi, eventuali parametri (la chiave di sessione per esempio) e seguendone tutto lo svolgimento: gli operatori sono convinti di aver comunicato segretamente, mentre tutto il contenuto dei messaggi è stato intercettato:

  1. $A \to E(B): \ start(k_A) \quad E(A) \to B: \ start(k_A)$
  2. $B \to E(A): \ start(k_B) \quad E(B) \to A: \ start(k_B)$
  3. $A, B, E: k_s = k_A \oplus k_B$
  4. $A \to E(B): \ \{\ldots\}_{k_s} \quad E(A) \to B: \ \{\ldots\}_{k_s}$

- *reflection*, in cui l'avversario può mandare indietro un messaggio appena inviato, spacciandolo come risposta al precedente:

  1. $A \to E(B): \ \{start(k_s)\}_{k_{AB}}$
  2. $E(B) \to A: \ \{start(k_s)\}_{k_{AB}}$
  3. $A \to E(B): \ \{\ldots\}_{k_s}$

1.4.1 Evitare il riutilizzo dei messaggi

Una delle proprietà più importanti per un messaggio di un protocollo crittografico è la *freshness*: va evitato in ogni modo che l'avversario ricicli messaggi da sessioni precedenti [5], bisogna quindi aggiungere delle informazioni legate al tempo. Le soluzioni principali sono tre:

- *numero di sequenza*, che è incrementato ad ogni esecuzione del protocollo:

  1. $A \to B: \ \{\ldots, seq_A\}_k$

  B deve verificare che seq_A sia maggiore dell'ultimo numero di sequenza memorizzato; ovviamente ogni operatore deve mantenere un numero di sequenza per ogni destinatario ed uno per ogni mittente, inoltre dev'essere predisposto un meccanismo *sicuro* per azzerare il contatore una volta raggiunto il valore massimo;

- *timestamp*, che è ricavato dall'orologio della macchina al momento di comporre il messaggio:

  1. $A \to B: \ \{\ldots, t_A\}_k$

[5] Questo accorgimento evita fondamentalmente attacchi di tipo known-key.

B deve verificare che il timestamp t_A ricada all'interno di un intervallo di validità prestabilito, calcolato in base a parametri quali il tempo medio di trasmissione ed elaborazione del messaggio ed il ritardo media tra gli orologi degli operatori; deve inoltre verificare di non aver già ricevuto quel timestamp, perché in quel caso si tratterebbe sicuramente di un duplicato; questo meccanismo presuppone un sistema *sicuro* di sincronizzazione degli orologi, che non è un requisito banale;

- *nonce*[6], che è il più semplice da implementare, ma che richiede un incremento del numero dei messaggi:

  1. $B \rightarrow A :\ N_B$
  2. $A \rightarrow B :\ \{\ldots, N_B\}_k$

  B deve verificare che il nonce cifrato corrisponda a quello da lui inviato in precedenza.

1.4.2 Gestione delle chiavi

Come si è già avuto modo di accennare, uno dei problemi principali affrontati dai protocolli crittografici è la *gestione delle chiavi*: per stabilire una comunicazione sicura due operatori devono condividere una chiave simmetrica, in quanto sarebbe troppo oneroso cifrare l'intero scambio con uno schema a chiave pubblica. Tuttavia, anche nel caso in cui siano dotati di una chiave *a lungo termine*[7], è bene che per cifrare la comunicazione si servano di una chiave *a breve termine*: in questo modo riducono drasticamente il numero di messaggi che un avversario può crittanalizzare per tentare di scoprire le chiavi segrete.

Un avversario può sempre riuscire a scoprire la chiave di sessione, ma questo normalmente richiede un tempo superiore alla durata della comunicazione, la fine della quale generalmente segna lo scadere della validità della chiave. L'avversario può apprendere il contenuto della comunicazione passata, se ha avuto l'accortezza di registrarla, ma questo non compromette quelle future, posto che il protocollo sia progettato correttamente.

A quanto detto sopra va aggiunto che ogni coppia che comunichi tramite chiavi a lungo termine deve condividerne una: in un ambiente che vanti un alto numero di operatori il numero di chiavi necessarie sale molto rapidamente:

$$n_k = \frac{n_{op}(n_{op} - 1)}{2}$$

n_k cresce quadraticamente rispetto al numero di operatori.

[6] Sta per *number used only once.*

[7] Una chiave, condivisa tra i due operatori in via definitiva, che non viene cambiata a meno che non sia compromessa.

Trusted Third Party

La soluzione è chiamare in causa un *trusted third party*[8] (TTP), che funga da accentratore: chiunque voglia comunicare avvisa il TTP che si mette in contatto col destinatario. Esistono due approcci possibili:

- *Key Distribution Center (KDC)*, che genera le chiavi di sessione e le distribuisce alle controparti;
- *Key Translation Center (KTC)*, che si limita ad inoltrare al destinatario la chiave di sessione generata dal mittente.

Ovviamente il TTP dev'essere in grado di comunicare con ogni altro operatore, quindi deve contare una chiave di sessione per ognuno[9]. Questo si può realizzare in due modi:

- mantenendo una tabella in cui ad ogni operatore si associa la rispettiva chiave, approccio che potrebbe essere dispendioso, nonché risultare disastroso in caso di compromissione della tabella;
- utilizzando dei certificati: il TTP cifra con una sua chiave privata k_T l'associazione identità-chiave:

$$Cert_A = E_{k_T}(A, k_A)$$
$$Cert_B = E_{k_T}(B, k_B)$$
$$\dots$$

  a questo punto può decidere se mantenere i certificati localmente, occupando risorse, oppure distribuirli ai vari operatori, esponendoli agli attacchi dell'avversario, per poi verificarne il contenuto quando necessario:

$$A \to T: \ Cert_A, Cert_B$$

Anche nel caso in cui si utilizzi uno schema a chiave pubblica è necessario l'intervento del TTP, infatti, pur non sussistendo il problema del numero di chiavi[10], va in qualche modo garantito il legame tra l'operatore e la rispettiva chiave pubblica: se anche il TTP dispone di una coppia (pk_T, sk_T), può generare le coppie (pk, sk) degli operatori, distribuire le chiavi segrete attraverso un *canale sicuro*[11] e mettere a disposizione di tutti dei certificati del tipo:

$$Cert_A = E_{sk_T}(A, pk_A)$$

---

[8] Una terza parte fidata.

[9] In questo modo il numero di chiavi necessarie coincide con quello degli operatori: $n_k = n_{op}$.

[10] È necessaria solo una coppia (pk, sk) per ogni operatore: $n_{kc} = n_{op}$

[11] L'informazione non deve transitare in alcun canale di comunicazione accessibile all'avversario.

in questo modo chiunque conosca pk_T[12], è in grado di decifrare $Cert_A$, verificare l'autenticità del legame (A, pk_A) e regolarsi di conseguenza.

1.4.3 Principali protocolli di key-distribution

Di seguito sono presentati alcuni esempi di protocolli effettivamente in uso; il primo è a chiave condivisa, il secondo sfrutta un TTP, mentre il terzo utilizza uno schema a chiave pubblica.

Authenticate Key-Exchange Protocol-2 (AKEP-2)
Questo protocollo fa uso di un hash function parametrica h (→ §.1.3.3) e di un MAC (→ §.1.3.1), e presuppone la presenza di una chiave condivisa k:

1. $A \rightarrow B:\ R_A$
2. $B \rightarrow A:\ B, A, R_A, R_B, MAC_k(B, A, R_A, R_B)$
3. $A \rightarrow B:\ A, R_B, MAC_k(A, R_B)$
4. $A, B : k_s = h_k(R_B)$

l'utilizzo del MAC garantisce l'integrità, quello dei nonce R la freshness.

Kerberos
Kerberos è in realtà una suite di protocolli, tra cui il seguente; fa uso di un KDC e di timestamp per garantire la freshness:

1. $A \rightarrow T:\ A, B, N_A$
2. $T \rightarrow A:\ \{k_s, A, L\}_{k_B}, \{k_s, N_A, A, L\}_{k_A}$
3. $A \rightarrow B:\ \{k_s, A, L\}_{k_B}, \{A, t_A\}_{k_s}$
4. $B \rightarrow A:\ \{A, t_A\}_{k_s}$

$\{k_s, A, L\}_{k_B}$ prende il nome di *ticket*, mentre $\{A, t_A\}_{k_s}$ quello di *authenticator*. Il parametro L indica la durata della validità della chiave k_s: se A desidera avviare una nuova sessione del protocollo e la chiave è ancora valida, può evitare di contattare nuovamente T, aggiornando il timestamp:

$$A \rightarrow B:\ \{k_s, A, L\}_{k_B}, \{A, t'_A\}_{k_s}$$

[12] Può essere pubblicata attraverso qualche canale affidabile.

X.509 strong three-way authentication
In questo caso vengono utilizzati nonce per garantire la freshness e firma digitale quanto riguarda per l'autenticazione:

1. $A \rightarrow B: \ N_A$
2. $B \rightarrow A: \ \{B, k_B\}_{pk_A}, Sign_B(N_B, N_A, A, \{B, k_B\}_{pk_A})$
3. $A \rightarrow B: \ \{A, k_A\}_{pk_B}, Sign_A(N_A, N_B, B, \{A, k_A\}_{pk_B})$
4. $A, B : k_s = k_A \oplus k_B$

Da questi esempi risulta chiaro che la negoziazione di chiavi di sessione, implica l'autenticazione mutua delle parti.

1.4.4 Autenticazione

I protocolli di autenticazione hanno come obbiettivo la corretta *identificazione* degli operatori della comunicazione: a seconda del protocollo e delle necessità questa può risultare unidirezionale o mutua.

L'entità che vuole autenticarsi prende il nome di *claimant*, mentre quella presso cui il claimant si vuole autenticare prende il nome di *verifier*.

La dicitura *corretta* identificazione implica due aspetti fondamentali:

- *fairness*, il protocollo deve consentire l'autenticazione di ogni operatore onesto certificando la sua identità:
  1. $A \rightarrow B: \ claim(A)$
  2. $B \rightarrow A: \ ok(A), claim(B)$
  3. $A \rightarrow B: \ ok(B)$
- *non impersonation*, non dev'essere possibile riciclare un'autenticazione di un operatore per autenticarsi presso terzi:
  1. $A \rightarrow E: \ claim(A)$
  2. $E \rightarrow B: \ claim(A)$
  3. $B \rightarrow E: \ ok(A)$
  4. $E \rightarrow A: \ ok(A)$

Il sistema più comune per provare la propria identità in questo ambito, è la dimostrazione della conoscenza di un'informazione segreta; è fondamentale che però questa dimostrazione avvenga senza che il segreto venga compromosse, altrimenti l'intero processo perderebbe il suo valore.

L'approccio più indicato per ottenere questo risultato è il *challenge-response*: il verifier propone una sfida con un parametro variabile di volta in volta, il claimant fornisce una risposta che dipende sia dalla sfida che dal segreto.

Le soluzioni adottate più comunemente sono quelle basata su timestamp, in cui la sifda implicita consiste nel cifrare un timstamp valido:

1. $A \to B: \ \{t_A, B\}_{k_{AB}}$
2. $B \to A: \ \{t_A, A\}_{k_{AB}}$

l'indificatore B serve ad evitare attacchi di tipo reflection:

1. $A \to E(B): \ \{t_A\}_{k_{AB}}$
2. $E(B) \to A: \ \{t_A\}_{k_{AB}}$

in questo modo B impersonerebbe E con A; al termine del protocollo, si ottiene la mutua autenticazione.

L'alternativa è l'autenticazione basata su nonce; la sfida, in questo caso esplicita, consiste nel cifrare un nonce con la chiave segreta:

1. $B \to A: \ N_B$
2. $A \to B: \ \{N_B, B, C\}_{k_{AB}}$

dove C è un *confounder*, un elemento casuale utile ad evitare attacchi di tipo chosen-plaintext:

1. $E(B) \to A: \ msg$
2. $A \to E(B): \ \{msg, B\}_{k_{AB}}$

Un esempio reale di protocollo di autenticazione bilaterale è lo *SKID3*, che utilizza una funzione hash parametrica ($\to$ §.1.3.3):

1. $B \to A: \ N_B$
2. $A \to B: \ N_A, h_{k_{AB}}(N_B, N_A, B)$
3. $B \to A: \ h_{k_{AB}}(N_A, N_B, A)$

È importante sottolineare che negli schemi a chiave simmetrica i messaggi sono ripudiabili, perché entrambi gli operatori potrebbbero averli creati. Al contrario gli schemi a chiave pubblica forniscono anche non ripudiabilità poiché nella sfida implicano una cifratura tramite chiave privata; esistono due possibilità:

- *decifratura*, il claimant decifra una sfida cifrata con la sua chiave pubblica:

  1. $B \rightarrow A: \ h(N_B), \{N_B, B\}_{pk_B}$
  2. $A \rightarrow B: \ N_B$

  dove l'identificatore B evita attacchi di tipo reflection, mentre $h(N_B)$ è un *witness*: l'hashing della sfida testimonia che il testo in chiaro era già conosciuto da B; questo evita attacchi di tipo chosen-ciphertext;

- *firma digitale*, il claimant firma una sfida, come nel caso dell'*ISO/IEC 9798-3* per l'autenticazione bilaterale basata su nonce:

  1. $B \rightarrow A: \ N_B$
  2. $A \rightarrow B: \ N_A, B, Sign_A(N_A, N_B, B)$
  3. $B \rightarrow A: \ A, Sign_B(N_B, N_A, A)$

  i nonce fungono anche da confounder, evitando che l'avversario si faccia firmare messaggi a piacimento.

Capitolo 2

Validazione di protocolli tramite analisi statica

La teoria presentata in [BFM04-1] e [BFM04-2] propone un metodo per l'analisi statica della specifica di un protocollo crittografico di *autenticazione*. Con specifica si intende la codifica del protocollo in un opportuno linguaggio formale dotato di sufficiente espressività da permettere di esaurire la descrizione del protocollo in tutti i suoi aspetti.

A partire da una specifica di questo tipo, l'analisi elaborata consente di validare il protocollo tramite la verifica di condizioni *statiche*[1], ottenendo quindi la garanzia formale dell'avvenuta autenticazione degli operatori della comunicazione.

Una della caratteristiche fondamentali di questa analisi è il concetto di *località*: sono prese in considerazione *separatamente* le porzioni di protocollo di competenza dei vari partecipanti: se ogni porzione risulta corretta, cioé supera con successo l'analisi, il processo di validazione termina con successo. Poiché le porzioni di protocollo esaminate singolarmente corrispondono agli operatori della comunicazione si parla di *correttezza locale* (→ §.2.3.1).

I metodi di analisi formale si sono rivelati degli strumenti decisamente validi per valutare la bontà della progettazione dei protocolli, permettendo di scoprire errori in protocolli considerati sicuri oppure confermando la correttezza di altri. A partire da questo presupposto si dimostra formalmente la validità dei risultati dell'analisi proposta: il *safety theorem* [BFM04-1] stabilisce che protocolli costituiti da porzioni localmente corrette sono *sicuri*, cioè immuni da ogni attacco che un avversario E decida di portare in uno scenario che corrisponda al modello di Dolev-Yao (→ §.1.2.2). Quindi un protocollo che superi con successo il processo di validazione garantisce l'autenticazione dei partecipanti.

Detto questo va sottolineato che, data la composizionalità delle componenti del protocollo nell'approccio adottato, se la validazione ha successo per

[1] Condizioni di cui è sempre possibile verificare il rispetto, cioè condizioni *decidibili*.

una sessione del protocollo, questo significa che anche un numero arbitrario di sessioni contemporanee è corretto.

L'analisi prende in esame la struttura dei messaggi scambiati dai partecipanti sfruttando un meccanismo di tagging[2] appositamente ideato con lo scopo di assegnare un significato univoco alle singole componenti. I controlli effettuati derivano dalla definizione di un insieme di regole che schematizzano le forme che i messaggi possono assumere e le condizioni alle quali possono essere trasmessi.

La dimostrazione del *safety theorem* si basa sull'assunzione che i tag siano mantenuti nell'implementazione dei protocolli, ma questo non dovrebbe costituire una grande limitazione, dato che l'uso di tag come soluzione all'ambiguità dei messaggi è una pratica piuttosto diffusa.

2.1 Tagging dei messaggi

Il motivo classico che porta un protocollo di sicurezza ad essere attaccabile è la ricezione scorretta del contenuto informativo di un messaggio, la *mancata comprensione*: tipicamente questo porta a sopravvalutare le garanzie fornite dal messaggio. Una regola fondamentale per evitare di cadere in questo errore è progettare il contenuto del messaggio in modo che sia autoesplicativo, che nulla sia lasciato all'interpretazione della controparte.

Ad esempio gli attacchi di tipo *reflection* (→ §.1.4) sono spesso dovuti al mancato rispetto di questa regola:

1. $B \rightarrow E(A): \ \{M\}_{k_{AB}}$
2. $E(A) \rightarrow B: \ \{M\}_{k_{AB}}$

In questo caso l'assunzione sbagliata da parte di B è che sia A l'autore del secondo messaggio, quando questo non indicato da nessuna parte.

Dotando di tag le componenti fondamentali del messaggio, si fornisce una distinzione precisa tra dati e metadati, impedendo che ne possa essere equivocato il significato. Le parti critiche di un messaggio possono essere classificate in identità, sfide, chiavi, messaggi da autenticare o testi ulteriormente cifrati. Il sistema di regole proposto in [BFM04-1] propone un insieme di tag così definito:

- le *identità* dei partecipanti hanno tag Id;
- la *chiavi di sessione* hanno tag Key;
- i *nonce* hanno tre possibili tag che individuano il ruolo preteso dal mittente nell'esecuzione del protocollo: Claim indica un *claimant*, Verif indica un *verifier* (→ §.1.4.4), mentre Owner indica il *proprietario* di una chiave di sessione da autenticare.

[2]Una marcatura di qualche tipo dei componenti del messaggio.

Dato che l'obbiettivo del sistema di tag è rendere univoca l'interpretazione del messaggio, non sarebbe sufficiente indicare semplicemente il tipo delle varie componenti

$$\{A : \mathsf{Id}, n : \mathsf{Nonce}, m : \mathsf{Message}\}_{k_{BT}}$$

poiché in generale rimarrebbero più possibili interpretazioni del significato del messaggio:

1. B chiede a T di autenticarlo con A
2. T informa B che A vuole autenticarsi con B
3. T informa B che M è una chiave di sessione condivisa con A

utilizzando il sistema di tag proposto questi tre scenari sono individuati senza possibili ambiguità:

1. $\{A : \mathsf{Id}, n : \mathsf{Claim}, m\}_{k_{BT}}$
2. $\{A : \mathsf{Id}, n : \mathsf{Verif}, m\}_{k_{BT}}$
3. $\{A : \mathsf{Id}, n : \mathsf{Owner}, m : \mathsf{Key}\}_{k_{BT}}$

Il sistema di regole proposto in [BFM04-2] propone invece un insieme di tag differente:

- le *identità* dei partecipanti hanno sempre tag Id;
- i *messaggi* da autenticare hanno tag Auth;
- i *nonce* hanno quattro possibili tag che individuano sempre il ruolo del mittente (→ §.2.3.3): Claim, Verif, Claim? e Verif?.

2.2 Il linguaggio di codifica

Il formalismo utilizzato per la narrazione dei protocolli prende il nome di *ρ-spi* ed è un'elaborazione di due linguaggi esistenti: lo *spi calculus* [AG99] e il suo derivato *Lysa* [BBD+03].

La sintassi del *ρ-spi* (→ T.2.1) prevede l'insieme dei nomi $\mathcal{N}$ e quello delle variabili $\mathcal{V}$, soprainsiemi rispettivamente di

$$\mathcal{N}_r = \{k, m, n\} \quad \text{e} \quad \mathcal{V}_r = \{x, y, z\}$$

inisemi dei caratteri riservati; gli elementi di entrambi possono ricevere un tagging, espresso con la notazione $identificatore : tag$. I tag sono una particolare varietà di nomi, come le identità appartenenti a $\mathcal{ID} \subset \mathcal{N}$ di cui si individuano i sottoinsiemi riservati:

$$\mathcal{ID}_r = \{I, J\} \quad \mathcal{I}_P = \{A, B\} \quad \mathcal{I}_T = \{T\} \quad \mathcal{I}_E = \{E\}$$

Tabella 2.1: La sintassi del ρ-*spi*.

Notazione: I e $I_1 \neq I_2$ assumono i valori in $\mathcal{ID}$, C assume tutti i valori definiti per i tag, a rappresenta un nome o una variabile.

$\mathcal{M}$ ::=	*Patterns*
m, n, k	names
x, y, z	variables
$a : C$	tagged data
$\mathsf{key}(a)$	key ($\mathsf{key} \in \{\mathsf{Pub/Priv}\}$)

P, Q ::= *Processes*	
$I \triangleright S$	(principal)
$I \triangleright ! S$	(replication)
$P \mid Q$	(composition)
$\mathsf{let}\ k = \mathsf{sym\text{-}key}(I_1, I_2).P$	(key assignment)
$\mathsf{let}\ k = \mathsf{asym\text{-}key}(I).P$	(key assignment)

S ::= *Sequentialprocesses*	
$\mathbf{0}$	(nil)
$\mathsf{new}(n).S$	(restriction)
$\mathsf{in}(\mathcal{M}_1, \ldots, \mathcal{M}_n).S$	(input)
$\mathsf{out}(\mathcal{M}_1, \ldots, \mathcal{M}_n).S$	(output)
$\mathsf{encrypt}\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}_{\mathcal{M}_0}\ \mathsf{as}\ x.S$	(symmetric encryption)
$\mathsf{encrypt}\{\mid\mathcal{M}_1, \ldots, \mathcal{M}_n\mid\}_{\mathcal{M}_0}\ \mathsf{as}\ x.S$	(asymmetric encryption)
$\mathsf{decrypt}\ x\ \mathsf{as}\ \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}_{\mathcal{M}_0}.S$	(symmetric decryption)
$\mathsf{decrypt}\ x\ \mathsf{as}\ \{\mid\mathcal{M}_1, \ldots, \mathcal{M}_n\mid\}_{\mathcal{M}_0}.S$	(asymmetric decryption)
$\mathsf{run}(I_1, I_2, \mathcal{M}?).S$	(run)
$\mathsf{commit}(I_1, I_2, \mathcal{M}?).S$	(commit)

rispettivamente insieme delle identità generiche, dei principal, del trusted server, e dell'avversario. Le chiavi simmetriche sono indicate con la normale notazione k_s, mentre le parti pubbliche e private di una chiave asimmetrica k_a sono indicate rispettivamente con la notazione $\mathsf{Pub}(k_a)$ e $\mathsf{Priv}(k_a)$.

Ad un protocollo corriponde un *processo* P, formato dalla composizione parallela di due o più *principal*, ovvero gli assegnamenti ai partecipanti I dalla porzione di protocollo S di loro competenza, e dalle dichiarazioni delle chiavi utilizzate. L'operatore di *replicazione* $\triangleright!$ indica che ci possono essere un numero arbitrario di copie di S assegnate ad I.

La porzione di protocollo S è definita *sequential process*, poiché si tratta di una sequenza atomica di istruzioni che non prevede interruzioni o sdoppiamenti dell'esecuzione in rami paralleli; $\mathbf{0}$ indica il sequential process nullo, che pone fine all'esecuzione.

Le primitive definite dal ρ-*spi* sono definite in coppie complementari (ad eccezione di new):

- $\mathsf{new}(n).S$ introduce un nuovo nome n nello scope di S;

- $\mathsf{in}(D_1, \ldots, D_n).S$ legge dal canale di comuncazione il messaggio composto dall'insieme degli argomenti $\{D_1, \ldots, D_n\}$, inserendo nello scope di S tutti gli identificatori D_i liberi come variabli;

Tabella 2.2: Esempio di codifica in ρ-*spi*.

Protocol	$\triangleq$	let k_{AB} = sym-key$(A, B)(A \triangleright \mathit{Initiator}(A, B) \mid B \triangleright \mathit{Responder}(B, A))$
Initiator(A,B)	$\triangleq$	new(m).in(x).run(A, B).encrypt$\{x, m\}_{k_{AB}}$ as y.out(y)
Responder(B, A)	$\triangleq$	new(n_B).out(n_B).in(y).decrypt y as $\{n_B, z\}_{k_{AB}}$.commit(B, A)

- out$(D_1, \ldots, D_n).S$ inserisce nel canale di comuncazione il messaggio composto dall'insieme degli argomenti $\{D_1, \ldots, D_n\}$;
- decrypt x as $\{D_1, \ldots, D_n\}_d.S$ inserisce nello scope di S tutti gli identificatori D_i liberi come variabili[3];
- encrypt $\{D_1, \ldots, D_n\}_d$ as x $.S$ lega la variabile x al testo cifrato fino al termine di S;
- run$(I_1, I_2, \mathcal{M}?).S$ dichiara che I_1 sta iniziando una sessione del protocollo con I_2, autenticando l'eventuale messaggio $\mathcal{M}$;
- commit$(I_1, I_2, \mathcal{M}?).S$ dichiara che I_1 ha concluso con successo una sessione del protocollo con I_2, autenticando l'eventuale messaggio $\mathcal{M}$.

Il ρ-*spi* assume che ci sia un unico canale di trasmissione, pubblico e accessibile a tutti i partecipanti, avversari compresi, in cui di volta in volta si immettono i messaggi e da cui li si prelevano. Va sottolineato che la struttura sintattica del ρ-*spi* impedisce di immettere direttamente nel canale di comunicazione testi cifrati, senza passare per una encrypt.

Le componenti dei messaggi possono essere esaminate singolarmente tramite un meccanismo di pattern-matching secondo il quale gli elementi dotati di tag non possono essere messi in corrispondenza con quelli che non ne hanno.

Si può trovare in T.2.2 la codifica del classico protocollo di autenticazione attaccabile tramite *reflection*:

1. $B \rightarrow A: \; n_B$
2. $A \rightarrow B: \; \{n_B, m\}_{k_{AB}}$

2.3 La tecnica di validazione

La tecnica di validazione proposta in [BFM04-1] si applica a tutti i protocolli che possono essere codificati tramite una sequenza di dichiarazioni di *chiavi* e assegnamenti di *sequential process*:

$$\mathsf{keys}(k_1, \ldots, k_n).(I_1 \triangleright !S_1 | \ldots | I_n \triangleright !S_n)$$

[3] encrypt e decrypt accettano cifratura simmetrica e asimmetrica con la stessa semantica.

insieme che include tutti i protocolli di maggiore interesse reperibili al momento; il costrutto keys denota brevemente l'insieme delle dichiarazioni delle chiavi k_i.

Ogni processo $\mathsf{keys}(k_1, \ldots, k_n).I_i \triangleright !S_i$ viene sottoposto al controllo di correttezza locale: se tutti i processi risultano localmente corretti il protocollo è valido.

2.3.1 Safety Theorem

La dimostrazione del *safety theorem* richiede alcune ulteriori restrizioni sulle modalità tramite cui le chiavi sono distribuite o immesse nel canale di comunicazione da parte del *trusted server*. Dato un processo $\mathsf{keys}(k_1, \ldots, k_n).I \triangleright S$ si assume che:

- le chiavi a lungo termine non siano mai immesse nel canale di comunicazione;
- solo i principal e non i TTP possano ricevere le chiavi di sessione;
- le chiavi di sessione siano distribuite dai TTP ad al massimo due principal, e solo una volta;

Quando queste richieste sono soddisfatte, il principal $I \triangleright S$ è sicuro per quanto riguarda la dichiarazione delle chiavi, o $\{k_1, \ldots, k_n\}$-*safe*.

Definizione 2.3.1 (Correttezza locale). *Sia P il processo descritto sopra $\mathsf{keys}(k_1, \ldots, k_n).I \triangleright S$ dove k_i è una chiave a lungo termine condivisa tra I_i e J_i, si dice che P è* localmente corretto *se e solo se $I \triangleright S$ è $\{k_1, \ldots, k_n\}$-safe ed è possibile derivare il giudizio*

$$I; \varnothing; \Pi_{k_1,\ldots,k_n} \vdash S$$

con

$$\Pi_{k_1,\ldots,k_n} = k_1 \mapsto key_{sym}(I_1, J_1), \ldots, k_m \mapsto key_{sym}(I_m, J_m),$$
$$k_{m+1} \mapsto key_{asym}(I_{m+1}), \ldots, k_n \mapsto key_{asym}(I_n)$$

Definizione 2.3.2 (Correttezza). *Un protocollo si dice* corretto *se il corrispondente processo $\mathsf{keys}(k_1, \ldots, k_n).(I_1 \triangleright S_1 | \ldots | I_m \triangleright S_m)$ è corretto, cioè se $\forall i \in \{1, \ldots, m\}$ il processo $\mathsf{keys}(k_1, \ldots, k_n).I_i \triangleright S_i$ è localmente corretto.*

Il *safety theorem* stabilisce che un protocollo corretto è *sicuro*: la correttezza locale dei singoli principal è una condizione sufficiente per la sicurezza dell'intero protocollo[4].

Teorema 2.3.1 (Safety). *Sia $P = \mathsf{keys}(k_1, \ldots, k_n).(I_1 \triangleright S_1 | \ldots | I_m \triangleright S_m)$. Se P è corretto, allora il corrispondente protocollo è sicuro.*

[4]Per la dimostrazione si veda [BFM04-1].

2.3.2 Regole di analisi sintattica

Le regole di validazione riportate in T.2.3 e in T.2.4 sono costituite da derivazioni di giudizi della forma $I; \Gamma; \Pi \vdash S$, che validano il processo S assegnato ad I tenendo conto degli elementi presenti nei contenitori Γ e Π. Il contenitore Γ costruisce una cronologia delle primitive incontrate durante la scansione del sequential process, mentre il contenitore Π tiene traccia dei nonce generati e del relativo stato: al momento della generazione tramite new un nonce n ha stato *unchecked*, l'applicazione delle regole può portare n nello stato *checked*; naturalmente le regole sono definite in modo tale che l'esame di un nonce già in stato *checked* faccia fallire il processo di validazione. I contenitori possono avere al più un elemento per ogni nome o variabile presene nello scope del sequential process S.

Si assume inoltre che ogni messaggio, sia esso cifrato o in chiaro, possa contenere un solo componente tra $\{\mathsf{Claim}, \mathsf{Verif}, \mathsf{Owner}\}$, al più un componente con tag Id e al più un componente con tag Key, senza alcuna restrizione sull'ordine.

Le prime due regole in T.2.3 stabiliscono quali siano le modalità a disposizione di un *principal* A per autenticare un altro principal B:

- secondo la regola (AUTHENTICATE CLAIM) ad A è consentito eseguire una commit solo nel caso in cui in precedenza abbia generato un nonce n e abbia ricevuto un testo cifrato $\{B : \mathsf{Id}, n : \mathsf{Claim}, \ldots\}_k$ in cui il nonce n fosse lo stesso predentemente generato e k una chiave condivisa con B o con il *trusted server* T;

- analogamente la regola (AUTHENTICATE VERIF) stabilisce che ad A è consentito eseguire una commit solo se in precedenza ha generato un nonce n e ha ricevuto un testo cifrato $\{A : \mathsf{Id}, n : \mathsf{Verif}, \ldots\}_k$ in cui il nonce n fosse lo stesso predentemente generato e k una chiave condivisa con B;

- secondo (AUTHENTICATE OWNER) invece ad A è consentito eseguire una commit solo nel caso in cui abbia ricevuto da T un messaggio cifrato del tipo $\{B : \mathsf{Id}, n : \mathsf{Owner}, x : \mathsf{Key}, \ldots\}_k$ con nonce n generato in precedenza, ed abbia inoltre ricevuto almeno un messaggio cifrato con la chiave di sessione x.

La correttezza di queste forme di autenticazione è garantita solamente nel caso in cui in ruoli implicitamente definiti dai tag siano rispettati e sia rispettata la *freshness* di nonce e chiavi di sessione ($\rightarrow$ §.1.4.1). Le altre regole definite in T.2.3 hanno appunto lo scopo di assicurare il rispetto di queste proprietà. Le regole (CLAIMANT), (VERIFIER) e (OWNER) stabiliscono le modalità in cui A può dichiarare la sua volontà di autenticarsi con B:

- secondo (CLAIMANT) A può richiedere di autenticarsi presso B col ruolo di *claimant* inviandogli direttamente un messaggio cifrato del

Tabella 2.3: Correttezza locale: regole per i principal e il TTP.

$\Pi(x) = enc\{\ldots\}_d$ significa che $x \mapsto enc\{\ldots\}_d \in \Pi$.
$\Pi(\bullet) = enc\{\ldots\}_d$ significa che esiste x tale che $\Pi(x) = enc\{\ldots\}_d$.

Regole per il Claimant e il Verifier

Authenticate Claim

$$\frac{\begin{array}{c} A;\Gamma, n : \text{checked}; \Pi \vdash S \\ \Pi(\bullet) = dec\{B : \mathsf{Id}, n : \mathsf{Claim}, \ldots\}_k \qquad \Pi(k) \in \{key_{sym}(A,T), key_{sym}(A,B)\} \end{array}}{A;\Gamma, n : \text{unchecked}; \Pi \vdash \mathsf{commit}(A,B).S}$$

Authenticate Verif

$$\frac{A;\Gamma, n : \text{checked}; \Pi \vdash S \qquad \Pi(\bullet) = dec\{A : \mathsf{Id}, n : \mathsf{Verif}, \ldots\}_k \qquad \Pi(k) = key_{sym}(A,B)}{A;\Gamma, n : \text{unchecked}; \Pi \vdash \mathsf{commit}(A,B).S}$$

Authenticate Owner

$$\frac{\begin{array}{c} A;\Gamma, n : \text{checked}; \Pi \vdash S \qquad \Pi(\bullet) = dec\{B : \mathsf{Id}, n : \mathsf{Owner}, y : \mathsf{Key}, \ldots\}_k \\ \Pi(k) = key_{sym}(A,T) \qquad \Pi(\bullet) = dec\{D_1, \ldots, D_m\}_y \\ (\Pi(\bullet) = enc\{D'_1, \ldots, D'_m\}_{y'} \quad \text{implies } \exists i \text{ s.t. } D'_i \text{ does not match } D_i) \end{array}}{A;\Gamma, n : \text{unchecked}; \Pi \vdash \mathsf{commit}(A,B).S}$$

Claimant

$$\frac{A;\Gamma;\Pi, B \mapsto run, y \mapsto enc\{A : \mathsf{Id}, x : \mathsf{Claim}\}_k \vdash S \qquad \Pi(k) = key_{sym}(A,B)}{A;\Gamma;\Pi, B \mapsto run \vdash \mathsf{encrypt}\{A : \mathsf{Id}, x : \mathsf{Claim}, \ldots\}_k \ \mathsf{as}\ y.S}$$

Verifier

$$\frac{A;\Gamma;\Pi, B \mapsto run, y \mapsto enc\{B : \mathsf{Id}, x : \mathsf{Verif}\}_k \vdash S \qquad \Pi(k) = key_{sym}(A,T)}{A;\Gamma;\Pi, B \mapsto run \vdash \mathsf{encrypt}\{B : \mathsf{Id}, x : \mathsf{Verif}, \ldots\}_k \ \mathsf{as}\ y.S}$$

Owner

$$\frac{\begin{array}{c} A;\Gamma;\Pi, B \mapsto run, y \mapsto enc\{D_1, \ldots, D_m\}_x \vdash S \\ \Pi(\bullet) = dec\{B : \mathsf{Id}, n : \mathsf{Owner}, x : \mathsf{Key}, \ldots\}_k \qquad \Pi(k) = key_{sym}(A,T) \end{array}}{A;\Gamma;\Pi, B \mapsto run \vdash \mathsf{encrypt}\{D_1, \ldots, D_m\}_x \ \mathsf{as}\ y.S}$$

Run

$$\frac{A;\Gamma;\Pi, B \mapsto run \vdash S}{A;\Gamma;\Pi \vdash \mathsf{run}(A,B).S}$$

Regole per il TTP

Ttp Forward & Check

$$\frac{\begin{array}{c} T;\Gamma, n : \text{checked}; \Pi, y \mapsto enc\{A : \mathsf{Id}, x : \mathsf{Claim}\}_{k_{BT}} \vdash S \\ \Pi(\bullet) = dec\{B : \mathsf{Id}, n : \mathsf{Verif}, \ldots\}_{k_{AT}} \qquad \Pi(k_{BT}) = key_{sym}(B,T) \qquad \Pi(k_{AT}) = key_{sym}(A,T) \end{array}}{T;\Gamma, n : \text{unchecked}; \Pi \vdash \mathsf{encrypt}\ \{A : \mathsf{Id}, x : \mathsf{Claim}, \ldots\}_{k_{BT}} \ \mathsf{as}\ y.S}$$

Ttp Forward

$$\frac{\begin{array}{c} T;\Gamma;\Pi, y \mapsto enc\{A : \mathsf{Id}, x : \mathsf{Claim}\}_{k_{BT}} \vdash S \\ \Pi(\bullet) = dec\{B : \mathsf{Id}, x : \mathsf{Verif}, \ldots\}_{k_{AT}} \qquad \Pi(k_{BT}) = key_{sym}(B,T) \qquad \Pi(k_{AT}) = key_{sym}(A,T) \end{array}}{T;\Gamma;\Pi \vdash \mathsf{encrypt}\{A : \mathsf{Id}, x : \mathsf{Claim}, \ldots\}_{k_{BT}} \ \mathsf{as}\ y.S}$$

Ttp Distribute

$$\frac{T;\Gamma;\Pi, y \mapsto enc\{A : \mathsf{Id}, x : \mathsf{Owner}, k_s : \mathsf{Key}\}_k \vdash S}{T;\Gamma;\Pi \vdash \mathsf{encrypt}\ \{A : \mathsf{Id}, x : \mathsf{Owner}, k_s : \mathsf{Key}, \ldots\}_k \ \mathsf{as}\ y.S}$$

Tabella 2.4: Correttezza locale: regole generiche.

$$\textsc{Nil}\quad \frac{}{I;\Gamma;\Pi \vdash \mathbf{0}} \qquad \textsc{New}\quad \frac{I;\Gamma, n : \text{unchecked}; \Pi \vdash S \qquad n \text{ fresh in } \Gamma}{I;\Gamma;\Pi \vdash \mathsf{new}(n).S}$$

$$\textsc{Key}\quad \frac{I;\Gamma;\Pi, k \mapsto key_{sym}(A,B) \vdash P}{I;\Gamma;\Pi \vdash \mathsf{let}\ k = \mathsf{sym\text{-}key}(A,B).P} \qquad \textsc{Input}\quad \frac{I;\Gamma;\Pi \vdash S}{I;\Gamma;\Pi \vdash \mathsf{in}(\ldots).S} \qquad \textsc{Output}\quad \frac{I;\Gamma;\Pi \vdash S}{I;\Gamma;\Pi \vdash \mathsf{out}(\ldots).S}$$

$$\textsc{Decryption}\quad \frac{I;\Gamma;\Pi, y \mapsto dec\{D_1,\ldots,D_m\}_d \vdash S}{I;\Gamma;\Pi \vdash \mathsf{decrypt}\ y\ \mathsf{as}\ \{D_1,\ldots,D_m\}_d.S}$$

$$\textsc{Encryption}\quad \frac{\begin{array}{c} I;\Gamma;\Pi.y \mapsto enc\{d_1,\ldots,d_m\}_d \vdash S \\ \Pi(\bullet) = dec\{\ldots, d : \mathsf{Key}, \ldots\}_k \text{ implies } \Pi = \Pi'.B \mapsto run.x \mapsto enc\{\ldots\}_d, \Pi'' \end{array}}{I;\Gamma;\Pi \vdash \mathsf{encrypt}\{d_1,\ldots,d_m\}_d\ \mathsf{as}\ y.S}$$

tipo $\{A : \mathsf{Id}, n : \mathsf{Claim}, \ldots\}_{k_{AB}}$ dove k_{AB} è una chiave a lungo termine condivisa con B, mentre x in teoria dovrebbe essere una sfida ricevuta da B in precedenza, ma questo non è imposto da (Claimant) poiché il controllo spetta a B;

- (Verifier) invece presuppone la presenza di un TTP: A può richiedere a T di autenticarsi presso B, con quest'ultimo nel ruolo di *verifier*, tramite un messaggio del tipo $\{B : \mathsf{Id}, n : \mathsf{Verif}, \ldots\}_{k_{AT}}$ in cui k_{AT} è una chiave a lungo termine condivisa con T, mentre x consiste nella sfida generata da B o T;

- tramite (Owner) A può spedire un messaggio cifrato $\{D_1, \ldots, D_n\}_{k_s}$ come conferma dell'avvenuta ricezione della chiave di sessione k_s, a patto di aver ricevuto dal trusted server T un messaggio cifrato del tipo $\{B : \mathsf{Id}, n : \mathsf{Owner}, k_s : \mathsf{Key}, \ldots\}_k$ e di aver effettuato una run con B in precedenza.

Tutte e tre le regole avviano una nuova sessione del protocollo, devono quindi essere precedute da una run che segnali esplicitamente questo fatto: si richede che la run sia l'ultima istruzione memorizzata per assicurarsi che queste istruzioni siano utilizzato solamente una volta, all'avvio del protocollo.

Le ultime tre regole in T.2.3 definiscono il comportamento del *trusted server* T:

- tramite (Ttp Forward & Check) T può generare un messaggio cifrato del tipo $\{A : \mathsf{Id}, x : \mathsf{Claim}, \ldots\}_{k_{BT}}$ nel caso in cui in precedenza abbia ricevuto un messaggio del tipo $\{B : \mathsf{Id}, n : \mathsf{Verif}, \ldots\}_{k_{AT}}$, abbia verificato la sifda n, e K_{AT} e K_{BT} siano chiavi a lungo termine condivise rispettivamente con A e B;

- tramite (TTP FORWARD) T può generare un messaggio cifrato del tipo $\{A : \mathsf{Id}, x : \mathsf{Claim}, \ldots\}_{k_{BT}}$ nel caso in cui in precedenza abbia ricevuto un messaggio del tipo $\{B : \mathsf{Id}, x : \mathsf{Verif}, \ldots\}_{k_{AT}}$, e K_{AT} e K_{BT} siano chiavi a lungo termine condivise rispettivamente con A e B; in questo caso è B che si occupa di verificare la sfida x;

- (TTP DISTRIBUTE) invece stabilisce che T può distribuire delle nuove chiavi di sessione ai principal tramite l'invio di messaggi cifrati del tipo $\{A : \mathsf{Id}, x : \mathsf{Owner}, k_s : \mathsf{Key}, \ldots\}_k$, dove k è una chiave a lungo termine condivisa con la controparte.

Le regole definite in T.2.4 permettono di validare le primitive rimanenti, popolando i contenitori coerentemente con quanto atteso dalle regole definite in T.2.3: (NIL), (NEW), (KEY), (INPUT), (OUTPUT) e (DECRYPTION) praticamente si limitano a far proseguire il processo con modifche minime ai contenitori. (ENCRYPTION) viene utilizzata nei casi in cui non si applichi nessuna delle regole definite in T.2.3, si sostituisce infatti ad (OWNER) quando si tratta di validare cifrature di messaggi tramite chiavi di sessione: questa situazione è lecita solo se prima il protocollo è stato avviato tramite una (RUN), che a sua volta implica l'applicazione di una (OWNER) con la stessa chiave di sessione k_s; per garantire questa il rispetto di questa condizione (ENCRYPTION) richiede che siano presenti in Π i due eventi consecutivi:

$$A \mapsto run(B).x \mapsto enc\{\ldots\}_{k_s}$$

2.3.3 Sistema di tipi ed effetti

I protocolli di autenticazione sono generalmente basati su schemi *sfida-risposta* in cui si prova la conoscenza di una chiave segreta cifrando o decifrando una nuova sfida. Questo sistema di regole prende in considerazione protocolli che sfruttano come sfida i nonce, individuandone tre categorie di sfida:

- *Public-Out Secret-Home (*POSH*)*, in cui il nonce è spedito in chiaro ed è mandato indietro cifrato;

- *Secret-Out Public-Home (SOPH)*, in cui accade l'inverso;

- *Secret-Out Secret-Home (SOSH)*, in cui il nonce è cifrato in entrambe le direzioni.

Tabella 2.5: Definizioni dei tipi.

a, b rapprsentano nomi o variabili.

Types

T	$::=$	$key_{sym}(I, J)$	long-term key
		$key_{asym}(I)$	asymmetric component
		$key_{pub}(I)$	public key
		$key_{priv}(I)$	private key
		Un	untrusted
		$nonce(I, J, M)$	secret nonce
		$enc(f)$	ciphertext

Atomic effects

f	$::=$	$\oslash$	empty effect
		$fresh(n)$	public nonce freshness
		$fresh(n, I, J, M)$	secret nonce freshness
		$run(I, J, M)$	run
		$in(M)$	input
		$dec\{M_1, \ldots, M_n\}_{M_0}$	decryption

Environment Well-Formedness

$$\textsc{Env } \oslash \quad \oslash \vdash \diamond$$

$$\textsc{Env Name and Var}\quad \frac{a \neq I \qquad a \notin dom(\Gamma) \qquad \Gamma \vdash \diamond \qquad T \text{ depends on } M \Rightarrow \Gamma \vdash M : Un}{\Gamma, a : T \vdash \diamond}$$

$$\textsc{Projection}\quad \frac{\Gamma, a : T, \Gamma' \vdash \diamond}{\Gamma, a : T, \Gamma' \vdash a : T}$$

Typing Rules for Messages

$$\textsc{Identitiy} \quad \Gamma \vdash I : Un$$

$$\textsc{Public Key}\quad \frac{\Gamma \vdash k : key_{asym}(I)}{\Gamma \vdash \mathsf{Pub}(k) : key_{pub}(I)}$$

$$\textsc{Private Key}\quad \frac{\Gamma \vdash k : key_{asym}(I)}{\Gamma \vdash \mathsf{Priv}(k) : key_{priv}(I)}$$

$$\textsc{Subsumption}\quad \frac{\Gamma \vdash N : T' \qquad \Gamma \vdash T' <: T}{\Gamma \vdash N : T}$$

$$\textsc{Enemy Knowledge}\quad \frac{T \in \{key_{sym}(E, I), key_{asym}(E), key_{priv}(E), key_{pub}(I), nonce(E, I, M)\}}{\Gamma \vdash T <: Un}$$

Nozioni di base

Le definizioni dei tipi e le regole per derivare i giudizi su di essi sono presenti in T.2.5. Il contenitore dei tipi Γ è un insieme ordinato di coppie nome o varibile e relativo tipo.

I tipi servono a regolare l'utilizzo dei relativi valori durante il processo di autenticazione. I tipi distinguono chiavi simmetriche oppure asimmetriche con le relative istanze pubbliche o private; come è lecito attendersi, le chiavi pubbliche sono accessibili a tutti, mentre quelle private sono riservate esclusivamente ai proprietari. Ogni elemento non dotato di tag e potenzialmente conosciuto dall'avversario ha tipo Un (*untrusted*). Un nonce usato da I e J in una sessione *SOPH/SOSH* per l'autenticazione di un messaggio $\mathcal{M}$ ha tipo iniziale $nonce(I, J, \mathcal{M})$: al momento di rispedirlo indietro il tipo viene posto a Un in modo che possa essere trasmesso anche in chiaro. I parteciapnti onesto possono cifrare dei messaggi soltanto sotto determinate ipotesi: il testo cifrato che deve rispettare l'ipotesi rappresentata dall'effetto f ha tipo $enc(f)$.

Esistono poi delle condizioni alle quali deve sottostare un contenitore per essere *ben formato*: secondo (ENV $\oslash$) un contenitore vuoto è sempre ben formato, altrimenti (ENV NAME AND VAR) stabilisce che un contenitore può contenere al più un'occorrenza per ogni nome o variabile a; infine se un tipo ha dei parametri questi devono avere tipo Un, il che praticamente equivale a imporre che $\mathcal{M}$ abbia tipo Un, dato che secondo (IDENTITIY) tutte le identità hanno tipo Un.

Ci sono infine due regole per definire dei sottotipi: la regola (ENEMY KNOWLEDGE) definisce la conoscenza dell'avversario, che può essere dotato sia di chiavi a lungo termine sia di chiavi asimmetriche; conosce inoltre tutte le chiavi pubbliche e può conoscere i nonce segreti a lui destinati; perciò tutti questi elementi sono sottotipi di Un. (SUBSUMPTION) sara discussa nei prossimi paragrafi.

Il processo di tipizzazione

È necessario premettere che, per tenere conto dei differenti tipi individuati, al momento della creazione di un nonce bisogna specificarne il tipo:

$$\mathsf{new}(n : T). \quad \text{con } T \in \{Un, nonce(I, J, \mathcal{M})\}$$

La tipizzazione di un processo P avviene tramite un giudizio del forma $\Gamma \vdash P : e$, che significa che P può essere tipato rispetto all'insieme di tipi in Γ e soddisfacendo le ipotesi espresse tramite gli effetti presenti in e. La tipizzazione di un sequential process S eseguito da I si svolge in modo analogo: $I; \Gamma \vdash S : e$.

Il contenitore e è un multinsieme che ha per elementi gli effetti riportati in T.2.5:

- La *freshness* di un nonce n è espressa tramite gli effetti $fresh(n)$ o $fresh(n, I, J, \mathcal{M})$ a seconda che sia utilizzato in una sessione *POSH*, nel qual caso n ha tipo Un, oppure *SOPH/SOSH*, che impone il tipo $nonce(I, J, \mathcal{M})$. L'effetto appropriato è inserito in e al momento della creazione del nonce e rimosso quando una `commit` ne verifica la *freshness*; questo è un altro metodo per garantire che un nonce sia verificato al più una volta, il che implica che ci sia al più una `commit` per ogni nonce.

- Quando un'operazione di cifratura corrisponde all'avvio di una sessione del protocollo, deve essere preceduta dalla corrispondente run; l'effetto $run(I, J, \mathcal{M})$ ha appunto questo scopo: viene inserito in e al momento di tipare la run e rimosso all'occorrenza della relativa encrypt.

- La ricezione della componente M di un messaggio comporta l'inserimento in e dell'effetto $in(M)$, mentre la decifratura di un testo $\{M_1, \ldots, M_n\}_{M_0}$ è registrata tramite l'inserimento nel contenitore e dell'effetto $dec\{M_1, \ldots, M_n\}_{M_0}$.

Le regole per tipare un processo P, definite in T.2.6, si dividono in regole generiche e regole di autenticazione. (Symmetric Key) e (Asymmetric Key) tipano le dichiarazioni delle chiavi, limitandosi ad inserirle in Γ con il tipo appropriato. (Replication) e (Par) non fanno altro che applicare il processo di tipizzazione agli elementi successivi, unendo i contenitori degli effetti dei processi composti parallelamente.

La regola (Identity Assignment) formalizza il passaggio della analisi dal processo P ai singoli sequential process S, che possono essere eseguiti solamente da partecipanti onesti, difatti l'avversario è modellato implicitamente dalla semantica. Le regole relativei alle primitive del sequnetial process sono lineari: (Nil) consente di tipare il processo nullo, (Run) aggiunge l'effetto $run(A, I, \mathcal{M})$ ad e, (Input) inserisce in Γ tutte le variabili libere con tipo Un, ed in e gli effetti $in(M_i)$ corrispondenti alle componenti del messaggio ricevuto; infine (Output) stabilisce che solo gli elementi di tipo Un possono essere immessi nella rete di comunicazione.

Alla primitiva new possono essere applicate due regole diverse a seconda del tipo del nonce: (New Name) in caso di sessione *POSH*, (New Nonce), in caso di sessione *SOPH/SOSH*; il tipo $nonce(I, J, \mathcal{M})$ impedisce che n sia immesso nella rete in chiaro.

(Symmetric Encrypt) e (Asymmetric Encrypt) tipano le operazioni di cifratura successive a quella che avvia il protocollo, richiedono quindi che in e non sia presente l'effetto $run(I, J, \mathcal{M})$ a segnalare l'avvenuto avvio.

(Symmetric Decrypt) e (Asymmetric Decrypt) assegnano alle variabili libere che trovano nel testo da decifrare i tipi più specifici a disposizione, questo significa semplicemente che a tutti gli elementi che non hanno tag Claim? o Verif? è assegnato il tipo Un, mentre a quest'ultimi è assegnato il tipo $nonce(I, J, \mathcal{M})$.

Sessione di autenticazione *POSH*

In una sessione di autenticazione *POSH* J genera un nuovo nonce n con tipo Un (New Name) e lo immette nella rete; I lo riceve e crea un testo cifrato tramite (POSH Request) allo scopo di autenticare un messaggio $\mathcal{M}$ con J; una volta ricevuto il testo cifrato J autentica I e $\mathcal{M}$ utilizzando (POSH Commit).

Tabella 2.6: Processo di tipizzazione.

In Symmetric Decrypt, $\Gamma \vdash K : key_{sym}(I,J) \Rightarrow A \in \{I,J\}$.
In Asymmetric Decrypt, $\Gamma \vdash K : key_{priv}(I) \Rightarrow A = I$.
In Symmetric e Asymmetric Decrypt, $\Gamma_{\overline{A}} = \Gamma[n : Un/n : nonce(I,A,M)]$, $\forall n, I, M$.

Symmetric Key
$$\frac{\Gamma, k : key_{sym}(I,J) \vdash P : e}{\Gamma \vdash \mathsf{let}\ k = \mathsf{sym\text{-}key}(I,J).P : e}$$

Asymmetric Key
$$\frac{\Gamma, k : key_{asym}(I) \vdash P : e}{\Gamma \vdash \mathsf{let}\ k = \mathsf{asym\text{-}key}(I).P : e}$$

Replication
$$\frac{\Gamma \vdash A \rhd S : e}{\Gamma \vdash A \rhd !S : e}$$

Par
$$\frac{\Gamma \vdash P : e_P \qquad \Gamma \vdash Q : e_Q}{\Gamma \vdash P|Q : e_P + e_Q}$$

Identity Assignment
$$\frac{A; \Gamma \vdash S : e}{\Gamma \vdash A \rhd S : e}$$

Nil
$$\frac{\Gamma \vdash \diamond}{A; \Gamma \vdash \mathbf{0} : e}$$

Run
$$\frac{A; \Gamma \vdash S : e + [run(A,I,M)]}{A; \Gamma \vdash \mathsf{run}(A,I,M).S : e}$$

Input
$$\frac{A; \Gamma, fv(M_1,\ldots,M_n) : Un \vdash S : e + [in(M_1) + \ldots + in(M_n)]}{A; \Gamma \vdash \mathsf{in}(M_1,\ldots,M_n).S : e}$$

Output
$$\frac{\forall i \in [1,n] \qquad \Gamma \vdash M_i : Un \qquad A; \Gamma \vdash S : e}{A; \Gamma \vdash \mathsf{out}(M_1,\ldots,M_n).S : e}$$

New Name
$$\frac{A; \Gamma, n : Un \vdash S : e + [fresh(n)]}{A; \Gamma \vdash \mathsf{new}(n : Un).S : e}$$

New Nonce
$$\frac{A; \Gamma, n : nonce(I,A,M) \vdash S : e + [fresh(n,I,A,M)]}{A; \Gamma \vdash \mathsf{new}(n : nonce(I,A,M)).S : e}$$

Symmetric Encrypt
$$\frac{A; \Gamma \vdash \{M_1,\ldots,M_n\}_K : enc(f) \qquad A; \Gamma, z : Un \vdash S : e}{A; \Gamma \vdash \mathsf{encrypt}\ \{M_1,\ldots,M_n\}_K\ \mathsf{as}\ z.S : e + [f]}$$

Asymmetric Encrypt
$$\frac{A; \Gamma \vdash \{|M_1,\ldots,M_n|\}_K : enc(f) \qquad A; \Gamma, z : Un \vdash S : e}{A; \Gamma \vdash \mathsf{encrypt}\ \{|M_1,\ldots,M_n|\}_K\ \mathsf{as}\ z.S : e + [f]}$$

Symmetric Decrypt
$$\frac{\begin{array}{c}\Gamma \vdash M : Un \quad A; \Gamma, x_1 : T_1, \ldots, x_m : T_m \vdash S : e + [dec\{M_1,\ldots,M_n\}_K] \\ fv(M_1,\ldots,M_n) = x_1,\ldots,x_m \quad T_1,\ldots,T_m \text{ sono i tipi più specifici tali che} \\ I; \Gamma_{\overline{A}}, x_1 : T_1, \ldots, x_m : T_m \vdash \{M_1,\ldots,M_n\}_K : enc(f)\end{array}}{A; \Gamma \vdash \mathsf{decrypt}\ M\ \mathsf{as}\ \{M_1,\ldots,M_n\}_K.S : e}$$

Asymmetric Decrypt
$$\frac{\begin{array}{c}\Gamma \vdash M : Un \quad A; \Gamma, x_1 : T_1, \ldots, x_m : T_m \vdash S : e + [dec\{|M_1,\ldots,M_n|\}_{\overline{K}}] \\ fv(M_1,\ldots,M_n) = x_1,\ldots,x_m \quad T_1,\ldots,T_m \text{ sono i tipi più specifici tali che} \\ I; \Gamma_{\overline{A}}, x_1 : T_1, \ldots, x_m : T_m \vdash \{|M_1,\ldots,M_n|\}_{\overline{K}} : enc(f)\end{array}}{A; \Gamma \vdash \mathsf{decrypt}\ M\ \mathsf{as}\ \{|M_1,\ldots,M_n|\}_K.S : e}$$

- Il giudizio deinito da (POSH REQUEST) permette la cifratura di un messaggio della forma $\{Id : \mathsf{Id}, n : C, \mathcal{M} : \mathsf{Auth}, \widetilde{M}\}_K$, dove $\widetilde{M}$ rapprsenta una sequenza di elementi. Il tag del nonce n e l'identità Id cambiano a seconda del tipo di K: se K è una chiave simmetrica condivisa con J, allora I comunica a J che I è il claimant tramite il tag Claim, oppure che a J spetta il ruolo di verifier tramite Verif. Poiché K è nota solo a I e J, i due messaggi significano la stessa cosa: I è il claimant e J è il verifier. Se invece K è la chiave privata di I va specificato chi sia il verifier atteso dato che l'identità del mittente è provata dalla cifratura con chiave privata. In ogni caso il messaggio $\mathcal{M}$ ha tag Auth. Dato che questa regola segnala l'inizio di una sessione *POSH*, deve essere presente tra gli effetti almeno una $run(I, J, \mathcal{M})$. Va sottolineato infine che il giudizio $\Gamma \vdash \widetilde{M} : Un$ implica che nessun elemento in $\widetilde{M}$ sia dotato di tag.

- Tramite (POSH COMMIT) J, dopo aver ricevuto il testo cifrato e averne verificato il nonce, accetta $\mathcal{M}$ e la richiesta di autenticazione di I. Il controllo di n si conclude con la rimozione da e dell'effetto $fresh(n)$, il che impedisce che n possa essere verificato un'altra volta.

Sessione di autenticazione *SOPH/SOSH*

J può iniziare una sessione *SOPH/SOSH* con I per autenticare un messaggio $\mathcal{M}$, generando un nuovo nonce n di tipo *nonce*$(I, J, \mathcal{M})$ tramite (NEW NONCE); il nonce va quindi cifrato tramite (SOPH/SOSH INQUIRY) ed inviato a I. Questo messaggio chiede a I se intende autenticarsi con J e se è d'accordo sul messaggio $\mathcal{M}$: quando I riceve il testo cifrato può confermare la richiesta in chiaro tramite (OUTPUT), oppure con un nuovo testo cifrato. In entrambi i casi il nonce deve avere tipo Un per poter essere immesso nella rete: il cast viene effettuato tramite (SOPH/SOSH CONFIRM). Quando J riceve indietro il nonce autentica I e $\mathcal{M}$ tramite (SOPH/SOSH COMMIT).

- (SOPH/SOSH INQUIRY) permette ad un principal di cifrare il nonce in un messaggio della forma $\{Id : \mathsf{Id}, n : C, \mathcal{M} : \mathsf{Auth}, \widetilde{M}\}_K$, in cui K può essere o la chiave pubblica di J o una chiave condivisa con J. I tagging possibili sono analoghi a quelli di (POSH REQUEST), hanno però forma *interrogativa*, dato che il messaggio costituisce una domanda. In questo modo si evita di confondere le sfide cifrate *SOPH/SOSH* con le risposte cifrate *POSH*.

- Tramite (SOPH/SOSH CONFIRM) I può confermare la propria intenzione di autenticare $\mathcal{M}$ presso J rispedendo indietro il nonce n appena ricevuto. Sia (OUTPUT) che le (ENCRYPTION) richiedono che le componenti dei messaggi abbiano tipo Un: (SOPH/SOSH CONFIRM) si occupa di convertire il tipo di n in Un, validando la run corrispondente.

Tabella 2.7: Regole di autenticazione.

Encryption Rules

POSH REQUEST

$$\frac{\Gamma \vdash N, M, \widetilde{M} : Un \qquad \Gamma \vdash K : T \qquad T, C, Id \text{ assumono i valori in } POSH(I,J)}{I; \Gamma \vdash \{\mathsf{Id}(Id), C(N), \mathsf{Auth}(M), \widetilde{M}\}_K : enc(run(I,J,M))}$$

POSH(I, J)

T	C	Id
$key_{sym}(I,J)$	Claim	I
$key_{sym}(I,J)$	Verif	J
$key_{priv}(I)$	Verif	J

SOPH/SOSH INQUIRY

$$\frac{\Gamma \vdash N : nonce(I,J,M) \qquad \Gamma \vdash M, \widetilde{M} : Un \qquad \Gamma \vdash K : T \qquad T, C, Id \text{ assumono i valori in } SOPH/SOSH(I,J)}{J; \Gamma \vdash \{\mathsf{Id}(Id), C(N), \mathsf{Auth}(M), \widetilde{M}\}_K : enc(\oslash)}$$

SOPH/SOSH(I, J)

T	C	Id
$key_{sym}(I,J)$	Claim?	I
$key_{sym}(I,J)$	Verif?	J
$key_{pub}(I)$	Verif?	J

SYMMETRIC ENCRYPT

$$\frac{\forall i \in [1,n] \qquad \Gamma \vdash M_i : Un \qquad \Gamma(K) \in \{Un, key_{sym}(I,J)\}}{I; \Gamma \vdash \{M_1, \ldots, M_n\}_K : enc(\oslash)}$$

ASYMMETRIC ENCRYPT

$$\frac{\forall i \in [1,n] \qquad \Gamma \vdash M_i : Un \qquad \Gamma(K) \in \{Un, key_{priv}(I), key_{pub}(J)\}}{I; \Gamma \vdash \{|M_1, \ldots, M_n|\}_K : enc(\oslash)}$$

Secret Nonce Typing

SOPH/SOSH CONFIRM

$$\frac{I; \Gamma, a : Un, \Gamma' \vdash S : e}{I; \Gamma, a : nonce(I,J,M), \Gamma' \vdash \mathsf{run}(I,J,M).S : e}$$

Authentication

POSH COMMIT

$$\frac{J; \Gamma \vdash S : e \qquad \Gamma \vdash n, M, \widetilde{M} : Un \qquad \Gamma \vdash K : T \qquad dec\{\mathsf{Id}(Id), C(n), \mathsf{Auth}(M), \widetilde{M}\}_K \in e \qquad T, C, Id \text{ assumono i valori in } POSH(I,J)}{J; \Gamma \vdash \mathsf{commit}(J,I,M).S : e + [fresh(n)]}$$

SOPH/SOSH COMMIT

$$\frac{J; \Gamma \vdash S : e \qquad \Gamma \vdash n : nonce(I,J,M) \qquad in(n) \in e \text{ or } dec\{n, \widetilde{M}\}_K \in e}{J; \Gamma \vdash \mathsf{commit}(J,I,M).S : e + [fresh(n,I,J,M)]}$$

- (SOPH/SOSH COMMIT) consente a J, una volta avuto indietro il nonce n, di verificarne la *freshness* e conseguentemente di autenticare I e $\mathcal{M}$. La rimozione dell'effetto $fresh(n, I, J, \mathcal{M})$ da e garantisce che il controllo del nonce non avvenga più di una volta.

2.4 Esempi di validazione

2.4.1 Analisi sintattica

In questo esempio si fa riferimento al protocollo codificato in T.2.2, che è attaccabile per *reflection* quando è avviato in sessioni multiple: difatti il processo let $k_{AB} = \mathsf{sym\text{-}key}(A, B).B \triangleright Responder(B, A)$ non è localmente corretto, affermazione facilmente verificabile dato che tutte le regole che validano la primitiva commit richiedono la precedente decifrazione di un messaggio dotato di tag, situazione che non può verificarsi in questo caso poiché il protocollo non presenta alcun tipo di tagging. Si ha quindi:

$$B; \Gamma; \Pi \nvdash commit(B, A).S$$

A evidenziare come l'errore sia di natura progettuale è il fatto che l'impossibilità di validare la commit persista aggiungendo un tag al nonce x:

$$\{x : \mathsf{Claim}, m\}_{k_{AB}}$$

difatti il problema del protocollo è la mancanza di un'etichetta che identifichi il mittente del messaggio, mancanza che impedisce l'applicazione dela regola appropriata (AUTHENTICATE CLAIM). Questo errore è corretto nel protocollo *ISO TWO-Steps Unilateral Autenthication*:

1. $B \rightarrow A: \ n_B$
2. $A \rightarrow B: \ \{A : \mathsf{Id}, n_B : \mathsf{Claim}, m\}_{k_{AB}}$

2.4.2 Sistema di tipi

In questo esempio si fa riferimento ad una versione leggermente modificata del protocollo SPLICE/AS:

1. $B \rightarrow A: \ B, n_B$
2. $A \rightarrow B: \ A, B, \{|B, n_B, \{|n_A|\}_{Pub(k_B)}|\}_{Priv(k_A)}$
3. $B \rightarrow A: \ \{|B, n_A|\}_{Pub(k_A)}$

L'obbiettivo di questo protocollo è la mutua autenticazione tra A e B: nei primi due messagi B inizia una sessione *POSH* per autenticare A, che risponde con una sifa cifrata per B avvalendsoi di uno schema *SOSH.* Il secondo messaggio presenta un errore:

1.a $B \rightarrow E: \; B, n_B$

1.b $E(B) \rightarrow A: \; B, n_B$

2.a $A \rightarrow E(B): \; A, B, \{|B, n_B, \{|n_A|\}_{Pub(k_B)}|\}_{Priv(k_A)}$

2.b $E \rightarrow B: \; E, B, \{|B, n_B, \{|n_A|\}_{Pub(k_B)}|\}_{Priv(k_E)}$

3.a $B \rightarrow E: \; \{|B, n_A|\}_{Pub(k_E)}$

3.b $E \rightarrow A: \; \{|B, n_A|\}_{Pub(k_A)}$

L'attacco è portato attraverso due sessioni parallele: E sfrutta la prima per impersonare B con A nella seconda, facendosi decifrare la componente $\{|n_A|\}_{Pub(k_B)}$. L'errore è evidenziato dalla mancata corrispondenza tra le asserzioni $\mathsf{commit}(A, B)$ e $\mathsf{run}(B, E)$:

1. $\mathsf{in}(B, x)$.
2. $\mathsf{new}(n_A).\mathsf{encrypt}\{n_A\}_{Pub(k_B)}$ as z.
   $\mathsf{encrypt}\{B, x, z\}_{Priv(k_A)}$ as $w.\mathsf{out}(A, B, w)$.
3. $\mathsf{in}(y).\mathsf{decrypt}\ y$ as $\{B, n_A\}_{Priv(k_A)}.\mathsf{commit}(A, B)$

Non esiste alcun tagging che permetta al sequential process illustrato sopra (corrispondente ad A) di portare a termine con successo la validazione: l'autenticazione tramite (SOPH/SOSH COMMIT) richiede che n_A abbia tipo *nonce*(B, A), il che implica che la relativa cifratura sia validata tramite (SOPH/SOSH INQUIRY). Poiché quest'ultima richiede che nel testo cifrato sia presente l'identità del mittente, il *type-checking* fallisce: basta allora aggiungere l'etichetta dell'identità per ottenere una versione corretta del protocollo:

2. $A \rightarrow B: \; A, B, \{|B, n_B, \{|\mathbf{A}, n_A|\}_{Pub(k_B)}|\}_{Priv(k_A)}$

A questo punto basta aggiungere il tagging appropriato per ottenere una versione del protocollo *sicura*:

1. $B \rightarrow A: \; B, n_B$
2. $A \rightarrow B: \; A, B, \{|B : \mathsf{Id}, n_B : \mathsf{Verif}, \{|A : \mathsf{Id}, n_A : \mathsf{Verif?}|\}_{Pub(k_B)}|\}_{Priv(k_A)}$
3. $B \rightarrow A: \; \{|B, n_A|\}_{Pub(k_A)}$

Capitolo 3

ProtOK: il tool

Questo tool è stato realizzato con l'obbiettivo di automatizzare il più possibile il processo di analisi statica e validazione descritto nel capitolo dedicato (→ §.2): a partire dalla specifica del protocollo in *ρ-spi* (→ §.2.2), per ogni istruzione si tenta di applicare con successo una tra le regole appartenenti all'insieme fornito al tool; se si riesce a validare tutte le istruzioni il protocollo è *localmente corretto*[1].

3.1 Linee guida

Al momento di iniziare la progettazione erano chiari principalmente due punti:

- l'algoritmo di validazione era già definito in modo sufficientemente dettagliato da consentire un'implementazione lineare e spedita;

- il sistema di regole basato sull'analisi sintattica era abbastanza stabile, ma il sistema di tipi era ancora in fase di definizione: quindi una delle esigenze prioritarie era la progettazione del tool in modo che fosse paramentrico rispetto al sistema di regole, così da consentire future modifiche; questa era in ogni caso una buona politica, dovendo comunque rappresentare le regole attraverso una struttura dati piuttosto articolata; certo, rendere il tool completamente parametrico imponeva alcuni vincoli, che saranno descritti dettagliatamente in §.3.2.

Si è deciso di utilizzare `Java` come linguaggio di sviluppo per le sue note proprietà di portabilità e sicurezza, confidando nel fatto che, non essendo l'algoritmo di validazione particolarmente impegnativo dal punto di vista computazionale, le richieste in termini di prestazioni non fossero così pressanti da dover ricorrere al `C++` o addirittura al `C`.

[1] Ovviamente il sistema di regole fornito dev'essere corretto perché la validazione abbia valore: a questo proposito si consiglia di leggere attentamente [BFM04-1] e [BFM04-2].

Una volta presa familiarità con la parte teorica sottostante, si è proceduto alla definizione di un'elementare struttura delle componenti: le parti fondamentali dovevano essere i parser, il nucleo di validazione, l'interfaccia grafica ed un supporto integrato per eventuali estensioni.

3.1.1 I parser

Ovviamente per ricevere in input la specifica di un protocollo ed il sistema di regole, il tool deve essere dotato di appositi parser: per quanto riguarda il *ρ-spi* è apparsa subito chiara l'esigenza di un *compiler compiler* sulla falsariga di `yacc`, che però producesse codice java. La scelta è ricaduta su `JavaCC` [jcc04], il tool ufficiale della comunità di sviluppatori di `java.net`, dopo un ballottaggio con `byacc` [Jam04], una versione di `yacc` modificata per produrre codice java su richiesta; il fattore principale della scelta è stato il supporto nativo per java, la maggiore integrabilità ed il precompilatore `JTree` che consente la creazione automatica di un albero sintattico minimale.

Per quanto riguarda il parsing del sistema di regole, si era inizialmente pensato di utilizzare `XML`: dopo una fase di studio si è arrivati ad un prototipo di schema e ad una prima stesura delle regole, ma la sintassi risultava eccessivamente complicata e ridondate (→ §.C.3); si è deciso perciò di ricorrere a `JavaCC` anche in questo caso, puntando su una sintassi *java-like*.

Modifiche alla sintassi dei protocolli

Inizialmente la sintassi del *ρ-spi* doveva rimanere virtualmente identica a quella descritta in §.2.2, successivamente ci si è resi conto che apportando alcune leggere modifiche si poteva ottenere una migliore efficienza del tool: l'algoritmo di validazione prevede che ogni istanziazione dei sequential process sia processata con i parametri della relativa chiamata.

Operativamente questo non ha alcuna necessità di accadere poiché, posto che i parametri attuali rispettino i tipi attesi, la correttezza locale non è minimamente influenzata dai nomi dei parametri:

$$Initiator(A, B) \text{ corretto } \Rightarrow Initiator(A', B') \text{ corretto}$$

basta quindi verificare che la chiamata sia semanticamente corretta imponendo una tipizzazione esplicita dei parametri, per poi limitarsi a validare le singole definizioni parametriche dei sequential process:

$$Initiator(A, B, k) \rightarrow A > Initiator(B : \mathrm{Id}, k : \text{sym-key}(A, B))$$

questa sintassi inoltre identifica univocamente l'operatore che esegue il sequential process. In appendice è presente una descrizione precisa della grammatica adottata (→ §.B.1).

3.1.2 Il nucleo di validazione

Il nucleo di validazione deve essenzialmente ricevere in input la specifica di un protocollo ed il sistema di regole. Si è deciso di associare permanentemente al nucleo un sistema di regole di default (→ §.A.1), che può essere cambiato in ogni momento: ogni volta che viene avviata la validazione di un protocollo si fa riferimento al sistema di regole corrente.

Per poter effettuare il backtracking è stato progettato un apposito gestore: non è altro che uno stack di stati; uno stato (*validation state*) contiene tutte le informazioni utili a dare un'istantanea della validazione al momento della creazione del punto di ritorno (*backtracking node*). Effettuando un backtrack, il nucleo di validazione si riporta nello stato presente in cima allo stack.

Poiché il sistema delle regole è parte integrante dell'algoritmo di validazione (ad un certo punto si deve verificare che la regola corrente sia valida), parte della logica dell'algoritmo è distribuita nelle classi che implementano l'albero sintattico del sistema di regole. Questa situazione impedisce di svincolare completamente il nucleo di validazione dall'implementazione dell'albero sintattico, perciò si è dovuto ricorrere ad una soluzione ibrida descritta dettagliatamete in §.5.2.4.

3.1.3 Interfaccia grafica

Inizialmente il progetto era stato pensato per poter essere utilizzato come applet, questo implicava l'utilizzo delle librerie grafiche `AWT`; con il complicarsi delle funzionalità implementate, si è preferito privilegiare l'intuitività dell'interfaccia e ricorrere alle componenti grafiche messe a disposizione dalle librerie `swing`, accantonando l'idea dell'applet. Attualmente il tool è avviabile in remoto come applicazione *Java Web Start*.

3.1.4 Eventuali estensioni

Il tool è stato pensato per poter essere esteso tramite un sistema di plugin il cui uso principale sarà illustrato in §.3.2.2; essenzialmente si ha la possiblità di aggiungere ulteriori funzionalità legate ai sistemi di regole.

In futuro il supporto dovrebbe essere allargato al modulo di input dei protocolli e delle regole, così da consentire l'utilizzo di parser e sintassi differenti, ad esempio lo schema `XML` inizialmente accantonato.

3.2 Regole di validazione

Come detto il tool è stato progettato per essere completamente parametrico rispetto al sistema di regole: il problema principale consiste nell'ideare una sintassi sufficientemente precisa da consentire una rappresentazione chiara e lineare dei sistemi di regole già esistenti, e al contempo abbastanza espressiva da non limitare eccessivamente la progettazione di nuovi sistemi di regole potenzialmente del tutto diversi.

Per prima cosa bisogna osservare che, essendo l'analisi sintattica ed il sistema di tipi due approcci discretamente differenti, il secondo aspetto del problema non si discosta eccessivamente dal primo, e questo fatto si è rivelato decisamente utile.

3.2.1 Sintassi di base

Partendo dagli aspetti comuni dei due sistemi si può definire una sintassi di base: entrambi i sistemi definiscono regole dotate di nome; inoltre ad ognuna sono associate l'identità dell'operatore della comunicazione, un insieme di condizioni di applicazione ed uno di eventuali azioni; infine ogni regola si applica ad una precisa istruzione, con un certo pattern di parametri. Ecco quindi come si può definire una generica regola:

```
rule RULE(I)
{
  { instr(p1, p2, p3) }

  conditions { /* definizione delle condizioni */ }

  actions { /* definizione delle eventuali azioni */ }
}
```

dove `RULE` è il nome della regola, `I` rappresenta l'identità dell'operatore[2], `instr` è il nome dell'istruzione cui applicare la regola, e i vari p_i sono i parametri dell'istruzione. I parametri e l'identità sono identificatori che vanno per prima cosa dichiarati nell'apposita sezione; i tipi assegnabili[3] sono quelli usati nell'esempio seguente:

```
declarations
{
  Principal I, J;
  Variable p, p1, p2;  // c'è differenza tra nomi e variabili
  Name p3;
  Key sk, ak;
}
```

come si è potuto vedere, è possibile inserire dei commenti con la classica sintassi `C/C++/Java`: la doppia barra `//` commenta tutto ciò che segue fino

[2]Questa identità viene messa in corrispondenza con quella effettiva dell'operatore che esegue il sequential process durante il pattern-matching (→ §.3.2.3).

[3]Hanno lo stesso significato che nel ρ-*spi* standard.

Tabella 3.1: Contenitori standard.

`SetContainer`	modella un insieme semplice
`OrdContainer`	modella un insieme ordinato
`MultisetContainer`	modella un multinsieme
`MapContainer`	modella un insieme di elementi a cui è associato un valore

al termine della riga; inoltre viene ignorato tutto ciò che è compreso tra `/*` e `*/`.

Contenitori
Un altro aspetto che i due sistemi condividono è che le condizioni e le azioni definite da entrambi implicano operazioni su *contenitori*. I contenitori vanno dichiarati nell'apposita sezione, che precede quella degli identificatori:

```
containers
{
  ContainerType c1;
  ContainerType c2;
}
```

`ContainerType` identifica il nome del tipo del contenitore, mentre `c1` è il nome del contenitore stesso. Esistono quattro tipi di container già forniti con il tool (→ T.3.1), mentre altri possono essere definiti esternamente (→ §.3.2.2).

Operazioni, condizioni e azioni
Tutti i contenitori mettono a disposizione un set di operazioni standard (→ T.3.2), che possono essere invocate all'interno di condizioni o azioni[4]:

```
rule RULE(I)
{
  { instr(p1, p2, p3) }

  conditions { /* definizione delle condizioni */ }

  actions
  {
    c1.add(p1);
```

[4]Tutte le operazioni che non ritornano esplicitamente un valore booleano, assumono valore `true`.

Tabella 3.2: Operazioni standard sui contenitori.

`add(p)`	inserisce il parametro `p` nel container
`add(p,val)`	inserisce il parametro `p` nel container, associandogli un valore `val`
`contains(p)`	ritorna `true` se l'elemento `p` è presente nel container
`delete(p)`	elimina l'elemento `p` dal container

```
    c1.add(p2);
    c1.delete(p3);
  }
}
```

La definizione delle condizioni è leggermente più articolata, perché una condizione è composta da una coppia ipotesi-tesi; l'ipotesi è opzionale, nel caso non sia specificata assume valore `true`. Una condizione si definisce così:

```
rule RULE(I)
{
  { instr(p1, p2, p3) }

  conditions
  {
    ipothesis(c1.contains(p1));
    thesis(c1.contains(p2) & !c1.contains(p3));

    thesis(c1.contains(p2));
  }

  actions { /* definizione delle azioni */ }
}
```

come si evince dall'esempio, è possibile combinare i risultati di operazioni diverse tramite i canonici operatori booleani `&` (and), `|` (or) e `!` (not); se tutte le condizioni ritornano `true` oppure se non ci sono condizioni definite, vengono eseguite le azioni.

Tag

Come è stato spiegato nel capitolo precedente (→ §.2.1), i *tag* sono una parte fondamentale dell'algoritmo di validazione: entrambi i sistemi di regole definiscono degli insiemi di tag, seppur differenti. La definizione dei tag è un'operazione molto delicata, perché influenza il comportamento dei

Tabella 3.3: Tipi di tag.

`Tag`	tipo di tag generico
`KeyTag`	tipo di tag per marcare chiavi
`TypeTag`	tipo di tag utilizzato per tipare gli identificatori
`NonceTag`	tipo di tag per marcare i nonce
`MessageTag`	tipo di tag per marcare i messaggi
`IdentityTag`	tipo di tag per marcare le identità
`CiphertextTag`	tipo di tag per marcare i messaggi cifrati

parser delle regole e dei protocolli, senza contare l'algoritmo di *tagging inference* (→ §.4). Un messaggio non può contenere più di un tag per tipo (→ T.3.3), inoltre tutti i tag generici definiti in una volta sola sono mutuamente esclusivi[5]:

```
tags
{
    IdentityTag id;
    Tag claim, verif;
}
```

la dichiarazione dei tag è l'ultima e segue quella degli identificatori. A questo punto è possibile utilizzare i tag definiti per aggiungere i tag agli identificatori:

```
rule RULE(I)
{
  { instr({p1:claim, p2:verif, p3}:sk) } // questo è un errore
}
```

l'esempio sopra è sbagliato, perché due tag definiti isieme, cioè mutualmente esclusivi, sono presenti nello stesso messaggio; sarebbe stato corretto se la definizione fosse stata:

```
tags
{
    IdentityTag id;
    Tag claim;
    Tag verif;
}
```

[5]Ignorare questi vincoli significa andare incontro a messaggi di errore di sintassi da parte dei parser.

Tipi di parametri
Un'istruzione può avere come parametri identificatori, semplici o con tag, messaggi e istruzioni. La sintassi di un messaggio prevede il nome del messaggio (opzionale), una serie di identificatori e una chiave di cifratura o decifratura:

```
// p1 è un messaggio cifrato con chiave simmetrica sk
p1 = {I:id, p2}:sk

// p2 è un messaggio cifrato con la parte pubblica di ak
p2 = {|p3|}:Pub(ak)

// un messaggio decifrato con la parte privata di ak
{|p3|}(:Priv(ak))
```

poiché le operazioni non sono altro che istruzioni definite su contenitori, si usano gli stessi tipi di parametri; si può quindi scrivere:

```
rule RULE(I)
{
  { instr({I:id, p1:claim, p2}:sk) }

  actions
  {
    c1.add(instr({I:id, p1:claim}:sk));
    c1.add(instr(p2));
    c2.add(p3);
  }
}
```

bisogna inoltre tenere presente che i parametri possono essere visti anche come contenitori, quindi si potrà, ad esempio, verificare il contenuto di un messaggio:

```
rule RULE(I)
{
  { instr(p2 = {I:id, p1:claim, p3}:sk) }

  conditions
  {
    thesis(p2.contains(p3));  // true
    thesis(p3.contains(p2));  // false
    thesis(p3.contains(p3));  // true
  }
}
```

Infine è possibile assegnare ad un identificatore un valore *testuale* con la seguente sintassi:

```
rule RULE(I)
{
  { instr(p) }

  actions
  {
    c1.add(p="identificatore")
  }
}
```

Validazione di un sequential process
Nella stesura di un sistema di regole va considerato che un sequential process ha una sua firma, tramite la quale sono dichiarate le identità e le chiavi, ed un'istruzione di chiusura, l'istruzione nulla:

$$A > \mathit{Initiator}(B : \mathrm{Id}, k : \text{sym-key}(A, B)) \ := \ \ldots \ .0$$

se non si fornisce una regola l'istruzione nulla, sarà impossibile portare a termine con successo una validazione; la definizione standard consigliata della *regola nulla* è:

```
rule NIL(I)
{
  { 0 }
}
```

come si può intuire, essa non fa assolutamente niente, poiché non sono presenti né condizioni né azioni: questa regola valida l'istruzione nulla incondizionatamente; ovviamente è possibile definirla a proprio piacimento.

È inoltre possibile specificare delle regole per validare le dichiarazioni della firma:

```
rule IDENTITY_DECL(I)
{
  { id(J) }
}

rule SYMMETRIC_KEY_DECL(I)
{
  { let(k = sym-key(I, J)) }
}
```

le *pseudo-istruzioni* `id` e `let` hanno appunto lo scopo di fornire un “accesso” alle dichiarazioni della firma[6]; le regole definite nell’esempio non fanno nulla e non sono obbligatorie, dato che la firma non è un’istruzione del protocollo, ma possono risultare utili nel caso uno voglia manipolare i contenitori in base alla firma del sequential process. Va tenuto presente che una volta definite delle condizioni la validazione può fallire già all’altezza della firma.

3.2.2 Estensioni

Quanto descritto fino adesso è stato progettato grazie alle analogie tra i due sistemi, analogie di carattere essenzialmente strutturale; è a livello operazionale che emergono delle differenze: il sistema di analisi sintattica tende a sfruttare le operazioni elementari, mentre il sistema di tipi necessita di funzioni più complesse per effettuare il type-checking.

Quasi subito è apparso chiaro che per fornire un supporto operazionale al più vasto insieme di sistemi di regole definibile, l’approccio ideale è predisporre il tool all’estensione tramite *plug-in*. L’estensibilità al momento lavora in due direzioni:

- possono essere forniti nuovi contenitori sotto forma di classi che riscrivano il comportamento delle operazioni di base[7] o ne forniscano di nuove;

- possono essere fornite nuove operazioni su contenitori definiti a parte, sotto forma di classi aggiuntive che implementino l’interfaccia generica `ExternalOperation` (→ §.D.2).

Le operazioni definite *dal contenitore* sono dette `native`, mentre quelle definite *esternamente* sono dette `external`.

Si può sfruttare un nuovo contenitore `NewContainer`[8] semplicemente usandone il nome nella dichiarazione:

```
containers
{
  NewContainer nc;

  /* ulteriori dichiarazioni */
}
```

a questo punto è possibile fare uso delle operazioni definite sul contenitore come illustrato precedentemente[9]:

[6] Bisogna tenere presente che `sym-key(A,B)` equivale a `sym-key(B,A)`.
[7] L’interfaccia `Container` (→ §.D.1) descrive in dettaglio il set di operazioni di base.
[8] Le classi aggiuntive devono appartenere al package `it.unive.dsi.protok.plugins`.
[9] Infatti i container forniti col tool sono anch’essi estensioni.

```
nc.natop(p)
```

Se invece si vogliono richiamare delle operazioni esterne su un contenitore, bisogna specificare che l'operazione va cercata esternamente; questo si può fare a tre livelli:

- *operation*, si avvisa che la singola operazione è esterna:

  ```
  actions
  {
    (external)c1.extop(p)
  }
  ```

- *condition clause*, si avvisa che tutte le operazioni della clausola della condizione sono esterne:

  ```
  conditions
  {
    ipothesis(c1.contains(p) & c2.contains(p));
    external thesis(c1.extop(p) & c2.extop(p));
  }
  ```

- *conditions/actions*, si avvisa che tutte le operazioni delle condizioni o azioni sono esterne:

  ```
  rule EXT_RULE(I)
  {
    { instr(p, p1) }

    external conditions
    {
      ipothesis (c1.extop(p1));
      thesis(c1.extop(p) & c2.extop(p));
    }

    external actions
    {
      c1.extop(p2);
      c2.extop(p2);
      (native)c2.add(p1);  // l'operazione è nativa
    }
  }
  ```

inoltre ovunque si può usare la direttiva `external`, è possibile specificare la direttiva complementare `native` con significato opposto.

3.2.3 Pattern-matching

Le regole vengono associate alle istruzioni della specifica del protocollo tramite *pattern-matching*. La sintassi *ρ-spi* viene tradotta in quella più "universale" delle regole, quindi è possibile far combaciare esattamente un'istruzione con la regola associata:

```
decrypt xT as {nA:verif, B:id, kAB}:kAT.
```

viene tradotto in

```
decrypt(xT={nA:verif, B:id, kAB}(:kAT))
```

quindi, definendo una regola apposita, è possibile gestire la `decrypt`:

```
rule A_DECRYPTION(I)
{
  { decrypt(p={J:id, p1:verif, p2}(:k1)) }

  actions { /* definizione delle azioni */ }
}
```

in questo caso, supponendo che `A` sia l'operatore che esegue il sequential process a cui l'istruzione appartiene, la mappa di sostituzione sarà:

```
[I=A, p=xT, p1=nA, J=B, p2=kAB, k1=kAT]
```

tutti gli identificatori della regola *liberi* vengono mappati su quelli nelle posizioni corrispondenti dell'istruzione, con un distinguo: l'ordine delle componenti di un messaggio non ha alcuna importanza per il pattern-matching.

Poiché può essere complicato prevedere il numero di identificatori senza tag all'interno di un messaggio, identificatori che spesso non hanno alcuna influenza sulla validazione del protocollo, è possibile utilizzare il costrutto `...` per fare matching con tutti i valori privi di tag:

```
rule A_MORE_GENERAL_DECRYPTION(I)
{
  { decrypt(p={J:id, p1:verif, ...}(:k1)) }

  actions { /* definizione delle azioni */ }
}
```

questa versione gestisce un numero decisamente maggiore di casi:

```
decrypt xT as {nA:verif, B:id, kAB, z}:kAT.
```

```
[I=A, p=xT, p1=nA, J=B, ...=[kAB, z], k1=kAT]
```

il costrutto `...` può essere utilizzato ovunque sia permesso un identificatore semplice.

Poiché si può essere del tutto disinteressati alla struttura del messaggio, cifrato o meno che sia, è stato definito il costrutto `:::` che fa matching con qualsiasi identificatore, sia semplice che con tag, e si usa in modo analogo al precedente:

```
rule THE_MOST_GENERAL_DECRYPTION(I)
{
  { decrypt(p={:::}(:k1)) }

  actions { /* definizione delle azioni */ }
}
```

che gestisce tutti i casi:

```
decrypt xT as {nA:verif, B:id, kAB, z}:kAT.

[I=A, p=xT, :::=[nA:verif, B:id, ...], ...=[kAB, z], k1=kAT]
```

Tutti i valori mappati vengono mantenuti al momento di passare alla valutazione delle condizioni; questo significa che tutti gli identificatori che possiedono già un'associazione nella mappa di sostituzione, assumono i rispettivi valori, gli altri sono soggetti ad *auto pattern-matching*, ovvero per ogni operazione si esamina il contenitore su cui è definita per controllare che non sia possibile effettuare qualche ulteriore associazione: il processo è strettamente sequenziale e prende in esame uno ad uno tutti gli elementi del contenitore. Poiché non è possibile definire una logica generale per questo processo, non essendo nota a priori la semantica dell'operazione corrente, si procede per tentativi (grazie al backtracking) fino a quando non si trova una soluzione soddisfacente o si esauriscono gli elementi del contenitore. Considerando l'esempio precedente si avrà:

```
rule THE_MOST_GENERAL_DECRYPTION(I)
{
  { decrypt(p={:::}(:k1)) }

  conditions { thesis(c1.contains(k1=sym-key(I,J))); }

  actions { c2.add(decrypt({:::}(:k1))); }
}
```

la mappa precedente porterà ad un'operazione così definita:

```
c1.contains(kAT=sym-key(A,J))

c1=[kBT=sym-key(B,T), kAT=sym-key(A,T)]
```

ora, ad un osservatore esterno può apparire chiaro che l'associazione mancante è `J=T`, ma il nucleo di validazione non ha alcuna idea del significato dell'operazione `contains`, perciò prova prima a fare matching con il primo elemento del contenitore, fallendo poiché dovrebbe inserire un'associazione `kAT=kBT` nonostante l'identificatore `kAT` non sia libero; si rivolge quindi al secondo elemento ottenendo l'associazione corretta `J=T`.

Se la valutazione delle condizioni ha successo, si procede ad eseguire le azioni: in questo caso le operazioni subiranno le sostituzioni definite dalla mappa, ma non ci sarà alcun pattern-matching aggiuntivo. Se lo si desidera, si possono sfruttare le condizioni per effettuare un auto pattern-matching, tramite l'operazione esterna (quindi applicabile su tutti i contenitori) `match`, che non influisce sulla valutazione delle condizioni perché ritorna sempre `true`. Nell'esempio si avrà:

```
rule THE_MOST_GENERAL_DECRYPTION(I)
{
  { decrypt(p={:::}(:k1)) }

  conditions
  {
    thesis(c1.contains(k1=sym-key(I,J)));
    external thesis(c2.match(n="identifier"));
  }

  actions
  {
    c2.set(n="identified");
    c2.add(decrypt({:::}(:k1)));
  }
}
```

se `c2` non contiene un elemento associabile a `n`, l'azione verrà eseguita utilizzando `n` come parametro attuale, altrimenti la mappa sarà aggiornata di conseguenza:

```
c2=[id="identifier"]

c2.set(id="identified")
c2.add(decrypt({nA:verif, B:id, kAB, z}(:kAT)))
```

3.2.4 Direttive

Per personalizzare ulteriormente il comportamento del tool sono state definite tre direttive: una diretta al pattern-matching, le altre due al gestore del backtracking.

`@ra` questa direttiva serve a resettare temporaneamente l'associazione del costrutto `...` nella mappa di sostituzione, il che può risultare utile in determinati contesti; si utilizza prima di una clausola di condizione e ha come *scope* l'intera clausola:

```
rule RULE(I)
{
  { instr(...) }

  conditions
  {
    ipothesis(c1.contains({...}:k1));
    @ra thesis(c2.contains(p={...}:k2));
  }

  actions { /* definizione delle azioni */
}
```

il primo costrutto sarà sostituito con il valore associato nella mappa, il secondo con l'eventuale risultato di un auto pattern-matching.

`@exist`, `@univ` queste direttive sono alternative e complementari: servono a modificare il comportamento standard del gestore del backtracking; normalmente il nucleo di validazione in presenza di una `ipothesis` applica un criterio di *quantificazione universale*, ovvero tenta in tutti i modi di soddisfare i controlli definiti dalla clausola; al contrario, in presenza di una `thesis` viene applicata la *quantificazione esistenziale*: basta un fallimento per decretare il fallimento della valutazione della clausola, e quindi quello di tutte le condizioni, che come si è detto sono in relazione di and logico. Questo comportamento normalmente è sensato, se per qualche motivo si volesse alterarlo è possibile specificare la direttiva desiderata prima della clausola:

```
conditions
{
  @exist ipothesis(c1.contains({...}:k1));
  @ra @univ thesis(c2.contains(p={...}:k2));
}
```

ovviamente assegnando alle clausole le loro direttive di default, il comprtamento del gestore del backtracking rimarrà immutato.

Come si può notare dall'esempio precedente, le direttive vanno espresse in un certo ordine:

```
[@ra] [@exist | @univ] condition_clause
```

Nota: in appendice si possono trovare le versioni definitive delle codifiche dell'analisi sintattica (→ §.C.2.1) e del sistema di tipi (→ §.C.2.2).

3.3 Funzionalità del tool

Il tool è stato pensato per costituire un semplice *IDE*: è un sistema multifinestra dotato di editor del codice sorgente, esecuzione e "debug" del codice, console di output. È possibile effettuare le operazioni più comuni sui file: creare, aprire, chiudere, salvare.

Tramite l'*editor* si può modificare la sorgente del protocollo, tenendo sempre sotto controllo la sintassi, verificata su richiesta con l'apposito comando. Esiste poi una finestra per la localizzazione dei warnings che possono essere generati dal parser ρ-*spi* in caso fallisca qualche controllo semantico.

È inoltre possibile, come ampiamente annunciato in precedenza, selezionare e visualizzare file contenenti la specifica di altri sistemi di regole, che possono essere forniti anche sotto forma di plug-in (→ §.A.3). Va tenuto presente che questa funzionalità è disponibile solamente quando non vi sono file sorgente aperti, infatti cambiare il sistema di regole può invalidare la sintassi del protocollo a causa dell'insieme dei tag definiti (→ §.3.2.1, *Tag*): se un protocollo non è sintatticamente valido secondo quel sistema di regole, probabilmente non è stato pensato per essere validato con quel sistema[10].

La modalità di visualizzazione primaria delle regole è in forma di albero: oltre che una maggiore chiarezza e comprensibilità, questo sistema fornisce la possibilità di evidenziare di volta in volta quale regola viene applicata dal nucleo di validazione. Infatti oltre alla modalità di validazione standard, in cui, dopo aver validato il protocollo, il tool presenta il risultato, esiste la modalità di *validazione passo per passo*: è in questa situazione che l'approccio multifinestra si giustifica maggiormente: la sorgente del protocollo non è più modificabile e viene selezionata l'istruzione corrente, l'albero delle regole mostra quale momento della validazione sia in corso, evidenziando le regole o le operazioni correnti, mentre in un'apposita finestra sono mostrati lo stato dei contenitori e le associazioni presenti nella mappa di sostituzione (→ §.3.2.3).

[10] Un protocollo senza tag è sempre sintatticamente corretto per quanto riguarda i sistemi di regole.

È inoltre sempre possibile ispezionare la console di output per ricostruire l'operato del nucleo di validazione,tramite i messaggi del sistema di log che permette di impostare il livello di verbosità del tool (→ §.3.3.2). Poiché la verbosità decrementa le prestazioni, è presente un'opzione che permette al tool di adattare o meno la verbosità alla granularità dei passi di validazione (→ §.3.3.1, *Validazione passo per passo*).

Una delle funzionalità più interessanti presenti nel tool è il sistema di *tagging inference*, che sarà descritto dettagliatamente in §.4.

3.3.1 Il nucleo di validazione

Per prima cosa va detto che il nucleo di validazione è un modulo avviabile e può portare a termine una validazione autonomamente; il protocollo può essere inserito al momento da tastiera, oppure si può passare il nome di un file contenente la specifica del protocollo come parametro della linea di comando.

Come modulo il nucleo mette a disposizione una serie di funzionalità: consente di effettuare il parsing di una sorgente di regole, per ottenere il corrispondente albero sintattico; la sorgente può essere quella di default, oppure può essere fornita esternamente. Permette inoltre di effettuare il parsing della specifica di un protocollo.

Oltre alla possibilità di validare l'intero protocollo si può avviare la validazione su un singolo sequential process, sulla firma del sequential process o su una singola istruzione.

Algoritmo di validazione

L'algoritmo di validazione (→ ∀.3.3.1) inizia esaminando la firma del primo sequential process, le cui dichiarazioni sono trattate come istruzioni autonome, per poi rivolgersi alla prima istruzione vera e propria.

La validazione di un'istruzione è divisa in due fasi: per prima cosa si verifica che ci siano regole applicabili, ovvero si ottiene dal sistema di regole l'insieme di quelle nella cui definizione compare il nome dell'istruzione corrente; in questa fase non vengono presi in considerazione i parametri:

```
rule AN_INPUT(I)
{
  { input(p1, p2) }
}

rule INPUT(I)
{
  { input(...) }
}
```

Tabella 3.4: Granularità dei passi di validazione.

straight on è la modalità utlizzata di default: la validazione procede senza interruzioni;

sequential process la validazione si ferma all'inizio di ogni sequential process;

instruction la validazione si ferma ad ogni istruzione;

rule la validazione si ferma all'applicazione di una nuova regola;

operation la validazione si ferma ad ogni operazione eseguita sui container.

```
rule GEN_INPUT(I)
{
  { input(:::) }
}
```

tutte e tre le regole sarebbero prese in considerazione, pur avendo un istruzione corrente del tipo:

`input(k:key).`

A questo punto si esamina la prima regola dell'insieme: si prova a fare pattern matching con l'istruzione secondo le regole descritte in §.3.2.3; se l'istruzione è applicabile, si ottiene una mappa di sostituzione con le relative associazioni e si verifica il rispetto della regola, valutando le condizioni (se presenti) ed eventualmente eseguendo le azioni definite.

Se l'applicazione della regola ha successo, si passa all'istruzione successiva, non prima di aver creato un nodo di backtracking qualora ci fossero ancora regole da provare. Se invece l'applicazione della regola fallisce, si verifica che non esistano nodi di backtracking dovuti all'auto pattern-matching: in caso affermetivo si effettua un backtrack, altrimenti si passa alla successiva regola dell'insieme. Se non ce ne sono si tenta un backtrack; se non ci sono nemmeno nodi di backtracking disponibili, la validazione fallisce.

Una volta validato un intero sequential process, lo stack del gestore di backtracking viene azzerato, infatti il concetto di *correttezza locale* si basa proprio sull'indipendenza dei singoli sequential process (→ §.2). Se tutti i sequential process vengono validati con successo, il protocollo è *localmente valido*.

Validazione passo per passo

Un'ultima caratteristica del nucleo è la capacità di suddividere la validazione in passi impostandone la *granularità* (→ T.3.4): il nucleo si fermerà in corrispondenza dello "snodo" specificato dal livello di granularità corrente, attendendo il permesso di proseguire; poiché quest'operazione è effettivamente bloccante (il thread della validazione entra in stato `sleep`), è necessario sfruttare questa caratteristica in un ambiente *multithreaded*. In questa situazione sarà possibile prelevare un'*istantanea* dello stato del nucleo, autorizzare la prosecuzione della validazione, imporre al nucleo di tornare all'istruzione precedente, oppure arrestare la validazione.

Algoritmo 3.3.1 *L'algoritmo di* validazione *verifica la correttezza locale del protocollo ricevuto in ingresso.*

```
→ input : protocol
→ output : valid

spindex := 1
state := get_current_state(btstack)
for seqproc := protocol[spindex] do
   valid := true
   if spindex = 1 then  validate_declaration(seqproc) end
   iindex := get_instr_index(state)
   for instruction := seqproc[iindex] do
      valid := false
      rules := get_matching_rules(get_name(instruction))
      rindex := get_rule_index(state, rules)
      for rule := rules[rindex] do
         substmap := pattern_match(instruction, rule)
         if not is_empty(substmap) then
            valid := evaluate_rule(rule)
            available:=auto_pmatch_available(btstack)
            if available and not valid then
               state := backtrack(btstack)
               rindex := get_rule_index(state, rules)
               continue
            end
            if rindex ≤ length(rules) then
               if not valid then continue end
               if not available then save(btstack, state) end
            end
            break
         end
      end
      if not valid then break end
   end
   if not valid and available(btstack) then
      state = backtrack(btstack)
      spindex := get_sp_index(state)
   end
end
return valid
```

3.3.2 Il sistema di log

Il sistema di log ha come funzione principale la regolazione della verbosità dell'output, ma non si limita a questo: poiché si occupa tra l'altro di stampare i messaggi di *errore* e di *warning*, ha anche la funzione non meno importante di collezionare questi messaggi e metterli a disposizione, insieme ad alcune utili informazioni aggiuntive, come la sorgente che ha originato il messaggio ed eventualmente le coordinate del punto della sorgente che ha generato il messaggio.

É inoltre possibile configurare il log (→ §.A.1) in modo che di volta in volta stampi o meno l'ora, oppure il nome della componente che sta utilizzando il sistema di log. Infine è possibile disabilitare i warning.

Capitolo 4

Il sistema di tagging inference

Come accennato in [BFM04-1], l'unica vera difficoltà insita nell'utilizzo della teoria di analisi statica su cui si basa questo progetto, è la traduzione della specifica del protocollo dalla comune rappresentazione a frecce alla più rigorosa sintassi ρ-*spi*; non tanto per la codifica del protocollo in sé, che risulta abbastanza elementare dato il ridotto numero di istruzioni disponibili, quanto per il dover individuare i ruoli dei partecipanti alla comunicazione e conseguentemente dover applicare i tag corrispondenti agli identificatori.

Coerentemente con la direzione delle linee progettuali, si è deciso di cercare di automatizzare il più possibile anche questa operazione. Attualmente in fase di pre-analisi è necessario solamente fornire i tipi dei nonce (solo per il sistema di tipi) ed etichettare le coppie `encryption-decryption`, per ottenere un tagging del protocollo corretto secondo il sistema di regole corrente.

4.1 Il problema

Considerata la normale specifica di un protocollo, ad esempio l'ormai classica *ISO Two-Steps Unilateral Authentication*:

1. $B \to A: \; n_B$

2. $A \to B: \; \{A, n_B, m\}_{k_{AB}}$

per poterla sottoporre a validazione bisogna innanzitutto codificarla in ρ-*spi*; questa è un'operazione relativamente semplice per un essere umano, basta avere chiaro il significato del protocollo originale; infatti già questo caso esaurisce il numero di primitive a disposizione, che hanno un significato decisamente immediato:

```
Protocol ISO_TwoSteps_Unilateral_Authentication_Protocol:

A>InitiatorISO(B:Id, kAB:sym-key(A, B)):=

  new(m).
  in(x).
  run(A, B).
  encrypt {A, x, m}:kAB as y.
  out(y).
  0

B>ResponderISO(A:Id, kAB:sym-key(A, B)):=

  new(nB).
  out(nB).
  in(y).
  decrypt y as {A, nB, z}:kAB.
  commit(B, A).
  0

start.
let kAB = sym-key(A, B).
(A>InitiatorISO(B, kAB) | B>ResponderISO(A, kAB))
```

Il problema è che questa codifica *ρ-spi*, seppur perfettamente corretta, non può essere validata né tramite l'analisi sintattica né tramite il sistema di tipi: vanno aggiunti i tag agli identificatori. In questo caso l'operazione è semplice perché ci sono solamente due *principal*, che si scambiano un solo messaggio cifrato, eppure per chi non conosca bene i sistemi di regole ed i relativi tag non è esattamente immediata[1]:

```
encrypt {A:id, x:claim, m}:kAB as y.

decrypt y as {A:id, nB:claim, z}:kAB.
```

in casi in cui intervenga un TTP o ci sia uno scambio di messaggi cifrati più numeroso, l'operazione può diventare complicata e condurre a errori nel tagging che minerebbero la corretta validazione del protocollo. Una *corretta* automatizzazione è quindi consigliabile per ridurre l'impegno umano, ma anche per assicurarsi che il risultato della validazione sia significativo.

[1] In tutto il capitolo si fa riferimento al sistema di regole di analisi sintattica presente in appendice (→ §.C.2.1).

4.2 La soluzione adottata

Per automatizzare il tagging del protocollo è ovviamente necessario l'aiuto del sistema di regole: è quest'ultimo che definisce i tag utilizzabili e ne stabilisce il tipo[2], quindi è chiaro che, essendo il tool parametrico rispetto alle regole, altrettanto parametrico deve essere il sistema di *tagging inference*.

Per capire come si può realizzare l'inference, basta pensare ad un normale processo di validazione: dove entrano in gioco i tag? Al momento del pattern-matching tra l'istruzione e la regola: se l'istruzione contiene identificatori dotati di tag, normalmente può sfruttare regole che altrimenti le sarebbero precluse e che in genere risultano fondamentali per il corretto svolgimento della validazione. Nell'esempio si ha l'istruzione

```
encrypt {A, x, m}:kAB as y.
```

cui può essere applicata solo[3]

```
rule ENCRYPTION(I)
{
  { encrypt(y={...}:d) }

  conditions { /* definizione delle condizioni */ }

  actions { pi.add(y=enc({...}:d)); }
}
```

mentre la regola corretta sarebbe:

```
rule CLAIMANT(A)
{
  { encrypt(y={A:id, x:claim, ...}:k) }

  conditions { /* definizione delle condizioni */ }

  actions { pi.add(y=enc({A:id, x:claim}:k)); }
}
```

Poiché tutte le regole che validano l'istruzione `commit` richiedono che nel contenitore `pi` sia presente la registrazione della decifrazione di un messaggio dotato di tag, e poiché la `decrypt` corrispondente all'`encrypt` dell'esempio non contiene tag, la validazione non può terminare con successo.

In cosa consiste il problema? Se fosse possibile sfruttare la regola *corretta* per farsi suggerire quali tag servano per portare a termine la validazione,

[2]In effetti il *tag typing* è stato introdotto proprio per implementare il processo di tagging inference.

[3]In realtà è applicabile anche `OWNER`, ma il problema rimane.

Tabella 4.1: Assegnamento dei tipi agli identificatori.

***firma del sequential process* -** gli identificatori presenti nella firma sono classificati come `IdentityTag` oppure `KeyTag` a seconda del tipo dichiarato

`run(A,B,M)`, `commit(A,B,M)` - il messaggio da autenticare `M` viene classificato come `MessageTag`

`encrypt {...}:k as y` - la chiave `k` viene classificata come `KeyTag`, mentre l'identificatore `y` come `CiphertextTag`

`decrypt y as {...}:k` - la chiave `k` viene classificata come `KeyTag`, mentre l'identificatore `y` come `CiphertextTag`

sarebbe possibile *effettuare al momento il tagging* dell'istruzione corrente e della sua *complementare*:

1. $CLAIMANT \rightarrow \{\text{id, claim}\}$
2. $\{A, x, m\}_{kAB} \rightarrow \{A : \text{id}, x : \text{claim}, m\}_{kAB}$
3. $\{A, n_B, z\}_{kAB} \rightarrow \{A : \text{id}, n_B : \text{claim}, z\}_{kAB}$

Per quanto riguarda l'individuazione della regola corretta si può fare affidamento sul backtracking, mentre, come sarà illustrato in §.4.3, il tagging "consapevole" di un'istruzione e l'individuazione della sua istruzione complementare sono due operazioni dipendenti, ovvero a partire dalla seconda si può effettuare la prima.

4.3 L'algoritmo

Come si è intuito da quanto detto finora, l'algoritmo di tagging inference ($\rightarrow$ ∀.4.3.3) è una versione modificata di quello di validazione, difatti è stato implementato estendendo la classe `ValidationCore`, che a sua volta implementa le funzionalità del nucleo ($\rightarrow$ §.3.3.1).

Il processo di tagging inference passa per una fase di preanalisi del protocollo: a tutti gli identificatori che compaiono nella specifica è assegnato un *tipo di tag* applicabile; questa operazione è indispensabile per effettuare il tagging di un'istruzione senza essere costretti a provare tutte le combinazioni. Non si possono utilizzare i tipi degli identificatori della regola, perché potrebbero differire da quelli dell'istruzione: nomi e variabili possono comparire nelle stesse posizioni, inoltre le chiavi possono essere dichiarate dinamicamente tramite la primitiva `new` e quindi figurare come nomi. In pratica non si avrebbe un'interpretazione univoca del ruolo degli identificatori.

La scelta dei tipi è effettuata in base alla posizione in cui compare l'identificatore: normalmente le istruzioni del *ρ-spi* forniscono un'indicazione univoca sul tipo dell'identificatore (→ T.4.1); in caso di conflitto viene assegnato il tipo più generico (`MessageTag` prevale su `CiphertextTag`), mentre tutti gli identificatori liberi sono classificati come `NonceTag`. Disponendo dell'*informazione sull'accoppiamento* tra istruzioni complementari `encrypt-decrypt`, è possibile incrociare i dati tra i rispettivi sequential process ed ottenere una tipizzazione degli identificatori completamente coerente con la loro semantica.

Le principali modifiche all'algoritmo di validazione sono state apportate al pattern-matching: inizialmente si verifica che l'istruzione non sia già stata sottoposta a tagging; in caso affermativo il vecchio tagging viene rimosso e viene applicato il successivo (se presente), altrimenti si tenta di *effettuare il tagging* dell'istruzione secondo l'algoritmo ∀.4.3.1.

Algoritmo 4.3.1 *L'algoritmo di* instruction tagging *riceve come parametri un'istruzione e una regola, restituendo i possibili tagging.*

$\rightarrow$ *input* : *instruction*, *rule*
$\rightarrow$ *output* : *taggings*

used_tags := *get_used_tags*(*rule*)
message := *get_message*(*instruction*)
for *each message component* **do**
  tag_type := *get_tag_type*(*component*)
  tags := *get_defined_tags*(*tag_type*)
  for *each defined tag* **do**
    if $tag \notin used\_tags$ **then** *skip to next iteration*
    $used\_tags := used\_tags \setminus \{tag\}$
    identifiers := *get_same_type_id*(*message*, *tag_type*)
    for *each identifier* **do**
      tagging := *clone*(*message*)
      $tagging[index(identifier)] := identifier : tag$
      $taggings := taggings \cup \{tagging\}$
      update_previous(*taggings*)
    end
  end
end
return *taggings*

In pratica, al momento di verificare se la regola è applicabile si controlla anche che possa fornire un tagging per l'istruzione corrente; riprendendo l'esempio precedente (→ §.4.2), il risultato dell'applicazione dell'algoritmo ∀.4.3.1 sarà:

$$taggings = \{\{A : id, x : claim, m\}_{k_{AB}}, \{A : id, x, m : claim\}_{k_{AB}}\}$$

i successivi passi di validazione escluderanno i tagging scorretti e premiaranno quelli corretti.

A questo punto è possibile effettuare il tagging dell'*istruzione complementare*; bisogna però precisare che l'algoritmo ∀.4.3.1 non viene applicato all'istruzione corrente se questa costituisce un *forward complement*: in pratica, non si effettua mai il tagging di un'istruzione il cui complemento è definito in un sequential process precedente, infatti questa operazione, modificando un elemento già esaminato, invaliderebbe il processo e costringerebbe a ricontrollare il sequential process modificato.

Con la politica adottata (*forward tagging*), se un forward complement non contiene elementi dotati di tag significa che al suo *backward complement* al momento è stato assegnato un tagging nullo. Questa situazione è illustrata dall'esempio:

```
encrypt {A, x, m}:kAB as y          (ENCRYPTION)
...
decrypt y as {A, nB, z}:kAB         (DECRYPTION)
commit(B,A)                         (ERRORE:BACKTRACK)
...
encrypt {A, x, m}:kAB as y          (CLAIMANT:TAGGING)
encrypt {A:id, x:claim, m}:kAB as y (CLAIMANT)
...
decrypt y as {A:id, nB:claim, z}:kAB (DECRYPTION)
commit(B,A)                         (AUTHENTICATE_CLAIM)
```

4.3.1 Condizioni di arresto

Come si è visto, anche il comportamento del gestore del backtracking è stato modificato, difatti ora esiste la possibilità di tornare ad un sequential process precedentemente esaminato e provare tagging differenti da prima. Questo fatto influenza le condizioni d'arresto: nel caso della validazione l'algoritmo si ferma appena si riesce a validare l'ultima istruzione dell'ultimo sequential process, ma nel caso della tagging inference ci possono essere più tagging del protocollo validi, quindi le condizioni d'arresto cambiano.

La condizione d'arresto ideale è che lo stack del gestore del backtracking sia *vuoto*, ma non è sempre detto che si verifichi: a livello teorico il processo richiede un numero finito di passi, quindi idealmente la condizione si verifica sempre, in pratica le specifiche dei protocolli possono comportare un numero di passi di backtrack talmente elevato da rendere il tempo d'attesa inaccettabile. Vanno quindi trovate delle condizioni di arresto alternative:

- si impone un limite al numero di passi di backtracking accettabili;
- si impone un limite al numero di tagging uguali accettabili prima d'intervenire sullo stack del gestore di backtracking.

La prima condizione è chiara, ma presenta un problema: come garantire che non vadano persi dei tagging? A livello empirico si è potuto sperimentare che i tagging corretti "emergono" relativamente presto, mentre ciò che allunga decisamente i tempi è il provare tutta la moltitudine di tagging scorretti. Sarà comunque descritto un algoritmo che permette di riorganizzare il codice del protocollo per ridurre al minimo i passi di backtracking ($\rightarrow \forall$.4.3.4).

La seconda condizione è più interessante: lo stack può contenere due tipi di nodi: quelli normali e quelli creati dall'algoritmo di tagging inference; questi si distinguono perché contengono l'informazione sul tagging dell'istruzione corrente al momento della creazione del nodo; sono quindi solo i nodi creati dall'inference ad avere importanza ai fini della terminazione. È stato perciò introdotto un controllo regolato dall'algoritmo $\forall$.4.3.2.

Algoritmo 4.3.2 *Questo algoritmo regola la* terminazione *del processo di tagging inference: riceve come parametri lo stack del gestore del backtracking, il limite di tagging uguali, il relativo contatore e il livello di profondità minimo raggiunto, ritornando un valore booleano che segnala se si sia raggiunto o meno il termine del processo.*
La funzione next_tagging_node *ritorna l'altezza nello stack del primo nodo con informazione sul tagging che riesce a trovare sotto il livello di altezza specificato come parametro.*

$\rightarrow$ *input* : *btstack*, *eqcounter*, *eqlimit*, *minlevel*
$\rightarrow$ *output* : *finished*

`if` $get\_level(btstack) = 0$ `then return` *true*
$finished := next\_tagging\_node(btstack, MAX\_INT) = 0$
`if` $eqcounter < eqlimit$ `then`
  $curlevel := get\_level(btstack)$
  $minlevel := min(minlevel, curlevel)$
`else`
  $eqcounter := 0$
  $minlevel := next\_tagging\_node(btstack, minlevel)$
  $finished := minlevel = 0$
`end`
`return` *finished*

In pratica, ogni volta che il contatore dei tagging uguali supera il limite, si scende sotto il livello minimo di altezza raggiunto fino a quel momento; in questo modo prima o poi si finisce per azzerare lo stack. Il quando dipende solo dal limite che si sceglie di imporre, dalla "pazienza" che si ha.

Finora si è parlato di tagging ripetuti, ma non si è mai spiegato il perché di questo fenomeno; in realtà è abbastanza semplice: per prima cosa si deve considerare che due o più regole potrebbero suggerire lo stesso tagging per una determinata istruzione; inoltre possono esserci normali nodi di backtracking subito prima del *tagging node* più recente: il fallimento di qualche regola prima della fine del processo di validazione li attiverebbe, potenzialmente generando un tagging identico all'ultimo accettato. In casi del genere l'unico rimedio è impostare il limite del contatore ad un valore abbastanza basso.

Algoritmo 4.3.3 *L'algoritmo di* tagging inference *calcola tutti i possibili tagging di un protocollo secondo il sistema di regole corrente; riceve in ingresso un protocollo ed il limite massimo di tagging uguali, ritornando i possibili tagging. Sfrutta svariati altri algoritmi:* compute_identifer_tag_types *calcola i tag assegnabili agli identificatori,* reorder_protocol *è l'algoritmo di riordinamento dei sequential process* (→ §.4.3.3), validate_protocol *è il normale algoritmo di validazione* ∀.3.3.1 *con le modifiche illustrate* in §.4.3 *nella fase di pattern matching, infine* is_finished *è l'algoritmo di terminazione* ∀.4.3.2.

$\rightarrow input : protocol, eqlimit$
$\rightarrow output : taggings$

$compute\_identifer\_tag\_types(protocol)$
$reorder\_protocol(protocol)$
$eqcounter := 0$
$minlevel := 0$
`repeat`
  `if` $validate\_protocol(protocol)$ `then`
    `if` $protocol \notin taggings$ `then`
      $taggings := taggings \cup \{clone(protocol)\}$
    `else`
      $eqcounter := eqcounter + 1$
    `end`
  `end`
  `if` $available(btstack) = 0$ `or` $finished$ `then break end`
  $finished := is\_finished(btstack, eqcounter, eqlimit, minlevel)$
  $state := get\_current\_state(btstack)$
`until` $finished$ `and` $no\ tagging\ info\ in\ state$
`return` $taggings$

4.3.2 Commit hints

Nella discussione finora non è mai stato affrontato un argomento fondamentale: cosa succede se una regola non è abbastanza informativa da suggerire i tag necessari per portare a buon fine la validazione? Un esempio chiarirà lo scenario: si supponga che la specifica del solito protocollo presenti le definizioni dei sequential process invertite:

```
B>ResponderISO(A:Id, kAB:sym-key(A, B)):=

  new(nB).
  out(nB).
  in(y).
  decrypt y as {A, nB, z}:kAB.
  commit(B, A).
  0

A>InitiatorISO(B:Id, kAB:sym-key(A, B)):=

  new(m).
  in(x).
  run(A, B).
  encrypt {A, x, m}:kAB as y.
  out(y).
  0
```

La validazione raggiungerebbe la `commit` senza ricevere nessun indizio sul tagging da applicare alla `decrypt`, e quindi fallirebbe miseramente, pur esistendo un tagging valido per il protocollo (come è stato ampiamente illustrato). Questo accade perché la `DECRYPTION` è così definita:

```
rule DECRYPTION(I)
{
  { decrypt(y={:::}(:d)) }
  actions { pi.add(y=dec({:::}(:d))); }
}
```

ovvero inserisce in `pi` qualsiasi cosa senza preoccuparsi del contenuto. Dove reperire quindi le informazioni necessarie? Chiaramente se è la `commit` lo sbarramento che impedisce alla validazione di essere portata a termine, devono esserci nelle regole che la validano tutte le informazioni necessarie a superare il controllo; queste informazioni sono dette *commit hints*.

I commit hints vengono raccolti al momento del parsing del sistema di regole; si verifica inoltre se esistono regole definite per la `decrypt` che contengano tag; in caso affermativo il tool permette di disabilitare i commit hints, altrimenti questi risultano indispensabili per la riuscita del processo di tagging inference.

Definizione 4.3.1 *Sia c un messaggio cifrato e $k = key(c)$ la sua chiave. Allora si definisce la sua* chiave complementare $\bar{k}$ *nel seguente modo:*

- $\bar{k} = k$, *se k è una chiave simmetrica;*
- $\bar{k} = Priv(k_a)$, *se $k = Pub(k_a)$ e k_a chiave asimmetrica;*
- $\bar{k} = Pub(k_a)$, *se $k = Priv(k_a)$ e k_a chiave asimmetrica.*

Definizione 4.3.2 *Sia k una chiave, allora la funzione $type(k)$ è così definita:*

- $type(k) =$ `SymKey`, *se k è una chiave simmetrica;*
- $type(k) =$ `AsymKey`, *se k è una coppia di chiavi (pubblica, privata);*
- $type(k) =$ `PrivAsymKey`, *se k è una chiave privata;*
- $type(k) =$ `PubAsymKey`, *se k è una chiave pubblica;*

Definizione 4.3.3 *Data una regola $R \in S_R$, sistema delle regole corrente, definita su un'istruzione di nome* `commit`, *sia $Cond_R$ l'insieme delle condizioni definite su R, allora per ogni operazione $o \in Cond_R$ il cui nome inizia con* `dec` *e il cui argomento è un messaggio cifrato c, si ha l'insieme di tag $I_t = \{t : t \in c\}$. In questo caso si dice* commit hint *la coppia*

$$(I_t, type(\bar{k}))$$

dove $\bar{k} = compl(key(c))$ è la chiave complementare della chiave di c. Si dice inoltre $C(R)$ l'insieme di tutti i commit hint C definibili su R.

I *commit hints* definiti nel sistema di regole corrente S_R appartengono quindi all'insieme:

$$C_H = \cup_{R \in S_R} C(R)$$

I commit hints servono quando, al momento di validare una `decrypt`, ci si trova ad applicare una regola che non contiene tag; in questo caso il primo tagging proposto è sempre quello nullo (l'istruzione rimane immutata), successivamente saranno provati gli altri tagging. Nell'esempio corrente i commit hints saranno:

$$C_H = \{(\{\mathrm{id}, \mathrm{owner}, \mathrm{key}\}, t_k), (\{\mathrm{id}, \mathrm{verif}\}, t_k), (\{\mathrm{id}, \mathrm{claim}\}, t_k)\}$$

con $t_k =$ `SymKey`; tra gli elementi appartenenti a C_H il primo insieme non è applicabile, quindi $\forall$.4.3.1 ritornerà:

$$taggings = \left\{ \begin{array}{l} \{A, n_B, z\}_{k_{AB}}, \\ \{A : \mathrm{id}, n_B, z : \mathrm{verif}\}_{k_{AB}}, \\ \{A : \mathrm{id}, n_B : \mathrm{verif}, z\}_{k_{AB}}, \\ \{A : \mathrm{id}, n_B, z : \mathrm{claim}\}_{k_{AB}}, \\ \{A : \mathrm{id}, n_B : \mathrm{claim}, z\}_{k_{AB}} \end{array} \right\}$$

tra questi solo il quinto elemento può soddisfare la successiva `commit` e completare con successo la validazione: il risultato finale è quello atteso.

4.3.3 Riorganizzazione dei protocolli

Nel problema descritto precedentemente (→ §.4.3.2) si accennava ad un'inversione dei sequential process: questo scenario è realistico poiché non è imposto un particolare ordine sulla relativa definizione. Inoltre va tenuto presente che riordinando le definizione dei sequential process secondo precisi criteri, è possibile ridurre il numero di passi di backtrack necessari a completare il processo, migliorandone le prestazioni. I fattori determinanti in questo senso, possono essere così descritti:

- *commit*, la presenza di una `commit` dopo una `decrypt` cui è applicabile un tagging, è indice di uno snodo fondamentale per l'algoritmo di inference: normalmente è la commit quella che pone più difficoltà per la validazione, quindi "affrontandola" il prima possibile si riducono sensibilmente i passi di backtrack necessari:

```
decrypt y as {A, nB, z}:kAB         (COMMIT_HNT:TAGGING)
decrypt y as {A, nB, z}:kAB         (DECRYPTION)
...
commit(B,A)                         (ERRORE:BACKTRACK)
...
decrypt y as {A:id, nB:claim, z}:kAB (DECRYPTION)
...
commit(B,A)                         (AUTHENTICATE_CLAIM)
```

- *distanza*, la distanza tra `decrypt` e `commit` è un altro fattore da tenere in considerazione: più queste sono vicine, meno passi di validazione ed eventualmente di backtracking sono necessari per validare la `commit`:

```
decrypt y as {A, nB, z}:kAB         (COMMIT_HNT:TAGGING)
decrypt y as {A, nB, z}:kAB         (DECRYPTION)
commit(B,A)                         (ERRORE:BACKTRACK)
decrypt y as {A:id, nB:claim, z}:kAB (DECRYPTION)
commit(B,A)                         (AUTHENTICATE_CLAIM)
```

- *informatività*, se un testo cifrato ha meno elementi del numero minimo di tag applicabili col sistema di regole corrente, è chiaro che non potrà avere un tagging: l'istruzione cui appartene quindi non influenza particolarmente il processo di inference; al contrario, se il numero di elementi è maggiore o uguale al minimo, l'istruzione con ogni probabilità subirà un tagging, vanno quindi privilegiate le istruzioni informative:

```
encrypt {A}:kAB as y               (ENCRYPTION)

encrypt {A, x, m}:kAB as y         (CLAIMANT:TAGGING)
encrypt {A:id, x:claim, m}:kAB as y (CLAIMANT)
```

Tabella 4.2: Risultati dell'algoritmo in millisecondi.

	ABT	ATB	BAT	BTA	TAB	TBA
ANS Shared Key	13369	28811	11006	**5298**	40539	18707
Amended SPLICE-AS	1693	-	**1682**	-	-	-
F Woo-Lam	7531	6249	**2534**	3074	6720	6380
Flawed	591	-	**330**	-	-	-
Flawed Woo-Lam	2043	1482	**2053**	2173	2654	2904
ISO Two-Pass	1382	-	**1322**	-	-	-
ISO Two-Steps	1332	-	**1282**	-	-	-
Wide Mouthed Frog	10415	7140	**3795**	4026	14401	9814
Simplified Woo-Lam	3815	2413	**1772**	1873	3915	3084

In grassetto sono evidenziati i valori individuati dall'algoritmo ∀.4.3.4.

- *molteplicità*, riprendendo il caso precedente, se un testo cifrato ha *esattamente* il numero di elementi previsti dalla regola corrispondente, il tagging (se effettuabile) sarà univoco; al contrario l'aumentare del numero degli elementi può aumentare il numero di possibili tagging, e di conseguenza il numero di iterazioni del processo di inference; è meglio ritardare il più possibile l'analisi di queste istruzioni, in modo da ridurre al minimo la lunghezza della parte di protocollo su cui ripetere il processo di validazione:

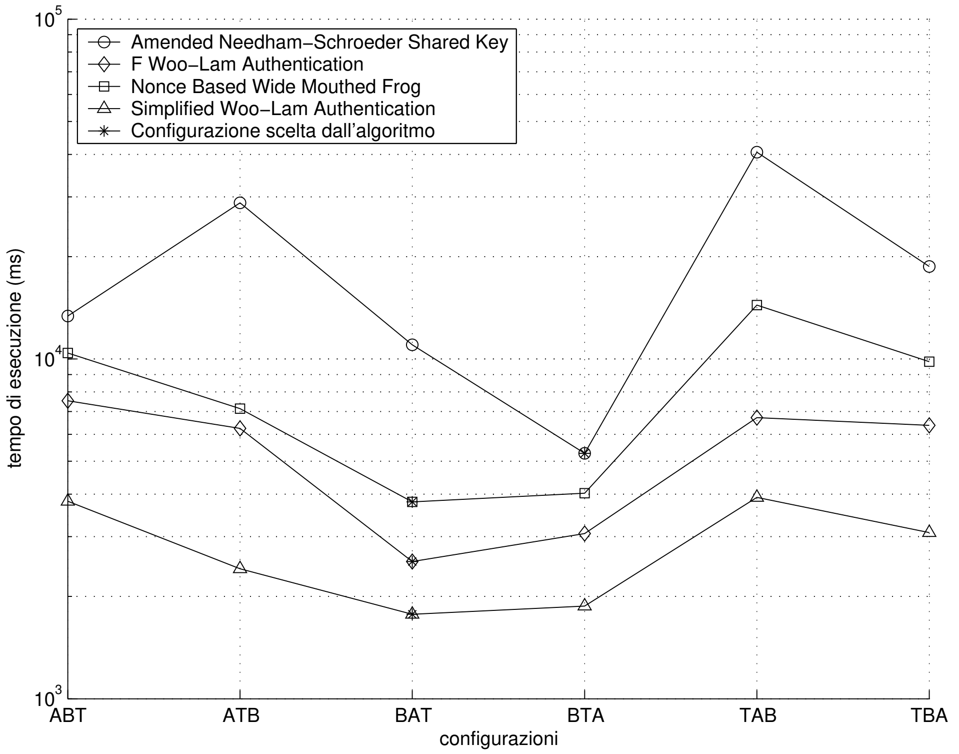
```
encrypt {A, x}:kAB as y            (CLAIMANT:TAGGING)
encrypt {A:id, nB:claim}:kAB as y  (CLAIMANT)

encrypt {A, x, m}:kAB as y         (CLAIMANT:TAGGING)
encrypt {A:id, x, m:claim}:kAB as y (CLAIMANT)
...                                 (ERRORE:BACKTRACK)
encrypt {A:id, x:claim, m}:kAB as y (CLAIMANT:TAGGING)
```

- *complementarità*, normalmente è conveniente che due sequential process collegati da istruzioni complementari vengano analizzati in successione, in modo che il tagging del backward complement possa al più presto essere sottoposto alla controprova della validazione del forward complement; occorre però tenere presente che, poiché le `decrypt` dipendono dalle `encrypt`, normalmente le regole più informative sono definite su quest'ultime: conviene quindi dare loro precedenza.

A partire da queste osservazioni è possibile definire un algoritmo di riordinamento delle definizioni dei sequential process: l'approccio adottato si articola in due fasi: nella prima (→ ∀.4.3.4) si fornisce un punteggio ad ogni

Figura 4.1: Andamento dei tempi rispetto alle configurazioni.



sequential process in base alla sua rispondenza alle caratteristiche elencate sopra, riordinando quindi i sequential process in base al punteggio ottenuto; nella seconda (→ ∀.4.3.6) si esamina l'ordine così ottenuto, cercando di riordinare le definizioni (se necessario) in modo tale da avere il maggior numero possibile di `encrypt` in corrispondenza dei *backward complement*. Concretamente la prima fase determina quale sia il miglior sequential process da cui iniziare la validazione[4], mentre la seconda determina l'ordine dei rimanenti sequential process, basandosi anche sui risultati della prima fase.

Questo algoritmo fornisce degli ottimi risultati[5]: nell'esperimento condotto sono stati provati tutti gli ordinamenti possibili dei protocolli a disposizione, ottenendo una tabella di tempi (→ T.4.2); l'algoritmo proposto individua invariabilmente l'ordinamento che minimizza il tempo d'esecuzione salvo in un caso in cui individua il terzo miglior tempo con uno scarto di pochi decimi dal primo.

---

[4] Questa costituisce una scelta fondamentale, come testimoniano i risultati riportati in figura (→ F.4.1).

[5] I tempi riportati in tabella (misurati in millisecondi) sono stati ottenuti su una macchina dotata di processore Pentium III 700Mhz e 384MB di RAM. Il tool è stato eseguito tramite il *Java Runtime Environment* 1.4.2 in ambiente *Windows*.

4.3.4 Risultati

Per quanto riguarda gli obbiettivi iniziali va detto che il sistema non è completamente autonomo, ma richiede l'intervento umano per la segnalazione delle coppie di istruzioni complementari: in pratica è necessario etichettarle con uno stesso identificatore all'interno del codice ρ-*spi*:

```
A>InitiatorISO(B:Id, kAB:sym-key(A, B)):=
  in(x).
  run(A, B).
1:encrypt {A, x}:kAB as y.
  out(y).
  0

B>ResponderISO(A:Id, kAB:sym-key(A, B)):=
  new(nB).
  out(nB).
  in(y).
1:decrypt y as {A, nB}:kAB.
  commit(B, A).
  0
```

Questo limite è dovuto principalmente alla natura del ρ-*spi* che, privilegiando l'analisi locale dei singoli sequential process, appiattisce la struttura della comunicazione rendendone difficile la ricostruzione automatizzata. Bisogna dire che, per motivi di tempo, questo problema non è stato studiato a fondo, quindi la sua soluzione potrebbe diventare uno degli obbiettivi per il lavoro futuro; non va comunque sopravvalutato l'impegno manuale necessario all'operazione di etichettamento.

Venendo ai risultati conseguiti, i test effettuati sembrano indicare la *correttezza* del sistema di tagging inference: in tutti i casi a disposizione ha individuato il tagging atteso, quando presente, permettendo inoltre di scoprire un bug nella codifica delle regole dell'analisi sintattica ($\rightarrow$ §.6.5).

Per quanto riguarda le prestazioni, sfruttando l'algoritmo $\forall$.4.3.4 i tempi rimangono ampiamente sotto la decina di secondi ($\rightarrow$ T.4.2), risultato che per un algoritmo esponenziale dovrebbe essere accettabile: il sistema è concretamente utilizzabile per lo scopo prefisso.

Algoritmo 4.3.4 *L'algoritmo di* riordinamento *riceve come parametri la definizione di un sequential process, il minimo e massimo numero di tag utilizzabili in un testo cifrato, ritornando il punteggio calcolato.*
Sfrutta inoltre delle costanti per assegnare i vari punteggi.

```
→ input : seqproc, mintag, maxtag
→ output : statistics

→ const : COMMIT_VALUE
→ const : DISTANCE_VALUE
→ const : TAGGABLE_VALUE

statistics := distance := 0
has_commit := taggable_decrypt := false
for each instruction ∈ seqproc do
  case get_name(instruction) of
  'commit' :
    if taggable_decrypt then
      statistics := statistics + COMMIT_VALUE
      has_commit := true
      if distance > 0 then
        distance := −1
        statistics := statistics − DISTANCE_VALUE
      end
    end
  'encrypt' : 'decrypt' :
    message := get_message(instruction)
    tagnumber := get_tag_number(message)
    if tagnumber ≥ mintag then
      statistics := statistics + TAGGABLE_VALUE
      if tagnumber > maxtag then
        statistics := statistics − (tagnumber − maxtag) end
      if is_decrypt(instruction) then
        taggable_decrypt := true
        distance := −1
      end
    end
    distance := distance + 1
  end
end
if has_commit then
  update_complement(seqproc, statistics) end
return statistics
```

Il comportamento di *update_complement* è descritto in ∀.4.3.5.

Algoritmo 4.3.5 *Questo algoritmo aggiorna il punteggio del sequential pocess complementare a quello ricevuto come parametro confrontandolo con quello ricevuto in ingresso.*

$\rightarrow$ *input* : *seqproc*, *statistics*
$\rightarrow$ *output* :

```
if has_complement(seqproc) then
   complement := get_complement(seqproc)
   compstat := get_statistics(complement)
   if compstat < statistics then
      set_statistics(complement, statistics)
      statistics := statistics + 1
   end
end
compstat := get_statistics(seqproc)
if compstat > statistics then statistics := compstat end
set_statistics(seqproc, statistics)
```

Algoritmo 4.3.6 *Questo algoritmo ricalcola i punteggi dei sequential process successivi al primo, privilegiando le* `encrypt` *in corrispondenza di un potenziale* backward complement.

```
→ input : seqprocesses, mintag
→ output :

→ const : COMMIT_VALUE
→ const : ENCRYPT_VALUE

max := 0
for each sequential process ∈ seqprocesses do
   statistics := get_statistics(seqproc)
   encrypt := 0
   if index = 1 then
      max := statistics
      statistics := statistics + COMMIT_VALUE
   end
   for each instruction ∈ seqproc do
      label := get_label(instruction)
      if index = 1 then
         labels := labels ∪ {label}
      else if is_encrypt(instruction) and label ∉ labels
         and get_tag_number(instruction) >= mintag then
         labels := labels ∪ {label}
         encrypt := encrypt + ENCRYPT_VALUE
      end
   end
   if encrypt > 0 then
      statistics := max + encrypt + length(seqprocesses) − index
   end
   set_statistics(seqproc, statistics)
end
```

Capitolo 5

Architettura del tool

Il tool è stato progettato in modo che risultasse il più modulare possibile, in modo da facilitare la gestione di eventuali estensioni e, più in generale, per garantire una buona *manutenibilità*.

Naturalmente la suddivisione in moduli e la relativa individuazione è partita dalla struttura dettata dalle linee guida descritte in §.3.1. Fondamentalmente il tool è diviso in otto moduli, più quelli relativi alle estensioni, che possono essere ricondotti più o meno direttamente alle componenti individuate in fase di preanalisi:

- `parser.jar`, `jjcparser.jar`, `rospi.jar` e `validationrules.jar` naturalmente svolgono le funzioni di parsing: il primo costituisce una generica interfaccia per il parsing di protocolli e regole di validazione, il secondo raccoglie componenti comuni ai parser generati da `JavaCC`, mentre gli ultimi due sono l'effettiva implementazione dei parser delle grammatiche definite; con questo approccio è possibile aggiungere con una certa facilità il supporto per nuove eventuali codifiche di protocolli e regole di validazione; al contrario i due parser dipendono completamente da `parser.jar` e `jjcparser.jar`, quindi i relativi moduli non sono autonomi;

- `core.jar` costituisce il modulo corrispondente al nucleo di validazione: è quasi completamente svincolabile dalle implementazioni dei parser generici contenute in `parser.jar`, fatta eccezione per alcune classi di enumerazione; questo modulo sfrutta le funzionalità messe a disposizione dalle interfacce presenti in `parser.jar` per effettuare la validazione di un protocollo rispetto ad un dato sistema di regole e l'inference dei possibili tagging di un protocollo rispetto al sistema di regole corrente; come già accennato (→ §.3.1.2), parte della logica dell'algoritmo di validazione è distribuita nell'implementazione delle classi dell'albero di sintassi delle regole di validazione: questo crea una dipendenza implicita che in teoria impedisce di prescindere dall'implementazione dell'albero sintattico delle regole di validazione; la soluzione adottata sarà descritta in §.5.2.4;

- `protok.jar` è il modulo di avvio, contenente le componenti dell'interfaccia grafica: fa uso delle interfacce definite nei moduli `core.jar` e `parser.jar` per implementare le funzionalità complete del tool: validazione passo per passo, creazione, modifica e controllo sintattico-semantico dei protocolli, importazione dei sistemi di regole, logging e così via; eccettuata l'istanziazione delle classi, è completamente indipendente dall'implementazione di `core.jar`;

- `log.jar` implementa il sistema di logging: è costituito da un'interfaccia comprendente un'implementazione minimale e dalla classe di log effettiva; tutti i moduli citati finora, tranne le implementazioni dei parser, dipendono da quest'ultimo, ma solo `protok.jar` istanzia effettivamente la classe d'implementazione;

- `util.jar` fornisce una serie di classi di supporto che sono utilizzate come classi della libreria standard da tutti i componenti del tool, estensioni escluse;

- `containers.jar`, `synt-analysis.jar` e `type-system.jar` sono moduli di estensione: il primo contiene le implementazioni dei contenitori definiti ed utilizzabili finora, mentre gli altri due contengono le implementazioni di operazioni esterne (→ §.3.2.2) specifiche rispettivamente del sistema di analisi sintattica e del sistema di tipi, oltre alle definizioni dei sistemi di regole corrispondenti.

Le dipendenze tra i moduli (→ F.5.1) sono decisamente lineari: tutti dipendono da `util.jar` come dalla libreria standard, mentre per il resto si ha una dipendenza di tipo *top-down* tra `protok.jar`, `core.jar` e `parser.jar`, ed una *bottom-up* tra le implementazioni dei parser ed i due moduli generici `parser.jar` e `jjcparser.jar`.

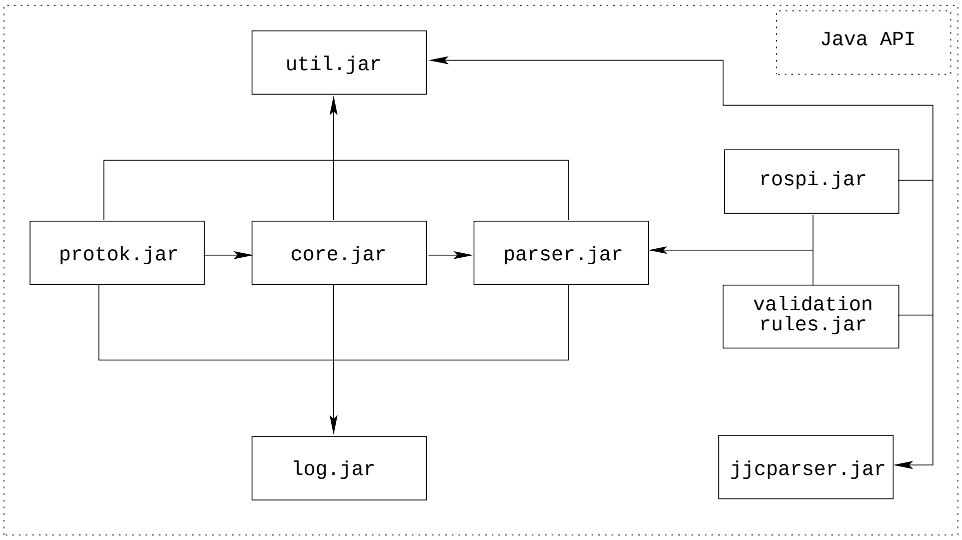
5.1 parser e jjcparser

La suddivisione in moduli si riflette sull'organizzazione in *package*: l'unione del contenuto dei moduli `parser.jar` e `jjcparser.jar` corrisponde a `it.unive.dsi.protok.core.parser`. Le classi appartenenti a questo modulo si possono dividere in quattro categorie fondamentali:

- *componenti dei parser*, utilizzate da entrambi i parser e, più in generale, da qualsiasi parser `JavaCC`;

- *strutture dati*, utilizzate per rappresentare gli alberi sintattici, le istruzioni ed i loro parametri;

Figura 5.1: Dipendenze dei moduli.



- *errori ed eccezioni*, utilizzate per la comunicazione degli errori sintattici o semantici: in particolare `SemanticError` è utilizzata per errori semantici considerati irrecuperabili, non "convertibili" in warning;

- *enumerazioni*, utilizzate per la trasmissione di informazioni tra i parser ed il livello superiore.

5.1.1 Componenti dei parser

La maggior parte delle classi del parser sono generate in automatico e non richiedono modifiche: `SimpleCharStream`, l'errore lessicale (considerato irrecuperabile) `TokenMgrError` [jcc04-1]; a queste si aggiungono le classi generate dal preprocessore `JJTree` [jcc04-2]: `Node` e `SimpleNode`, che invece sono state utilizzate come base per l'implementazione degli alberi sintattici e quindi modificate; è stato modificato anche l'errore sintattico `ParseException` con lo scopo di svincolare il parser generico dall'implementazione di `Token` tramite l'interfaccia `GenericToken`, conseguentemente lo stesso `Token` ha subito delle leggere modifiche.

È stata invece scritta manualmente la classe astratta `DefaultParser` che definisce le caratteristiche basilari del parser generico implementando l'interfaccia `Parser`; mette inoltre a disposizione una serie di operazioni utili per gestire i *warning*, generati dai parser tramite `SemanticException`, sfruttando l'interfaccia `Log.Message` del sistema di log. Queste operazioni costituiscono dei wrapper: al momento impediscono che le eccezioni siano passate ai livelli superiori "trasformandole" in warning, è comunque possibile modificare il comportamento del parser non gestendo le eccezioni che, essendo derivate da `RuntimeException`, non vanno dichiarate tramite la clausola `throws`.

Le eccezioni `SemanticException` sono sollevate principalmente dall'altra componente del parser generico: la classe `DefaultSymbolTable`; non è altro che una normale implementazione di `SymbolTable`, effettuata estendendo il contenitore standard `HashMap`, che fornisce alcuni semplici controlli semantici: permette di verificare se un identificatore sia già stato definito o meno, e fornisce un elementare *type-checking*; non è applicato alcun tipo di *scoping* ai nomi degli identificatori.

5.1.2 Alberi sintattici

Le classi che implementano gli alberi sintattici generici, si dividono in due gerarchie parallele: quella delle interfacce e quella delle effettive implementazioni. La prima ha come elemento di origine `it.unive.dsi.TreeNode`, una classe derivata dall'interfacccia `javax.swing.tree.TreeNode`, appartenente alla libreria grafica `Swing`, che definisce un nodo generico con i classici metodi di navigazione (per i dettagli si vedano le API java [sun04]); da questa derivano tutte le componenti degli alberi sintattici[1].

```
javax.swing.tree.TreeNode
  ▷ it.unive.dsi.protok.TreeNode
    ▷ it.unive.dsi.protok.core.parser.ProtocolNode
      ▷ it.unive.dsi.protok.core.parser.Protocol
      ▷ it.unive.dsi.protok.core.parser.SequentialProcess
      ▷ it.unive.dsi.protok.core.parser.InstructionNode
    ▷ it.unive.dsi.protok.core.parser.ValidationRulesNode
      ▷ it.unive.dsi.protok.core.parser.ValidationRules
      ▷ it.unive.dsi.protok.core.parser.Rule
      ▷ it.unive.dsi.protok.core.parser.Operations
        ▷ it.unive.dsi.protok.core.parser.Conditions
        ▷ it.unive.dsi.protok.core.parser.Condition
      ▷ it.unive.dsi.protok.core.parser.BoolNode
        ▷ it.unive.dsi.protok.core.parser.ConditionClause
        ▷ it.unive.dsi.protok.core.parser.AndCheck
        ▷ it.unive.dsi.protok.core.parser.OrCheck
        ▷ it.unive.dsi.protok.core.parser.NotCheck
      ▷ it.unive.dsi.protok.core.parser.Operation
```

Questa scelta è dovuta principalmente al fatto che così è possibile sfruttare la classe `javax.swing.tree.JTree` per visualizzare qualsiasi nodo, senza

[1] D'ora in avanti nella descrizione delle gerarchie di classi saranno adottate le seguenti convenzioni: ▷ significa *estende*, ◇ significa *implementa* oppure *è implementato da*, e ↑ significa *appartiene a*.

avere alcuna limitazione o controindicazione, poiché `TreeNode` definisce le stesse operazioni che avrebbero fatto parte di una generica interfaccia scritta da zero. In questo modo, a prescindere dall'effettiva implementazione dell'albero sintattico, si ottiene una struttura dati compatibile con quelle attuali anche a livello d'interfaccia grafica.

`Protocol`
L'interfaccia `Protocol` costituisce la rappresentazione dell'albero sintattico di un generico protocollo: fornisce delle funzionalità di navigazione avanzate, vale a dire la possibilità di scorrere le definizioni dei *sequential process* ed i *key binding* separatamente (→ §.2.2); permette inoltre di ottenere la rappresentazione sotto forma di stringa della sorgente del protocollo; dispone infine di alcuni metodi per la gestione dei tagging e di altri per il riordino delle definizioni dei sequential process.

`SequentialProcess`
Questa interfaccia ha un'impostazione analoga alla precedente: permette di scorrere distintamente istruzioni e dichiarazioni appartenenti alla firma del sequential process; dispone di metodi per la raccolta di altre informazioni utili associate al nodo, in particolare la *symbol table*, di cui esiste un'istanza per ogni sequential process poiché non è stato definito alcun meccanismo di scoping per i nomi degli identificatori. Tra i metodi di navigazione vale la pena di menzionare `getProtocol`, da preferire a `getParent` poiché non è detto che in una specifica implementazione il nodo corrispondente al protocollo sia genitore di quello del sequential process.

`InstructionNode`
Analogamente all'interfaccia precedente, `InstructionNode` fornisce un metodo `getSequentialProcess`, per la navigazione nella gerarchia logica; fornisce inoltre dei metodi per la gestione del nodo complementare ed altri per il trattamento della struttura dati rappresentante l'effettiva istruzione.

`ValidationRules`
Come nel caso delle interfacce dei protocolli, `ValidationRules` fornisce un sistema di navigazione avanzato: permette infatti di scorrere l'insieme di regole definite nel sistema. Fornisce inoltre metodi per la raccolta di tutte le definizioni collaterali del sistema di regole: identificatori, contenitori, tag e tipi dei tag definiti. Sono infine disponibili due metodi per il calcolo del minimo e massimo numero di elementi dotati di tag all'interno di un singolo testo cifrato.

`Rule`
L'interfaccia `Rule` non definisce alcuna funzionalità di navigazione aggiunti-

va: permette invece di accedere direttamente a tutte la caratteristiche della regola tramite i metodi corrispondenti.

`Operations`, `Conditions`, `Condition`
Lo scorrimento delle operazioni è demandato all'interfaccia `Operations`, che permette inoltre di ottenere l'impostazione corrente del `sourceType`, ovvero permette di determinare se a quel livello il nodo sia considerato `native` oppure `external` (→ §.3.2.2). `Conditions` aggiunge la possibilità di scorrere i nodi relativi alle singole condizioni. `Condition` invece, oltre alla possibilità di scorrere le singole clausole della condizione, fornisce dei metodi per il recupero dello stato corrente delle direttive (→ §.3.2.4).

`BoolNode`, `ConditionClause`, `AndCheck`, `OrCheck`, `NotCheck`
L'interfaccia `BoolNode` definisce un generico nodo a cui è associabile un valore booleano, permettendo di iterare tra gli operandi booleani che ne compongono l'espressione. Le clausole delle condizioni e gli operatori booleani devono implementare le corrispondenti interfacce in modo che i rispettivi nodi siano "riconosciuti" dal nucleo di validazione (→ §.5.2.4).

`Operation`
L'interfaccia `Operation` mette a disposizione tutte le informazioni associate al nodo corrispondente tramite i metodi corrispondenti, in particolare: il nome del contenitore, il nome dell'operazione, gli argomenti dell'operazione ed il `sourceType`.

La gerarchia delle implementazioni ha come radice la classe `SimpleNode`, un elemento specifico dell'architettura dei parser `JavaCC` che costituisce fondamentalmente un'implementazione minimale dell'interfaccia `Node`, sfruttata dal preprocessore `JJTree` per generare automaticamente l'albero sintattico di una sorgente durante il parsing. Potenzialmente `SimpleNode` è sufficiente per costruire l'intero albero, ma non per decorarlo adeguatamente: sono perciò presenti nei moduli dei parser delle sottoclassi riconoscibili dal prefisso `AST` nel nome che implementano la gerarchia delle interfacce. `SimpleNode` deriva a sua volta da una implementazione standard di `javax.swing.tree.TreeNode`, definita nel modulo `util.jar`.

```
it.unive.dsi.protok.util.DefaultTreeNode
  ▷ it.unive.dsi.protok.core.parser.SimpleNode
    ◇ it.unive.dsi.protok.core.parser.Node
    ▷ it.unive.dsi.protok.core.parser.rospi.RoSpiNode
      ▷ it.unive.dsi.protok.core.parser.rospi.AST*
    ▷ it...parser.validationrules.ValidationRulesNode
      ▷ it.unive.dsi.protok.core.parser.validationrules.AST*
```

5.1.3 Istruzioni e parametri

Le istruzioni ed i relativi parametri costituiscono una gerarchia articolata per riflettere la molteplicità di forme che possono assumere. Anche in questo caso la gerarchia si divide tra le interfacce e le relative implementazioni. Queste classi giocano un ruolo fondamentale, infatti nella gerarchia delle implementazioni è distribuita tutta la logica del pattern matching. Vale la pena di far notare che la radice logica della gerarchia delle interfacce è in realtà derivata da `Container`, l'interfaccia che modella un contenitore generico; in questo modo è possibile ispezionare il contenuto di un parametro come fosse un normale contenitore[2].

```
it.unive.dsi.protok.core.parser.Container
  ▷ it.unive.dsi.protok.core.parser.Data
    ▷ it.unive.dsi.protok.core.parser.TaggedData
    ▷ it.unive.dsi.protok.core.parser.Ciphertext
    ▷ it.unive.dsi.protok.core.parser.KeyInstance
    ▷ it.unive.dsi.protok.core.parser.Instruction
      ▷ it.unive.dsi.protok.core.parser.CipherInstruction
```

`Data`, `TaggedData`

La classe `Data` modella un parametro generico, che dispone di poche proprietà, tra cui la principale è il nome; altre permettono di stabilire se il parametro sia un semplice identificatore o qualcosa di più articolato, se il parametro contenga o meno elementi dotati di tag ed in tal caso quanti. Fornisce inoltre una serie di funzionalità per il pattern-matching: oltre al metodo dedicato, che permette di popolare una mappa di sostituzione, sono presenti due varianti del metodo di sostituzione: una semplice che a partire dalla mappa sostituisce gli elementi correnti con quelli associati, ed una automatica, che effettua anche il pattern-matching a partire da un altro elemento `Data` e da una mappdi sostituzione potenzialmente già popolata. Un ultimo metodo permette di ottenere il tipo della chiave associata al parametro, se presente.

L'interfaccia `TaggedData` aggiunge il supporto per il tagging: permette cioè di impostare il tag di un identificatore e di controllarne il valore.

`Ciphertext`

Questa interfaccia modella un messaggio (de)cifrato individuandone tre componenti: l'eventuale nome del messaggio, il testo del messaggio, la chiave di (de)cifratura:

```
{ d1, ..., dn } :k
```

[2]Ovviamente tutte le operazioni di modifica hanno implementazione nulla.

```
%
m= {|d1, ..., dn|} :Pub(k)
m= {|d1, ..., dn|} (:Priv(k))
```

Come si vede dall'esempio, si può definire del testo *astratto* all'interno del messaggio sfruttando il costrutto `...`; esiste quindi un metodo che permette di determinare se un messaggio sia o meno astratto. Sono inoltre disponibili delle funzionalità riguardanti la chiave: è possibile determinare se sia di cifratura o decifratura, si può inoltre verificare se corrisponde ad un determinato pattern; è disponibile anche un controllo per stabilire se la cifratura sia o meno simmetrica. Si pò infine ottenere l'insieme dei tag eventualmente associati agli elementi del messaggio.

`Instruction`, `CipherInstruction`
L'interfaccia `Instruction` modella la generica istruzione, dotata di nome e di un elenco di parametri; dipendendo da `Data` è essa stessa un parametro, quindi può comparire ovunque compaia un parametro semplice.
`CipherInstruction` è un'istruzione di (de)cifratura, contiene perciò come unico parametro un messaggio (de)cifrato; permette di ottenere l'istruzione complementare e di verificare se un'altra `CipherInstruction` sia o meno (complementarmente) compatibile:

```
encrypt(y={d1, d2}:k)
decrypt(x={m1, m2}(:d))
```

La gerarchia delle implementazioni è decisamente più articolata, in quanto rispecchia effettivamente tutte le forme che possono assumere i parametri. La radice della gerarchia è la classe astratta `RootData` che raccoglie le implementazioni delle funzionalità comuni a tutte le classi derivate.

```
it.unive.dsi.protok.core.parser.RootData
  ▷ it.unive.dsi.protok.core.parser.InstructionImpl
    ▷ it.unive.dsi.protok.core.parser.CipherInstructionImpl
      ▷ it.unive.dsi.protok.core.parser.Encrypt
      ▷ it.unive.dsi.protok.core.parser.Decrypt
  ▷ it.unive.dsi.protok.core.parser.SimpleData
    ▷ it.unive.dsi.protok.core.parser.AbstractData
      ▷ it.unive.dsi.protok.core.parser.AbstractTaggedData
    ▷ it.unive.dsi.protok.core.parser.AsymKeyData.KeyInstance
    ▷ it.unive.dsi.protok.core.parser.CompositeData
      ▷ it.unive.dsi.protok.core.parser.KeyData
        ▷ it.unive.dsi.protok.core.parser.AsymKeyData
    ▷ it.unive.dsi.protok.core.parser.TaggedDataImpl
      ▷ it.unive.dsi.protok.core.parser.ParametricTaggedData
    ▷ it.unive.dsi.protok.core.parser.ValueData
```

`AbstractData` è l'unica implementazione di un parametro da cui dipende direttamente il modulo `core.jar`, infatti, poiché il ruolo di questo elemento nella logica del pattern-matching è fisso, non si è voluto dare la possibilità di cambiarlo. D'altra parte vale la pena sottolineare che qualsiasi classe, posto che implementi l'interfaccia `Data`, è utilizzabile come parametro, quindi non esiste limite alle forme dei parametri che possono essere importati da un generico parser.

Le implementazioni descritte sopra si differenziano principalmente in due aspetti: la rappresentazione in forma di stringa e la logica di pattern-matching.

Algoritmo 5.1.1 *L'algoritmo di pattern matching di* `SimpleData` *riceve in ingresso il parametro, una lista di possibili matching e la mappa di sostituzione corrente, ritornando un valore che indica se il matching ha avuto successo.*

$\rightarrow$ *input* : $sd, paramlist, substmap$
$\rightarrow$ *output* : $matches$

```
if unmatchable then return false end
for each parameter ∈ paramlist do
  if already matched then
    if sd = param then
      paramlist := paramlist\{param}
      return true
    else skipto next parameter end
  end
  subst := get_subst(substmap)
  if subst = ∅ or is_abstract(subst) then
    unique := get_unique_identifiers(substmap)
    if param ∈ unique and param ∈ substmap then
      skipto next parameter end
    substmap := substmap ∪ {(sd, param)}
  else
    if subst ≠ param then return false end
    paramlist := paramlist\{param}
    return true
  end
end
return false
```

Sull'algortimo appena descritto si basano più o meno tutti gli algoritmi delle altre implementazioni: tendenzialmente viene percorsa la struttura del parametro fino a trovare i singoli identifcatori e su di essi si applica ∀.5.1.1. Ad esempio nel caso di `TaggedDataImpl` si verifica in aggiunta che il tag del parametro sia uguale a quello dell'elemento corrente della lista.

Se `SimpleData`, `TaggedDataImpl`, `InstructionImpl`, `CompositeData` e `AsymKeyData.KeyInstance` sono delle implementazioni abbastanza lineari di `Data`, `TaggedData`, `Instruction`, `Ciphertext` e `KeyInstance`, sono presenti anche classi che non sono caratterizzate da alcuna interfaccia specifica, poiché sono viste ad alto livello come semplici `Data`: `AbstractData` insieme a `AbstractTaggedData` aggiungono le funzionalità di pattern matching dei costrutti `...` e `:::`. Ci sono poi `KeyData` e `AsymKeyData` che modellano la dichiarazione di una chiave simmetrica o asimmetrica rispettivamente.

Infine `ParametricTaggedData` permette di aggiungere dei parametri ad un tag come fosse un'istruzione:

```
m:tag(d1,d2)
```

mentre `ValueData` permette di associare un valore stringa ad un messaggio:

```
m="string value"
```

5.1.4 Errori ed eccezioni

Le eccezioni e gli errori riportati dai parser si possono classificare essenzialmente in due categorie: recuperabili ed irrecuperabili. Tra i primi vi sono sia errori lessicali che semantici, mentre tra i secondi ricadono gli errori sintattici e la maggior parte di quelli semantici.

```
java.lang.Throwable
  ▷ java.lang.Error
    ▷ it.unive.dsi.protok.core.parser.TokenManagerError
    ▷ it.unive.dsi.protok.core.parser.SemanticError
  ▷ java.lang.Exception
    ▷ it.unive.dsi.protok.core.parser.ParseException
    ▷ java.lang.RuntimeException
      ▷ it.unive.dsi.protok.core.parser.SemanticException
```

L'errore lessicale è considerato irrecuperabile, perché una buona progettazione dei parser dovrebbe coprire tutti i casi possibili ed impedirne il manifestarsi [jcc04-3].

Il tipico errore semantico si verifica quando si tenta di effettuare il parsing di un protocollo che utilizza un insieme di tag differente da quello definito dal sistema di regole corrente (→ §.3.2.1), è irrecuperabile perché con ogni probabilità il sistema di regole corrente classificherebbe il protocollo come scorretto senza poterne verificare effettivamente la validità, risultando perciò completamente inadatto.

Al contrario l'errore di sintassi è considerato perfettamente "legale", tanto che il tool permette il controllo della sintassi durante la modifica della sorgente; i dettagli dell'errore sono presentati a video tramite la classe `LogError`, un wrapper per `ParseException` che implementa l'interfaccia `Log.Message`.

Come si è già spiegato in §.5.1.1, gli errori semantici sono normalmente convertiti in warning e sfruttano anch'essi `Log.Message` per la presentazione.

5.1.5 Enumerazioni

Le ultime componenti del package `it.unive.dsi.protok.core.parser` sono delle enumerazioni di valori: `EnumType` è una classe astratta utilizzata come base per elencare i tipi definiti da `SymbolTable`; i due parser utilizzano delle implementazioni complete di questa classe per istanziare la *symbol table*.

```
it.unive.dsi.protok.core.parser.EnumType
  ▷ it.unive.dsi.protok.core.parser.rospi.RoSpiType
  ▷ it.unive...parser.validationrules.ValidationRuleType
```

`OperationSourceType` fornisce i due valori che indicano se un nodo dell'albero sintattico delle regole di validazione ha tipo `NATIVE` oppure `EXTERNAL` (→ §.5.1.2, *Operations*).

Infine `KeyType` e `TagType` sono utilizzati dal sistema di tagging inference; la prima serve per determinare il tipo di una chiave ed il suo complementare, la seconda fornisce dei valori da assegnare come tipi ai tag.

5.2 core

Il modulo `core.jar` comprende due package: quello del nucleo di validazione `it.unive.dsi.protok.core` e quello dell'*evaluation tree*[3], descritto in §.5.2.4. Le classi all'interno del modulo possono essere suddivise in categorie come nel caso di `parser.jar`, ma risultano molto meno numerose:

- *componenti del nucleo*, sfruttano le funzionalità messe a disposizione da `parser.jar` per implementare l'algoritmo di validazione (→ ∀.3.3.1);

- *sistema di tagging inference*, sfrutta sia i moduli sottostanti sia il nucleo di validazione per implementare l'algoritmo ∀.4.3.3;

- *componenti di supporto per le estensioni*, utilizzate per garantire l'estensibilità del tool;

[3] `it.unive.dsi.protok.core.evaluationtree`.

- *errori e strutture dati*, utilizzate per la trasmissione di informazioni tra i moduli o in fase di elaborazione, come nel caso dell'evaluation tree.

5.2.1 Componenti del nucleo

Il nucleo è formato dall'interfaccia `Validation`, dalla relativa implementazione `ValidationCore`, dall'implementazione dello stato di validazione `ValidationState`, da `StepLevel`, l'enumerazione che fornisce i valori della granularità dei passi di validazione, e da `ValidationOutput`, una struttura dati utilizzata per memorizzare tutte le informazioni necessarie a visualizzare un'istantanea dello stato corrente del processo.

```
it.unive.dsi.protok.core.StepLevel
it.unive.dsi.protok.core.Parser
  ▷ it.unive.dsi.protok.core.Validation
    ◇ it.unive.dsi.protok.core.ValidationCore
      ◇ it.unive.dsi.protok.core.evaluationtree.Evaluator
it.unive.dsi.protok.core.ValidationOutput
it.unive.dsi.protok.util.BackTrackingHandler
  ↑ it.unive.dsi.protok.util.BackTrackingHandler.State
    ◇ it.unive.dsi.protok.core.ValidationState
```

Le funzionalità messe a disposizione da `Validation` si dividono in tre categorie: metodi per la regolazione dell'andamento del processo di validazione, metodi per il recupero delle informazioni sullo stato del nucleo, e metodi per la gestione del sistema di regole. `ValidationCore` è quindi composto di una serie di metodi per la validazione a vari livelli (protocollo, sequential process, istruzione, regola), una parte di metodi ausiliari per la gestione del backtracking e dello stato, dei metodi di support per il parsing di regole e protocolli, ed infine una serie di getter e setter che regolano le varie proprietà. Inoltre, trattandosi di un modulo avviabile, c'è un metodo `main` che consente di sfruttare autonomamente le funzionalità principali del sistema.

5.2.2 Sistema di tagging inference

Il sistema di tagging inference si compone essenzialmente dell'interfaccia `Inference` e della relativa implementazione `InferenceCore`, derivata dal nucleo di validazione.

```
it.unive.dsi.protok.core.Parser
  ▷ it.unive.dsi.protok.core.Validation
    ◇ it.unive.dsi.protok.core.ValidationCore
      ▷ it.unive.dsi.protok.core.InferenceCore
it.unive.dsi.protok.core.Inference
  ◇ it.unive.dsi.protok.core.InferenceCore
```

`Inference` mette a disposizione una serie di metodi per la regolazione delle opzioni del processo di tagging inference (→ §.4), in particolare: limite di tagging uguali, utilizzo dei commit hints e del riordino dei protocolli; ci sono poi i metodi per l'avvio e l'arresto del processo e per la gestione dei sistemi di regole.

`InferenceCore` si divide tra la parte di supporto alle proprietà definite nell'interfaccia e la più articolata implementazione degli algoritmi descritti in §.4: in particolare è stato riscritto il comportamento del metodo di pattern matching tra istruzione e regola, e dei metodi di gestione dello stato di backtracking; ci sono poi i metodi ausiliari per il calcolo dei tipi degli identifcatori, e per la rimozione dei nodi di backtrack inutili (per non dire dannosi). Il supporto per i commit hints è gestito tramite una classe separata di livello package, appartenente sempre al sorgente di `InferenceCore`. Come in `ValidationCore` è infine definito un metodo di avvio `main`.

5.2.3 Supporto per le estensioni

Il supporto alle estensioni è realizzato attraverso una serie di interfacce e di classi astratte che definiscono le caratteristiche essenziali che devono avere le estensioni; c'è poi una classe di configurazione che funge da raccordo per tutto il sistema.

```
it.unive.dsi.protok.core.parser.Container
  ◇ it.unive.dsi.protok.core.AbstractContainer
it.unive.dsi.protok.core.ExternalOperation
  ◇ it.unive.dsi.protok.core.DefaultOperation
it.unive.dsi.protok.core.Configuration
java.lang.ClassLoader
  ▷ it.unive.dsi.protok.util.JarClassLoader
```

L'interfaccia `Container` definisce le operazioni minimale che deve mettere a disposizione un contenitore: aggiunta e rimozione di un elemento, verifica della presenza, recupero del valore eventualmente associato e auto pattern-matching (→ §.3.2.3) di un dato elemento sul contenuto. Deve inoltre esserci la possibilità di svuotare il contenitore, di ottenere una rappresentazione del contenuto sotto forma di `java.util.Collection`, con relativo iteratore, o array. Infine un contenitore deve avere un nome.

`AbstractContainer` è un'implementazione astratta di `Container`: è utilizzata come base per i contenitori forniti con il tool, ed è pensata per costituire la base di tutti i contenitori che potranno essere definiti in futuro. Chiunque desideri aggiungere un contenitore è quindi incoraggiato a derivarlo da `AbstractContainer`, sebbene sia sufficiente implementare `Container`.

```
it.unive.dsi.protok.core.parser.Container
  ◇ it.unive.dsi.protok.core.AbstractContainer
    ▷ it.unive.dsi.protok.plugins.OrdContainer
    ▷ it.unive.dsi.protok.plugins.SetContainer
    ▷ it.unive.dsi.protok.plugins.MapContainer
    ▷ it.unive.dsi.protok.plugins.MultisetContainer
```

Come è facile intuire, `ExternalOperation` definisce l'interfaccia standard di un'operazione esterna; si compone essenzialmente di un metodo di esecuzione dell'operazione ed uno per l'impostazione della symbol-table relativa al sequential process corrente. `DefaultOperation` è un'implementazione astratta da cui sono derivate tutte le operazioni esterne fornite col tool.

```
it.unive.dsi.protok.core.ExternalOperation
  ◇ it.unive.dsi.protok.core.DefaultOperation
    ▷ it.unive.dsi.protok.plugins.addEach
    ▷ it.unive.dsi.protok.plugins.addFreeVar
    ▷ it.unive.dsi.protok.plugins.checkType
    ▷ it.unive.dsi.protok.plugins.checkWellFormedess
    ▷ it.unive.dsi.protok.plugins.contentEquals
    ▷ it.unive.dsi.protok.plugins.fresh
    ▷ it.unive.dsi.protok.plugins.isLast
    ▷ it.unive.dsi.protok.plugins.isNext
    ▷ it.unive.dsi.protok.plugins.match
```

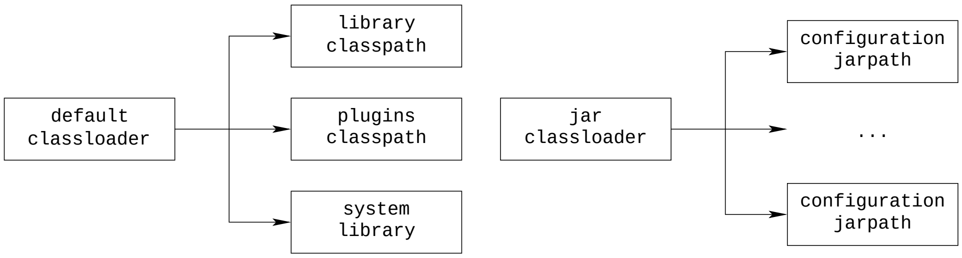
Ogni volta che si deve eseguire un'operazione esterna o si deve inizializzare un contenitore, ne va recuperata la classe: a questo scopo è stata definita la classe "statica" `Configuration`, che si occupa di scoprire l'effettiva locazione delle classi in base alla configurazione corrente all'invocazione del metodo apposito. A questo scopo utilizza `JarClassLoader` una classe derivata da `ClassLoader` utile per caricare le classi contenute in tutti moduli jar aggiuntivi che si riescono a reperire nei percorsi specificati dalla configurazione.

`JarClassLoader` funziona con il meccanismo di delega: quando gli viene richiesta una classe, prova prima ad interrogare il class loader di default, se questo non trova la classe richiesta, la cerca nella sua lista di classi caricate tramite i moduli jar aggiuntivi. Quando la classe richiesta è un plugin, normalmente entra in gioco `JarClassLoader`, difatti il class loader di default si occupa solamente delle classi reperibili nel *classpath*, cioè quelle presenti in:

```
$PLUGINS_DIR/it/unive/dsi/protok/plugins/
```

Per lo stesso motivo se si desidera caricare un sistema di regole *built-in*, o lo si colloca nel classpath, oppure lo si inserisce in un modulo jar e lo si carica tramite il `JarClassLoader` (→ F.5.2).

Figura 5.2: Funzionamento dei class loader.



5.2.4 Evaluation tree

L'albero di valutazione è la risposta alla necessità di svincolare l'implementazione dell'algoritmo di validazione da quella dell'albero sintattico del sistema di regole. Consiste in un'implementazione canonica di albero sintattico definita in `it.unive.dsi.protok.core.evaluationtree`: è costituita da una serie di nodi nei quali è distribuita parte della logica dell'algoritmo di validazione, in particolare la gestione delle direttive di backtracking (→ §.3.2.4) ed il backtracking a livello *operation*.

```
javax.swing.tree.TreeNode
  ▷ it.unive.dsi.protok.util.Node
    ◇ it.unive.dsi.protok.util.DefaultTreeNode
      ▷ it.unive.dsi.protok.core.evaluationtree.ETNode
        ◇ it.unive.dsi.protok.core.parser.ValidationRulesNode
        ▷ it.unive.dsi.protok.core.evaluationtree.ETBoolNode
          ◇ it.unive...core.evaluationtree.Evaluable
          ▷ it.unive...core.evaluationtree.ETAndCheck
          ▷ it.unive...core.evaluationtree.ETConditionClause
          ▷ it.unive...core.evaluationtree.ETNotCheck
          ▷ it.unive...core.evaluationtree.ETOpNode
            ▷ it.unive...core.evaluationtree.ETActions
            ▷ it.unive...core.evaluationtree.ETCondition
            ▷ it.unive...core.evaluationtree.ETConditions
            ▷ it.unive...core.evaluationtree.ETOperation
          ▷ it.unive...core.evaluationtree.ETOrCheck
        ▷ it.unive.dsi.protok.core.evaluationtree.ETRule
        ▷ it.unive...core.evaluationtree.ETValidationRules
```

Ogni nodo implementa la corrispondente interfaccia[4] definita nella gerarchia dell'albero sintattico delle regole (→ §.5.1.2).

[4] La corrispondenza si ottiene levando il prefisso `ET` dal nome del nodo; l'unica eccezione è `ETOpNode` che implementa `Operations`.

L'idea alla base dell'*evaluation tree* è la separazione di algoritmo e dati: dopo aver effettuato il parsing del sistema di regole, e aver ottenuto un albero sintattico di cui a livello teorico non è nota l'implementazione, si costruisce l'albero di valutazione ricalcando la forma di quello delle regole: ad ogni nodo dell'uno viene associato il nodo corrispondente dell'altro; così, al momento della valutazione, si prelevano i dati dall'albero sintattico ma si esegue il codice distribuito nell'implementazione dell'albero di valutazione. In questo modo è possibile mantenere fisso l'algoritmo di validazione pur essendo svincolati dall'effettiva implementazione dell'albero sintattico.

```
it.unive.dsi.protok.core.evaluationtree.EvalInfo
it.unive.dsi.protok.core.evaluationtree.Evaluable
  ▷ it.unive.dsi.protok.core.evaluationtree.Evaluator
javax.swing.tree.TreeNode
  ▷ it.unive.dsi.protok.core.parser.TreeNode
    ▷ it.unive.dsi.protok.core.parser.ValidationRules
      ▷ it.unive...core.evaluationtree.EvaluationTree
```

Ovviamente l'evaluation tree fornisce in più, rispetto all'albero sintattico, la possibilità di valutare una condizione, cioè di ottenerne il valore booleano associato. Questo viene effettuato tramite l'implementazione dell'interfaccia `Evaluable` da parte dei nodi booleani (tutti quelli "successivi" a `ETConditions`).

I nodi ricevono dal nucleo di validazione le informazioni di cui necessitano per la valutazione tramite la classe `EvalInfo`, tra cui l'*evaluator*, cioé la classe che si occupa effettivamente dell'esecuzione delle operazioni: l'interfaccia `Evaluator` mette a disposizione il metodo che ritorna il valore booleano restituito dall'escuzione dell'operazione oltre ad un metodo che consente ai componenti dell'evaluation tree di effettuare dei passi di backtrack; quest'ultima funzionalità è usata per implementare la quantificazione universale. Attualmente l'interfaccia `Evaluator` è implementata dal nucleo di validazione.

Infine l'interfaccia `EvaluationTree` fornisce un metodo per ottenere tutte le regole definite su un istruzione che abbia il nome specificato.

5.3 protok

Il modulo `protok.jar` raccoglie le componenti dell'interfaccia grafica presenti nel package `it.unive.dsi.protok`; sono stati definiti degli elementi particolari per rendere più comodo l'utilizzo del tool: le finestre possono essere "attaccate" (*docked*) ai quattro bordi del desktop principale del tool; nell'area del desktop che rimane libera è possibile usare le finestre normalmente, ridimensionandole e spostandole a piacimento. Ad ogni tipo di finestra corrisponde un apposito elemento: source editor, console di log, console degli errori e albero di visualizzazione delle regole.

```
javax.swing.JFrame
  ▷ it.unive.dsi.protok.Protok
javax.swing.JInternalFrame
  ▷ it.unive.dsi.protok.JDockableInternalFrame
javax.swing.JPanel
  ▷ it.unive.dsi.protok.JDockingDesktop
javax.swing.JTextArea
  ▷ it.unive.dsi.protok.JNotifyingTextArea
javax.swing.JPanel
  ▷ it.unive.dsi.protok.JProtocolSourceArea
javax.swing.JComboBox
  ▷ it.unive.dsi.protok.JTaggingBox
java.io.PrintStream
  ▷ it.unive.dsi.protok.JTextAreaStream
```

`Protok` contiene la maggior parte del codice dell'interfaccia grafica, comprende il sistema di gestione della validazione passo per passo[5], l'implementazione delle finestre di warning, di inference e delle regole di valdazione.

`JDockableInternalFrame` e `JDockingDesktop` realizzano il sistema di finestre *dockable*, cioè attaccabili ai bordi del desktop.

`JNotifyingTextArea` è una semplice implementazione di una area di testo che manda un segnale quando cambia il contenuto: è utilizzata per la console degli errori.

`JProtocolSourceArea` implementa l'editor della sorgente del protocollo, mettendo a disposizione tutte le funzionalità di source-checking, modifica ed evidenziazione del codice; dispone di un sistema di notifica delle modifiche al protocollo che può essere utilizzato anche esternamente: `Protok` ad esempio abilita il pulsante di salvataggio solo se il protocollo è stato modificato.

`JTaggingBox` è un elemento grafico che permette di selezionare il tagging desiderato oppure il protocollo originale: è sfruttato sia dalla finestra di inference che dal source-editor.

`JTextAreaStream` infine implementa la console di log: è derivato da `PrintStream`, quindi può essere utilzzato per redirezionare l'output del tool, come in effetti accade.

5.4 util

Come già accennato, il modulo `util.jar` mette a disposizione una serie di classi di utilità ed è assimilabile al package `java.util` della libreria stan-

[5]Che richiede un ambiente multithreaded (→ §.3.3.1).

dard. Alcuni dei componenti sono già stati descritti, altri sono sfruttati così diffusamente che non sarebbe stato possibile individuare un caso isolato.

Va premesso che quasi tutti i componenti del modulo, che appartengono al package `it.unive.dsi.protok.util`, implementano l'interfaccia `CloneableElement`; in questo modo le relative istanziazioni possono costituire i valori dei campi del generico stato di backtracking defìnto dall'interfaccia `BackTrackingHandler.State`. Questa impone che i valori siano clonabili cosicché l'implementazione dello stato di backtracking possa duplicarli al momento della creazione di un nodo: senza questo accorgimento i valori continuerebbero ad essere modificati pur essendo stati momentaneamente "accantonati". Ovviamente è compito di chi implementa le varie componenti fornire un'operazione di clonazione coerente con la logica della classe a cui si applica.

```
it.unive.dsi.protok.util.BackTrackingHandler
  ↑ it.unive.dsi.protok.util.BackTrackingHandler.State
  ↑ it.unive...util.BackTrackingHandler.LimitExcededException
java.lang.Cloneable
  ▷ it.unive.dsi.protok.util.CloneableElement
    ◇ it.unive.dsi.protok.util.CloneableIterator
      ◇ it.unive.dsi.protok.util.Iterator
      ▷ it.unive.dsi.protok.util.ArrayIterator
    ◇ it.unive.dsi.protok.util.CloneableList
      ◇ it.unive.dsi.protok.util.List
    ◇ it.unive.dsi.protok.util.CloneableMap
      ◇ it.unive.dsi.protok.util.Map
javax.swing.tree.TreeNode
  ▷ it.unive.dsi.protok.util.Node
    ◇ it.unive.dsi.protok.util.DefaultTreeNode
      ↑ it.unive.dsi.protok.util.DefaultTreeNode.NodeIterator
      ◇ it.unive.dsi.protok.util.CloneableElement
javax.swing.filechooser.FileFilter
  ▷ it.unive.dsi.protok.util.ExtensionBasedFileFilter
    ◇ java.io.FileFilter
java.lang.ClassLoader
  ▷ it.unive.dsi.protok.util.JarClassLoader
```

`BackTrackingHandler` è essenzialmente un wrapper per uno `Stack` di stati di backtracking; fornisce alcuni metodi per la creazione di nodi ed il relativo ripristino, mettendo a disposizione uno stato corrente che non appartiene allo stack, contenente solo nodi di backtrack. In particolare fornisce la possibilità di marcare i nodi di backtrack in modo che siano riconoscibili, e definisce operazioni applicabili a nodi con un marchio specifico. Permette infine di impostare un limite al numero di operazioni di backtrack effettuate superato il quale viene sollevata l'eccezione `LimitExcededException`.

`CloneableIterator` è una classe astratta che implementa l'operazione di clonazione semplicemente rendendo accessibile quella predefinita da `Object`; da essa deriva `ArrayIterator`, che fornisce la possibilità di scorrere gli elementi di un array.

`CloneableList` e `CloneableMap` sono delle implementazioni delle rispettive interfacce che abilitano esplicitamente il supporto per la clonazione; `CloneableMap` permette inoltre di scegliere se effettuate una clonazione *shallow*, cioè clonando solo i riferimenti agli elementi contenuti, oppure ricorsiva, richiamando l'operazione di clonazione dei singoli elementi, che in tal caso devono essere dei `CloneableElement`.

`DefaultTreeNode` costituisce un'implementazione abbastanza lineare di `TreeNode` (→ §.5.1.2): ai metodi di navigazione di base aggiunge il supporto per la creazione dei figli, la possibilità di ottenere una rappresentazione in formato stringa del sottoalbero di cui costituisce la radice, un'operazione di clonazione esplicita, e la possiblità di scorrere figli e discendenti in genere tramite la classe interna `NodeIterator`, che è a disposizione delle classi derivate.

`ExtensionBasedFilter` serve a filtrare degli elenchi dei file in base all'estensione e `JarClassLoader` permette di caricare le classi contenute nei moduli jar presenti nei percorsi indicati (→ §.5.2.3).

5.5 log

Il modulo `log.jar` è composto da due elementi: l'interfaccia `Log` e la relativa implementazione completa `DefaultLog`. `Log` mette a disposizione anche due implementazioni minimali sotto forma di classi interne: `VoidLog` che non stampa nulla e `SimpleLog` che stampa tutto.

```
it.unive.dsi.protok.log.Log
  ↑ it.unive.dsi.protok.log.Log.Message
  ↑ it.unive.dsi.protok.log.Log.VoidLog
    ▷ it.unive.dsi.protok.log.Log.SimpleLog
  ◇ it.unive.dsi.protok.log.DefaultLog
```

Essenzialmente `DefaultLog` permette di regolare la verbosità dell'output ricevuto tramie l'impostazione del livello di log corrente e l'informazione sul livello di log attribuito all'output da stampare: alcuni dei metodi utilizzabili definscono implicitamente il livello da attribuire all'output, mentre altri lo richiedono esplicitamente. È inoltre possibile regolare il formato degli header dell'output tramite proprietà di sistema, impostabili tramite la riga di comando. Infine `DefaultLog` tiene traccia dei *warning* emessi tramite una lista di `Log.Message`, che può essere consultata o azzerata a piacimento (→ §.5.1.1).

Capitolo 6

Casi di studio

In questo capitolo saranno illustrati degli esempi di esecuzione del tool: si è cercato di dare un quadro abbastanza completo delle operazioni effettuate, ma è comunque possibile trovare in allegato i log completi delle esecuzioni dei casi più significativi. Sempre in appendice sono presenti le codifiche complete dei sistemi di regole utilizzati dal tool, insieme ai sorgenti dei protocolli di cui ci si è servito per gli esempi di questo capitolo.

Le funzionalità principali del tool sono essenzialmente la validazione dei protocolli e la *tagging inference*: sarà quindi in questi ambiti che si concentreranno i casi di esecuzione illustrati nel prosieguo del capitolo. I primi due casi proposti mostrano la validazione dei protocolli usati come esempio nelle descrizioni dei due sistemi di regole presentati in §.2.3.2 e in §.2.3.3: sarà evidenziato soprattutto come in molti casi l'implementazione dell'algoritmo di validazione sia costretta ad agire in modo non deterministico.

Il caso successivo mostra la validazione del protocollo a chiave condivisa *Amended Needham-Schroeder*: sarà posto l'accento sull'impatto degli operatori di quantificazione sul processo di validazione, mostrando in particolare come abbiano permesso di affrontare una situazione decisamente intricata.

Sarà poi illustrato il normale utilizzo del sistema di tagging inference su una versione appositamente modificata del protocollo usato finora come esempio, l'*ISO Two-Steps Unilateral Authentication*: il processo sarà applicato al protocollo utilizzando entrambi i sistemi di regole a disposizione per poi confrontare i risultati.

Infine sarà descritto un caso in cui il sistema di tagging inference, individuando due possibili tagging (di cui uno non preventivato) per il protocollo *Nonce-Based Wide Mouthed Frog*, ha permesso di trovare un errore nella codifica delle regole di analisi sintattica.

Nell'ultima parte del capitolo saranno presentati i risultati ottenuti complessivamente tenendo conto degli obbiettivi iniziali, ed una valutazione del lavoro ancora da compiere.

6.1 Validazione: ISO Two-Steps

L'*ISO TWO-Steps* è il classico protocollo di autenticazione unilaterale basato su nonce; come ampiamente discusso in precedenza, il tagging necessario è nel messagio cifrato:

1. $B \rightarrow A: \ n_B$
2. $A \rightarrow B: \ \{A : \mathsf{Id}, n_B : \mathsf{Claim}, m\}_{k_{AB}}$

in appendice si può trovare la codifica completa in *ρ-spi* (→ §.B.2.3); per la validazione questo protocollo saranno utilizzate le regole di analisi sintattica (→ §.C.2.1).

L'esecuzione inizia col parsing del protocollo, durante la quale si verifica la corretta istanziazione dei sequential process:

```
let kAB = sym-key(A, B).
(A>InitiatorISO(B, kAB) | B>ResponderISO(A, kAB))
```

in questo caso i tipi dei parametri coincidono con quelli attesi, quindi non sarà emesso alcun *warning*; il processo di validazione inizia esaminando la firma del primo sequential process:

```
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      starting validation
[INF] [InitiatorISO] validating sequential process:
      InitiatorISO(B:Id, kAB = sym-key(A, B))
 ...
[INF] validating declaration: let(kAB = sym-key(A, B))
[INF] applying rule: SYM_KEY
[INF] performed: (native) pi.contains(kAB = sym-key(A, B))
[INF] rule conditions: evaluation successful
[INF] performed: (native) pi.add(kAB = sym-key(A, B))
[INF] rule actions: performing result = true
[INF] declaration validation successful
```

viene esaminata e validata le dichiarazione delle chiave simmetrica `kAB` tramite la regola (Symmetric Key), che richiede che la chiave non sia già stata definita in precedenza; si noti che la dichiarazione è validata tramite la pseudo-istruzione `let` (→ §.3.2.1, *validazione di un sequential process*). Arrivati a questo punto i contenitori hanno lo stato seguente:

```
gamma: []
pi:    [kAB = sym-key(A, B)]
```

il processo continua validando le istruzioni presenti nella definizione del sequential process:

```
[INF] -> validating instruction: new(m)
[INF] applying rule: NEW_NAME
[INF] performed: (external) gamma.fresh(m)
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.add(m = {"unchecked"})
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

l'istruzione new supera agevolmente i controlli definiti in (New Name), visto che non è ancora stato definito alcun nonce; la validazione procede analogamente applicando prima (Input) e quindi (Run), portando i contenitori nello stato seguente:

```
gamma: [m = {"unchecked"}]
pi:    [kAB = sym-key(A, B), B = {run()}]
```

al momento di validare l'encrypt il nucleo trova sette regole potenzialmente applicabili, prova quindi quella definita per prima:

```
[INF] -> validating instruction:
      encrypt {A:id, x:claim, m}:kAB as y
[INF] applying rule: OWNER
[INF] backtrack node here
[INF] performed: (native) pi.contains(kAB = sym-key(A, B))
[INF] performed: (native) pi.contains
      (any = {dec({B:id,n:owner,kAB:key, ...}(:kAB))})
[INF] rule conditions: evaluation failed
[INF] [OWNER] backtracking...
[INF] applying rule: OWNER
[INF] performed: (native) pi.contains(k = sym-key(A, T))
[INF] rule conditions: evaluation failed
```

(Owner) fallisce una prima volta perché non è possibile trovare traccia nella cronologia pi della decifratura di un messaggio appropriato; viene provata (e fallisce) una seconda volta a causa di un nodo di backtrack generato dall'*auto pattern-matching*, che durante il primo tentativo ha consentito di superare il controllo sulla chiave simmetrica k. Si noti che sia k che x, l'identificatore con tag key nel messaggio (de)cifrato cercato, vengono mappati in kAB, difatti la funzione di pattern-matching non è iniettiva: sono le identità non possono avere più di un mapping. A questo punto il processo prova (Encryption), la regola successiva, che però risulta inapplicabile poiché il messaggio da cifrare contiene degli elementi dotati di tag; passa allora a (Claimant):

```
[INF] applying rule: CLAIMANT
[INF] backtrack node here
[INF] performed: (native) pi.contains(kAB = sym-key(A, B))
```

```
[INF] performed: (external) pi.isLast(B = {run()})
[INF] rule conditions: evaluation successful
[INF] performed: (native) pi.add
      (y = {enc({A:id, x:claim}:kAB)})
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

grazie anche all'auto pattern-matching (Claimant) riesce nell'impresa di validare l'encrypt, come testimonia pi:

```
gamma: [m = {"unchecked"}]
pi:    [kAB = sym-key(A, B), B = {run()},
        y = {enc({A:id,x:claim}:kAB)]
```

arrivato a questo punto il nucleo conclude agevolmente il processo di validazione del primo sequential process e può passare al successivo, non prima di aver azzerato i contenitori e lo stack degli stati di backtrack: è infatti sufficiente trovare un "cammino" di derivazione per garantire la correttezza locale del sequential process. La firma del *responder* subisce un'elaborazione analoga alla precedente:

```
 ...
[INF] [InitiatorISO] successfully applied rules:
      [NEW_NAME, INPUT,RUN, CLAIMANT, OUTPUT, NIL]
[INF] [ResponderISO] validating sequential process:
      ResponderISO(A:Id, kAB = sym-key(A, B))
 ...
[INF] validating declaration: let(kAB = sym-key(A, B))
 ...
[INF] declaration validation successful
```

sono poi applicate senza problemi (New Name), (Output), (Input) e (Decryption), portando i contenitori nello stato seguente:

```
gamma: [nB = {"unchecked"}]
pi:    [kAB = sym-key(A, B),
        y = {dec({A:id, nB:claim, z}(:kAB))}]
```

dovendo validare la commit, il nucleo si ritrova a scegliere fra tre regole definite: (Authenticate Claim), (Authenticate Verif), (Authenticate Owner); fortunatamente la prima è quella corretta, quindi il processo si può concludere con successo senza troppo backtrack:

```
[INF] -> validating instruction: commit(B, A)
[INF] applying rule: AUTHENTICATE_CLAIM
[INF] backtrack node here
[INF] performed: (native) gamma.contains(nB = {"unchecked"})
```

```
[INF] backtrack node here
[INF] performed: (native) pi.contains
      (y = {dec({A:id, nB:claim, z}(:kAB))})
[INF] performed: (native) pi.contains(kAB = sym-key(B, T))
[INF] performed: (native) pi.contains(kAB = sym-key(B, A))
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.set(nB = {"checked"})
[INF] rule actions: performing result = true
[INF] instruction validation successful
[INF] -> validating instruction: 0
[INF] applying rule: NIL
[INF] instruction validation successful
[INF] [ResponderISO] sequential process validation successful
[INF] [ResponderISO] successfully applied rules:
      [NEW_NAME, OUTPUT, INPUT, DECRYPTION,
       AUTHENTICATE_CLAIM, NIL]
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      protocol local validation successful
```

6.2 Validazione: Amended SPLICE/AS

L'*Amended SPLICE/AS* è un protocollo di autenticazione mutua che sfrutta lo schema a chiave pubblica [FM04], sarà perciò validato tramite il sistema di tipi (→ §.C.2.2):

1. $B \rightarrow A:\ B, n_B$
2. $A \rightarrow B:\ A, B, \{|B : \mathsf{Id}, n_B : \mathsf{Verif}, \{|A : \mathsf{Id}, n_A : \mathsf{Verif?}|\}_{Pub(k_B)}|\}_{Priv(k_A)}$
3. $B \rightarrow A:\ \{|B, n_A|\}_{Pub(k_A)}$

in appendice si può trovare la codifica completa in *ρ-spi* (→ §.B.2.2).

Il processo di validazione inizia in modo analogo al precedente analizzando le istanziazioni dei sequential process, senza riscontrare anomalie. Il sistema di tipi definisce delle regole anche per la dichiarazione delle identità:

```
[INF] [Amended_SPLICE_AS_Protocol] starting validation
[INF] [InitiatorS] validating sequential process:
      InitiatorS(J:Id, kI = asym-key(I), kJ = asym-key(J))
 ...
[INF] validating declaration: id(J)
[INF] applying rule: IDENTITY
[INF] performed: (native) gamma.add(J, Un)
[INF] rule actions: performing result = true
```

```
[INF] declaration validation successful
[INF] validating declaration: let(kI = asym-key(I))
[INF] applying rule: ASYMMETRIC_KEY
[INF] performed: (native) gamma.add(kI, key_asym(I))
[INF] rule actions: performing result = true
[INF] declaration validation successful
```

in questo caso sono sfruttate entrambe le pseudo-istruzioni `id` e `let`, applicando rispettivamente (Identitiy) e (Asymmetric Key). Un'applicazione lineare di (New Name), (Output), (Input) e ancora (New Name) porta i contenitori nella seguente situazione:

```
gamma: [I=Un, kI=key_asym(I), kJ=key_asym(J),
        a=Un, J=Un, y=Un, nI=Un]
e:     [fresh(nI), in(J), in(I), in(y), fresh(a)]
```

al momento di validare la decrypt il processo trova nove regole potenzialmente applicabili, ma soltanto una si rviela effetivamente applicabile, la variante Asymmetric Verifier di (POSH Decryption):

```
[INF] -> validating instruction:
      decrypt y as {|I:id, nI:verif,z1:auth|}:Pub(kJ)
[INF] applying rule: AV_POSH_DECRYPTION
[INF] backtrack node here
[INF] performed: (external) gamma.checkType(kJ, key_asym(J))
[INF] performed: (external) gamma.checkType(nI, Un)
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.add(z1, Un)
[INF] performed: (external) gamma.addFreeVar()
[INF] performed: (native) e.add
      (dec({|I:id, nI:verif,z1:auth|}(:Priv(kJ))))
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

la validazione della successiva commit non è altrettanto agevole, difatti sono provate ripetutamente (a causa dei nodi di auto pattern-matching) le versioni Symmetric Claimant e Symmetric Verifier di (POSH Commit) prima di avere successo con la variante Asymmetric Verifier:

```
[INF] -> validating instruction: commit(I, J, z1)
[INF] applying rule: SC_POSH_COMMIT
[INF] backtrack node here
[INF] performed: (native) e.contains(fresh(nI))
[INF] performed: (native) e.contains
      (dec({J:id, nI:claim, z1:auth,...}(:k)))
[INF] rule conditions: evaluation failed
```

```
[INF] [SC_POSH_COMMIT] backtracking...
[INF] applying rule: SC_POSH_COMMIT
[INF] backtrack node here
[INF] performed: (native) e.contains(fresh(a))
[INF] performed: (native) e.contains
      (dec({J:id, a:claim, z1:auth,...}(:k)))
[INF] rule conditions: evaluation failed
[INF] [SC_POSH_COMMIT] backtracking...
[INF] applying rule: SC_POSH_COMMIT
[INF] performed: (native) e.contains(fresh(n))
[INF] rule conditions: evaluation failed
[INF] applying rule: SV_POSH_COMMIT
 ...
[INF] rule conditions: evaluation failed
[INF] applying rule: AV_POSH_COMMIT
[INF] backtrack node here
[INF] performed: (native) e.contains(fresh(nI))
[INF] backtrack node here
[INF] performed: (native) e.contains
      (dec({|I:id, nI:verif,z1:auth|}(:Priv(kJ))))
[INF] performed: (external) gamma.checkType(nI, z1, Un)
[INF] performed: (external) gamma.checkType(kJ, key_asym(J))
[INF] rule conditions: evaluation successful
[INF] performed: (native) e.delete(fresh(nI))
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

in particolare si può notare come il nucleo tenti di assegnare tutti i valori compatibili col parametro dell'effetto `fresh` per poi rinunciare e lasciare il valore originale. La successiva decrypt subisce la stessa sorte della precedente, essendo immediatamente individuata la regola corretta grazie al pattern-matching con l'istruzione:

```
[INF] -> validating instruction:
      decrypt z1 as {|J:id, x:verif?,M:auth|}:Priv(kI)
[INF] applying rule: AV_SOPH_SOSH_DECRYPTION
[INF] performed: (external) gamma.checkType(kI, key_asym(I))
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.add(x, nonce(I, J, M))
[INF] performed: (native) gamma.add(M, Un)
[INF] performed: (external) gamma.addFreeVar()
[INF] performed: (native) e.add
      (dec({|J:id, x:verif?,M:auth|}(:Pub(kI))))
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

dopo l'applicazione di (SOPH/SOSH Decryption) nella variante Asymmetric Verifier, i contenitori sono nello stato seguente:

```
gamma: [I=Un, kI=key_asym(I), kJ=key_asym(J), M=Un,
        z1=Un, a=Un,J=Un, y=Un, nI=Un, x=nonce(I, J, M)]
e:     [in(J), in(I), in(y), fresh(a),
        dec({|I:id, nI:verif, z1:auth|}(:Priv(kJ))),
        dec({|J:id, x:verif?, M:auth|}(:Pub(kI)))]
```

a questo punto si ha la validazione della `run` che avvia la sessione di conferma del nonce segreto `x`: inizialmente il nucleo tenta di applicare la (Run), ma questa non effettua il cast del tipo del nonce, facendo fallire la successiva `encrypt`, che richiede che tutti gli elementi contenuti nel messaggio abbiano tipo Un:

```
[INF] -> validating instruction: run(I, J, M)
[INF] applying rule: RUN
[INF] performed: (native) e.add(run(I, J, M))
[INF] rule actions: performing result = true
[INF] backtrack node here
[INF] instruction validation successful
[INF] -> validating instruction: encrypt {|I,x|}:Pub(kJ) as z2
[INF] applying rule: PUB_ASYMMETRIC_ENCRYPTION
[INF] performed: (external) gamma.checkType(I, x, Un)
[INF] rule conditions: evaluation failed
[INF] instruction validation failed
[INF] [encrypt] backtracking...
```

il processo di validazione ripiega sulla (SOPH/SOSH Confirm), provando tutti gli elementi in `gamma` prima di trovare il giusto suggerimento di pattern-matching, per un totale di sette passi di backtrack:

```
[INF] -> validating instruction: run(I, J, M)
[INF] applying rule: SOPH_SOSH_CONFIRM
[INF] backtrack node here
[INF] performed: (external) gamma.checkType(kI,nonce(I, J, M))
[INF] rule conditions: evaluation failed
[INF] [SOPH_SOSH_CONFIRM] backtracking...
 ...
[INF] applying rule: SOPH_SOSH_CONFIRM
[INF] backtrack node here
[INF] performed: (external) gamma.checkType(x, nonce(I, J, M))
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.set(x, Un)
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

superato l'ostacolo della run, la validazione procede spedita fino al termine del sequential process applicando prima la variante Public di (Asymmetric Encrypt), quindi (Output) e (Nil).

La validazione del *responder* è più lineare: le encrypt, al pari delle decrypt complementari, hanno un insieme di regole molto numeroso definito su di esse, ma normalmente solo una è applicabile; l'unica istruzione che incorre nel backtrack è la commit finale, che autentica J verificando il nonce segreto `nI`: sono tentate tutte le varietà di (POSH Commit) prima di trovare la regola corretta, la (SOPH/SOSH Commit):

```
gamma: [z=Un, I=Un, kI=key_asym(I), kJ=key_asym(J),
        M=Un, z1=Un, J=Un, y=Un, nI=nonce(J, I, M), x=Un]
e:     [fresh(M), fresh(nI, J, I, M), in(J), in(x),
        run(I, J, M), in(y), dec({|J, nI|}(:Pub(kI)))]


 ...
[INF] applying rule: SOPH_SOSH_COMMIT
[INF] backtrack node here
[INF] performed: (native) e.contains(fresh(nI, J, I, M))
[INF] performed: (native) e.contains(in(nI))
[INF] performed: (native) e.contains(dec({nI, ...}(:k)))
[INF] backtrack node here
[INF] performed: (native) e.contains(dec({|nI, J|}(:Pub(kI))))
[INF] performed: (external) gamma.checkType(nI,nonce(J, I, M))
[INF] rule conditions: evaluation successful
[INF] performed: (native) e.delete(fresh(nI, J, I, M))
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

superato questo punto la validazione può essere completata con successo:

```
[INF] [ResponderS] successfully applied rules:
      [NEW_NAME, NEW_SECRET_NONCE, INPUT, RUN,
       AV_SOPH_SOSH_INQUIRY, RUN, AV_POSH_REQUEST, OUTPUT,
       INPUT, PRIV_ASYMMETRIC_DECRYPTION, SOPH_SOSH_COMMIT,
       NIL]
[INF] [Amended_SPLICE_AS_Protocol]
      protocol local validation successful
```

6.3 Validazione: Amended Needham-Schroeder

L'*Amended Needham-Schroeder* è un protocollo a chiave condivisa che sfrutta un TTP per distribuire una chiave di sessione tra due partecipanti[1] [FM04]:

1. $A \rightarrow B: \ A$
2. $B \rightarrow A: \ \{A, n_B^0\}_{k_{BT}}$
3. $A \rightarrow T: \ A, B, n_A, \{A, n_B^0\}_{k_{BT}}$
4. $T \rightarrow A: \ \{n_A : \mathsf{Owner}, B : \mathsf{Id}, k_{AB} : \mathsf{Key}, z\}_{k_{AT}}$
5. $A \rightarrow B: \ z = \{k_{AB} : \mathsf{Key}, n_B^0 : \mathsf{Owner}, A : \mathsf{Id}\}_{k_{BT}}$
6. $B \rightarrow A: \ \{n_B\}_{k_{AB}}$
7. $A \rightarrow B: \ \{n_B - 1\}_{k_{AB}}$

Il particolare interessante di questo protocollo, oltre al notevole numero di messaggi scambiati, è nell'ultimo scambio: per evitare un attacco di tipo *reflection*, la conferma del nonce segreto n_B dev'essere inviata applicando a n_B una funzione nota che ne cambi il valore, pur mantenendolo verificabile; in questo caso la funzione è $f(x) = x - 1$. Questa situazione è prevista dal sistema di regole di analisi sintattica, con cui si può validare il protocollo a patto di utilizzare la quantificazione universale nella clausola *ipothesis* di una condizione, come sarà illustrato più avanti.

Nonostante il protocollo sia decisamente articolato, la validazione procede abbastanza linearmente, salvo nei casi delle commit e delle encrypt che richiedono qualche passo di backtrack prima di individuare la regola appropriata. Il momento interessante arriva durante la validazione della commit del *responder*, che verifica il nonce segreto decrementato:

```
pi:    [kBT = sym-key(B, T), x = {enc({A, nB}:kBT)},
        xT = {dec({nB:owner, A:id, kAB:key}(:kBT))},
        A = {run()}, y = {enc({a}:kAB)},
        zA = {dec({prec_a}(:kAB))}]
gamma: [nB = {"unchecked"}, a = {"unchecked"}]

 ...
[INF] applying rule: AUTHENTICATE_OWNER
[INF] backtrack node here
[INF] performed: (native) gamma.contains(nB = {"unchecked"})
[INF] backtrack node here
```

[1] La codifica in *ρ-spi* si può trovare in appendice (→ §.B.2.1).

```
[INF] performed: (native) pi.contains(kBT = sym-key(B, T))
[INF] backtrack node here
[INF] performed: (native) pi.contains
      (xT = {dec({A:id, nB:owner,kAB:key}(:kBT))})
[INF] backtrack node here
[INF] performed: (native) pi.contains(zA = {dec({prec_a}(:kAB))})
[INF] backtrack node here
[INF] performed: (native) pi.contains(x = {enc({A, nB}:kBT)})
[INF] performed: (external) zA.contentEquals(pi, x)
[INF] backtrack node here
[INF] performed: (native) pi.contains(y = {enc({a}:kAB)})
[INF] performed: (external) zA.contentEquals(pi, y)
[INF] performed: (native) pi.contains(z = {enc({:::}:yp)})
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.set(nB = {"checked"})
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

come si può vedere, `pi` contiene due cifrature che corripondono al pattern dell'ipotesi `pi.contains(z=enc({:::}:yp))`: grazie alla quantificazione universale vengono provati entrambi, oltre alla condizione spoglia, che corriponde all'ultimo tentativo dell'auto pattern-matching: non tentare alcun matching. Se non fosse attivata di default la quantificazione universale, una versione scorretta del protocollo che non decrementasse il nonce n_B passerebbe con successo la validazione. Si supponga quindi di cambiare il settimo messaggio nel modo seguente:

$$A \rightarrow B: \ \{n_B\}_{k_{AB}}$$

a questo punto, disabilitando la quantificazione universale, si otterrebbe un risultato scorretto:

```
[INF] applying rule: AUTHENTICATE_OWNER
 ...
[INF] performed: (native) pi.contains(zA = {dec({a}(:kAB))})
[INF] backtrack node here
[INF] performed: (native) pi.contains(x = {enc({A, nB}:kBT)})
[INF] performed: (external) zA.contentEquals(pi, x)
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.set(nB = {"checked"})
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

con la quantificazione universale abilitata il processo di validazione fallisce come ci si aspetterebbe:

```
[INF] applying rule: AUTHENTICATE_OWNER
 ...
[INF] performed: (native) pi.contains(zA = {dec({a}(:kAB))})
[INF] backtrack node here
[INF] performed: (native) pi.contains(x = {enc({A, nB}:kBT)})
[INF] performed: (external) zA.contentEquals(pi, x)
[INF] backtrack node here
[INF] performed: (native) pi.contains(y = {enc({a}:kAB)})
[INF] performed: (external) zA.contentEquals(pi, y)
[INF] rule conditions: evaluation failed
[INF] [AUTHENTICATE_OWNER] backtracking...
```

difatti il contenuto del messaggio cifrato `y={a}:kAB` è uguale a quello del messaggio ricevuto indietro `zA={a}:kAB`, il che rende vulnerabile il protocollo ad un attacco per *reflection*, come si accennava all'inizio della sezione.

6.4 Tagging inference: ISO Two-Steps

Il primo esempio di tagging inference illustra il tagging dell'*ISO TWO-Steps*, naturalmente in versione senza tag:

1. $B \rightarrow A : \ n_B$
2. $A \rightarrow B : \ \{A, n_B, m\}_{k_{AB}}$

come si accennava in precedenza, sono state apportate delle leggere modifiche al codice *ρ-spi* del protocollo (→ §.B.2.4): sono state etichettate le istruzioni complementari e si è reso il codice analizzabile tramite il sistema di tipi. È stato infatti necessario aggiungere il tipo del nonce al momento della generazione per mezzo della new, e dichiarare esplicitamente il messaggio da autenticare tramite la coppia run. e commit. Quest'ultima operazione ha richesto anche un cambiamento leggero, ma determinante, delle regole di analisi sintattica: poiché questa non definisce regole per l'autenticazione di messaggi, sono state modificate le regole esistenti in modo che ignorassero il messaggio da autenticare. Questo porta ad un sisema di regole scorretto da un punto di vista teorico, ma la dimostrazione ha comunque validità, perché la correzione del sistema con delle regole appropriate non modificherebbe sostanzalmente il comportamento dell'algoritmo, che è stato progettato per essere indipendente dal sistema di regole, come questo esempio vuole dimostrare.

6.4.1 Analisi sintattica

Il processo di tagging inference, sfruttando l'algoritmo di riordinamento del protocollo, valuta più conveniente partire dal *responder*, iin cui è presente la commit. Il processo di inference ricalca quello di validazione (→ §.6.1) fino alla decrypt: qui vengono sfruttati i *commit hints* per determinare i possibili tagging:

```
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      starting tagging inference
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      starting validation
 ...
[INF] applying rule: DECRYPTION
[INF] performed: (native) pi.add(y = {dec({A, nB, z}(:kAB))})
[INF] rule actions: performing result = true
[INF] instruction validation successful
[INF] -> validating instruction: commit(B, A, z)
[INF] applying rule: AUTHENTICATE_CLAIM
 ...
[INF] instruction validation failed
[INF] [commit] backtracking...
```

il primo tagging tentato è quello nullo, che ovviamente fallisce dato che i contenitori sono nello stato seguente:

```
gamma: [nB = {"unchecked"}]
pi:    [kAB = sym-key(A, B), y = {dec({A, nB, z}(:kAB))}]
```

e tutte le regole di autenticazione richiedono la decifrazione di un messaggio dotato di tag. Il tagging successivo è quello atteso, difatti permette la validazione della commit tramite (AUTHENTICATE CLAIM):

```
[INF] -> validating instruction: decrypt y as {A, nB, z}:kAB
[INF] current tagging: decrypt y as {A:id, nB:claim, z}:kAB
[INF] backtrack node here
[INF] applying rule: DECRYPTION
[INF] performed: (native) pi.add
      (y = {dec({A:id, nB:claim,z}(:kAB))})
[INF] rule actions: performing result = true
[INF] instruction validation successful
[INF] -> validating instruction: commit(B, A, z)
[INF] applying rule: AUTHENTICATE_CLAIM
[INF] backtrack node here
[INF] performed: (native) gamma.contains(nB = {"unchecked"})
[INF] backtrack node here
[INF] performed: (native) pi.contains
      (y = {dec({A:id, nB:claim,z}(:kAB))})
[INF] performed: (native) pi.contains(kAB = sym-key(B, T))
[INF] performed: (native) pi.contains(kAB = sym-key(B, A))
```

```
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.set(nB = {"checked"})
[INF] rule actions: performing result = true
[INF] instruction validation successful
 ...
[INF] [ResponderISO] successfully applied rules:
      [NEW_NONCE, OUTPUT, INPUT, DECRYPTION,
       AUTHENTICATE_CLAIM, NIL]
```

la validazione passa quindi all'*initiator*: poiché l'encrypt, che subisce lo stesso tagging della decrypt complementare, non viene considerata dall'algoritmo di tagging inference trattandosi di un *forward complement*, la validazione procede come in §.6.1 tentando prima (Owner) e quindi (Claimant), che prelude ad una corretta terminazione della validazione:

```
 ...
[INF] [InitiatorISO] successfully applied rules:
      [NEW_NONCE, INPUT, RUN, CLAIMANT, OUTPUT, NIL]
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      protocol local validation successful
```

a questo punto il processo di tagging inference ritorna all'ultimo nodo di backtrack con informazioni sul tagging, cioé la decrypt iniziale:

```
 ...
[INF] [ResponderISO] validating sequential process:
      ResponderISO(A:Id, kAB = sym-key(A, B))
[INF] -> validating instruction:
      decrypt y as {A:id, nB:claim, z}:kAB
[INF] previous tagging removed: decrypt y as {A, nB, z}:kAB
[INF] complement tagging removed: encrypt {A, x, m}:kAB as y
[INF] current tagging: decrypt y as {A:id, nB:verif, z}:kAB
[INF] applying rule: DECRYPTION
[INF] performed: (native) pi.add
      (y = {dec({A:id, nB:verif, z}(:kAB))})
[INF] rule actions: performing result = true
[INF] instruction validation successful
 ...
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      protocol local validation failed
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      tagging inference complete
```

il tagging però risulta scorretto è incapace di soddisfare le regole di autenticazione, quindi la validazione fallisce e, non essendoci più *tagging nodes*, il processo di tagging inference termina ritornando il tagging atteso.

6.4.2 Sistema di tipi

L'algoritmo di riordino del protocollo fornisce gli stessi risultati dell'esemipo precedente, quindi il processo inizia nuovamente dal responder. Analogamente a prima il momento determinante è la validazione della decrypt:

```
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      starting tagging inference
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      starting validation
 ...
[INF] applying rule: SYMMETRIC_DECRYPTION
[INF] backtrack node here
[INF] performed: (external) gamma.checkType(kAB, key_sym(A, B))
[INF] rule conditions: evaluation successful
[INF] performed: (external) gamma.addFreeVar(A, nB, z)
[INF] performed: (native) e.add(dec({A, nB, z}(:kAB)))
[INF] rule actions: performing result = true
[INF] instruction validation successful
[INF] -> validating instruction: commit(B, A, z)
[INF] applying rule: SC_POSH_COMMIT
 ...
[INF] [commit] backtracking...
```

inizialmente viene provata (Symmetric Decrypt), l'unica regola applicabile direttamente, ma le successive regole di autenticazione falliscono perché richiedono tutte la precedente decifratura di materiale dotato di tag. Tra le altre regole disponibili per la decrypt l'unica che permette un pattern-matching corretto "a meno dei tag" è la variante (Symmetric Claimant) di (POSH Decryption):

```
[INF] -> validating instruction: decrypt y as {A, nB, z}:kAB
[INF] current tagging:
      decrypt y as {A:id, nB:claim, z:auth}:kAB
[INF] applying rule: SC_POSH_DECRYPTION
[INF] performed: (external) gamma.checkType(kAB, key_sym(A, B))
[INF] performed: (external) gamma.checkType(nB, Un)
[INF] rule conditions: evaluation successful
[INF] performed: (external) gamma.addFreeVar()
[INF] performed: (native) gamma.add(z, Un)
[INF] performed: (native) e.add
      (dec({A:id, nB:claim, z:auth}(:kAB)))
[INF] rule actions: performing result = true
[INF] backtrack node here
[INF] instruction validation successful
```

questo tagging è l'unico possibile poiché i tre identificatori hanno tre tipi differenti: `A` ha tipo `IdentityTag`, `nB` ha tipo `NonceTag` e `z` `MessageTag`; questa combinazione non permette di trovare tagging permutati corretti rispetto al sistema di tag definito nelle regole correnti.

La validazione a questo punto procede linearmente fino al termine del sequential process e per tutto il successivo:

```
 ...
[INF] [ResponderISO] successfully applied rules:
      [NEW_NAME, OUTPUT, INPUT,
       SC_POSH_DECRYPTION, SC_POSH_COMMIT,NIL]
 ...
[INF] [InitiatorISO] successfully applied rules:
      [NEW_NAME, INPUT, RUN, SC_POSH_REQUEST, OUTPUT, NIL]
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      protocol local validation successful
[INF] [ISO_TwoSteps_Unilateral_Authentication_Protocol]
      tagging inference complete
```

il processo di tagging inference quindi termina restituendo il tagging atteso.

6.4.3 Risultati a confronto

I due esempi evidenziano come il comportamento dei due sistemi di regole sia decisamente simile: pur rispecchiando due approcci radicalmente differenti, esprimono entrambi lo stesso concetto di correttezza locale. I tagging ottenuti sono molto simili ed hanno identica semantica:

```
[Analisi sintattica]
  encrypt {A:id, x:claim, m}:kAB as y.
  decrypt y as {A:id, nB:claim, z}:kAB.

[Sistema di tipi]
  encrypt {A:id, x:claim, m:auth}:kAB as y.
  decrypt y as {A:id, nB:claim, z:auth}:kAB.
```

I tempi sono nello stesso ordine di grandezza, con un leggero vantaggio per il sistema di tipi che prova un solo tagging, quello corretto.

6.5 Tagging inference: Wide Mouthed Frog

Il protocollo *Wide Mouthed Frog* è un protocollo a chiave condivisa per la negoziazione di una chiave condivisa tramite KTC [FM04]. In questo esempio si prende in considerazione la versione basata su nonce[2]:

1. $A \rightarrow T : \; A$
2. $T \rightarrow A : \; n_T$
3. $A \rightarrow T : \; A, \{B, k_{AB}, n_T\}_{k_{AT}}$
4. $T \rightarrow B : \; *$
5. $B \rightarrow T : \; n_B$
6. $T \rightarrow B : \; \{A, k_{AB}, n_B\}_{k_{BT}}$

non ci sarebbe nulla di particolare da portare ad esempio, per quanto riguarda il funzionamento del tool, se non fosse che, durante il testing del processo di tagging inference su questo protocollo, è stato riscontrato un errore nella codifica del sistema di regole di analisi sintattica: venivano individuati due tagging possibili per il protocollo, di cui uno era quello atteso, ma l'altro appariva decisamente insolito:

3. $A \rightarrow T : \; A, \{B : \mathsf{Id}, k_{AB} : \mathsf{Verif}, n_T\}_{k_{AT}}$

la chiave di sessione viene trattata come nonce. In effetti nel protocollo il fatto che k_{AB} sia una chiave di sessione è del tutto implicito; inoltre nella codifica *ρ-spi* la chiave non è nemmeno trattata come messaggio da autenticare, includendola in una coppia run-commit, poiché si tratta appunto di una chiave di sessione. Di conseguenza il processo di tagging inference assegna all'identificatore `kAB` il tipo `NonceTag`, ed ecco che risultano due tagging possibili per il testo cifrato del terzo messaggio:

```
{B:id, kAB:verif, xT}:kAT
{B:id, kAB, xT:verif}:KAT
```

A questo punto ci si è chiesti come mai il primo tagging non veniva scartato: analizzando il comportamento del tool si è potuto riscontrare un errore nella definizione di (Ttp Forward). Nel caso del secondo tagging, quello atteso, la validazione procedeva in questo modo:

```
[INF] -> validating instruction:
      encrypt {A:id, xAB, xB:claim}:kBT as x
 ...
```

[2] La codifica in *ρ-spi* è disponibile in appendice (→ §.B.2.5)

```
[INF] applying rule: TTP_FORWARD_AND_CHECK
[INF] backtrack node here
[INF] performed: (native) gamma.contains(nT = {"unchecked"})
[INF] backtrack node here
[INF] performed: (native) pi.contains(kBT = sym-key(B, T))
[INF] backtrack node here
[INF] performed: (native) pi.contains(kAT = sym-key(A, T))
[INF] backtrack node here
[INF] performed: (native) pi.contains
      (xA = {dec({B:id, nT:verif, xAB}(:kAT))})
[INF] rule conditions: evaluation successful
[INF] performed: (native) gamma.set(nT = {"checked"})
[INF] performed: (native) pi.add
      (x = {enc({A:id, xB:claim}:kBT)})
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

nel secondo caso, non essendo applicabile (TTP FORWARD & CHECK) dato che il TTP non dichiara alcun nonce di nome `xAB`, viene provata (TTP FORWARD) che dovrebbe fallire poiché modella l'inoltro di un nonce ricevuto in precedenza, cosa che non accade:

```
[INF] applying rule: TTP_FORWARD
[INF] backtrack node here
[INF] performed: (native) pi.contains(kBT = sym-key(B, T))
[INF] backtrack node here
[INF] performed: (native) pi.contains(kAT = sym-key(A, T))
[INF] backtrack node here
[INF] performed: (native) pi.contains
      (xA = {dec({B:id, xAB:verif, nT}(:kAT))})
[INF] rule conditions: evaluation successful
[INF] performed: (native) pi.add
      (x = {enc({A:id, xB:claim}:kBT)})
[INF] rule actions: performing result = true
[INF] instruction validation successful
```

Questo comportamento anomalo ha permesso di scoprire l'errore nella definizione di `TTP_FORWARD`:

```
rule TTP_FORWARD(T)
{
  { encrypt(y={A:id, x:claim, ...}:kBT) }
  conditions { thesis( /* altre condizioni */
    pi.contains(any=dec({B:id, n:verif, ...}(:kAT)))); }
  actions { pi.add(y=enc({A:id, x:claim}:kBT)); }
}
```

È evidente che l'identificatore del nonce `n` nella decrypt dovrebbe diventare `x`, come nella encrypt: apportando questa modifica, il processo di tagging inference riporta un solo tagging possibile, quello atteso:

```
A > InitiatorWMF(B:Id, kAT:sym-key(A, T)) :=
 new(kAB).
 out(A).
 in(xT).
 run(A, B).
 encrypt {B:id, kAB, xT:verif}:kAT as x.
 out(A, x).
 0

B > ResponderWMF(A:Id, kBT:sym-key(B, T)) :=
 new(nB).
 out(nB).
 in(x).
 decrypt x as {A:id, xAB, nB:claim}:kBT.
 commit(B, A).
 0

T > ServerWMF(A:Id, kAT:sym-key(A, T),
              B:Id, kBT:sym-key(B, T)) :=
 new(nT).
 in(A).
 out(nT).
 in(A, xA).
 decrypt xA as {B:id, xAB, nT:verif}:kAT.
 in(xB).
 encrypt {A:id, xAB, xB:claim}:kBT as x.
 out(x).
 0
```

6.6 Conclusioni

Il lavoro svolto nell'ambito di questo progetto ha dato risultati complessivamente soddisfacenti: il tool manifesta un comportamento corretto e fornisce risultati in linea con quelli attesi (→ T.6.1), mentre le prestazioni non ne pregiudicano mai un utilizzo agevole. In particolare, il processo di validazione da un responso praticamente immediato, mentre quello di tagging inference comporta comunque tempi d'attesa ragionevoli (→ §.4.3.4), calcolando che la scelta della piattaforma d'implementazione è ricaduta su Java prima che fosse prevista la realizzazione di questa funzionalità. Va comunque considerato che si tratta di un algoritmo potenzialmente *non deterministico*.

Tabella 6.1: Risultati ottenuti dal tool.

	Corretto	Attaccabile	Tagging
Amended Needham-Schoreder	×		×
Amended SPLICE/AS	×		×
F Woo-Lam	×		×
Flawed		×	×
Flawed Woo-Lam		×	×
ISO Two-Pass	×		×
ISO Two-Steps	×		×
NB Wide Mouthed Frog	×		×
Simplified Woo-Lam	×		×

L'ultima colonna indica tutti i protocolli su cui il processo di tagging inference ha dato il risultato atteso.

D'altra parte bisogna sottolineare che il tool è stato sviluppato non perdendo mai di vista gli obbiettivi di estensibilità e manutenibilità: la prima, unita alla parametricità degli algoritmi rispetto ai sistemi di regole, garantisce grande flessibilità ed un possibile riutilizzo in ambiti relativamente diversi. La manutenibilità consentirà a chiunque abbia le cognizioni necessarie di apportare al tool le modifiche desiderate, ampliando il supporto per le codifiche o magari applicando qualche forma di ottimizzazione agli algoritmi.

Per quanto riguarda il lavoro futuro, rimangono sicuramente delle questioni aperte, come la possibilità di aggiungere nel processo di tagging inference anche il *nonce typing* e l'accoppiamento delle primitive di (de)cifratura, che lo renderebbero completamente automatico. Sicuramente potranno essere apportate anche migliorie a livello più pratico, ad esempio un interfaccia grafica meno spartana, ed il supporto per il caricamento dinamico dei moduli di estensione, ma in ogni caso i risultati fin qui raggiunti sembrano indicare che il progetto sia avviato nella giusta direzione.

Si ritiene infatti che il tool possa risultare decisamente utile sia nell'ambito della ricerca dalla quale trae origine, sia come generico strumento per l'analisi di protocolli crittografici: i sistemi di regole già forniti sono in un certo modo complementari e insieme coprono una gamma di scenari notevolmente ampia. Potrà quindi andare ad affiancare Cryptic [GJ04], un tool per l'analisi statica di protocolli crittografici sviluppato da Andy Gordon e Alan Jeffrey, basato sulla teoria di tipi da loro presentata in [GJ01] e [GJ02], del quale ProtOK raccoglie l'eredità.

Appendice A

Manuale di utilizzo

A.1 Avvio

Il tool dispone di due script utilizzati per l'avvio che si trovano nella directory `$PROTOK_HOME/bin`: la versione per Windows è `run.bat`, mentre `run.sh` è quella per UNIX/Linux. Entambi gli script caricano tutti i moduli presenti in `$PROTOK_HOME/lib` ed impostano il classpath della *Java Virtual Machine* in modo che i plugin siano cercati in `$PROTOK_HOME/plugins`. Tramite i due script è inoltre possibile specificare il nome del sistema di regole di default:

- il tool prova a cercare un file col nome ed il percorso specificati tramite l'opzione `defaultRulesSource`;

- se non lo trova, cerca una risorsa con quel nome e percorso nel classpath e nei moduli di estensione caricati all'avvio;

- se fallisce anche in questo caso, viene caricato il sistema di regole di default (contenuto nel modulo `core.jar`).

In entrambi gli script lo standard output del tool è redirezionato nel file di log `$PROTOK_HOME/protok.log`.

A.2 Interfaccia grafica

Il tool dispone di un'interfaccia grafica multifinestra in cui le singole finestre sono *dockable*, cioè possono essere attaccate ai quattro bordi del desktop principale tramite il menu Window, che ne consente anche la disposizione automatica tramite il comando Dispose.

Tramite il menu View è possibile portare in primo piano finestre eventualmente nascoste dalle altre, mentre la voce Validation Rules è in realtà un sottomenu che permette di visualizzare la finestra vera e propria tramite la voce Window, oppure i singoli tab in essa contenuti, corrispondenti al codice del sistema di regole corrente ed alla relativa visualizzazione ad albero.

La finestra Container visualizza lo stato dei contenitori e della mappa di sostituzione durante la validazione passo per passo, Console mostra il contenuto della console di log, mentre Protocol Source contiene un tab per ogni sorgente di protocollo aperta in quel momento.

Al di sotto del desktop principale si trovano tre tab: desktop è il tab corrente che contiene le finestre descritte sopra; console mostra un area di testo che visualizza il log, mette inoltre a disposizione dei comandi per cancellarne il contenuto e regolare la verbosità del logger; errors infine viene portata in primo piano ogni volta che si verifica un errore.

Esternamente all'area del desktop si possono trovare la barra di stato che mostra l'ultimo messaggio prodotto dal tool, mentre in alto ci sono le barre degli strumenti.

A.2.1 Barre degli strumenti

Sono disponibili quattro barre degli strumenti:

- la barra di *gestione dei file* comprende tre pulsanti che regolano le comuni operazioni eseguibili sui file: apertura, salvataggio e chiusura rispettivamente; se si immette un nome di file inesistente nella finestra di dialogo di apertura, sarà creato un nuovo file con quel nome, sempre che lo si salvi;

- la barra delle *regole* comprende due pulsanti: il primo permette di aprire un file contenente la codifica di un sistema di regole, il secondo permette di impostare come corrente il sistema di regole di default; i due pulsanti sono attivi solamente quando non ci sono file aperti, infatti cambiare il sistema di regole con dei protocolli aperti potrebbe provocare delle situazioni pericolose;

- la barra di *gestione del protocollo* permette di visualizzare la finestra dei warnings, passare dalla modalità di visualizzazione del protocollo a quella di modifica, selezionare il primo errore eventualmente reperibile, ed aprire la finestra di dialogo della tagging inference;

- la barra di *validazione* permette di controllare l'andamento del processo di validazione tramite i quattro pulsanti che rispettivamente: fanno tornare indietro di un passo, arrestano, completano, e fanno avanzare di un passo, la validazione; il componente successivo permette di scegliere la granularità dei passi di validazione, cioé quanto spesso si intende fermarsi: selezionare l'ultimo livello equivale a premere il tasto di completamento, in pratica la validazione non si ferma più; l'ultimo componente permette di collegare la verbosità del sistema di log alla granularità corrente dei passi di validazione: questo migliora le prestazioni del tool.

A.2.2 Finestra di editing dei protocolli

Se si modifica un protocollo e poi si tenta di chiuderlo, verrà visualizzato un pannello all'interno del tab corrispondente che domanda se si vuole salvare il protocollo. Un elemento analogo viene visualizzato quando il processo di tagging inference trova uno o più tagging da associare al protocollo: nel pannello è presente un componente che permette di scegliere se visualizzare all'interno del relativo tab il protocollo oppure uno dei suoi tagging.

A.2.3 Finestra di tagging inference

La finestra di tagging inference presenta dei controlli sopra e sotto l'area di visualizzazione della sorgente corrente: il primo controllo in alto serve a selezionare i tagging trovati oppure la sorgente originale; lo slider patience permette di selezionare il numero di tagging uguali trovati i quali si interviene sullo stack dei nodi di backtrack; i due checkbox successivi abilitano o disabilitano (quando possibile) i commit hints e l'algoritmo di riordinamento del protocollo.

La parte inferiore della finestra di dialogo contiene quattro pulsanti che servono rispettivamente ad avviare il processo di tagging inference, fermarlo, salvare il tagging corrente come file separato, o chiudere la finestra.

A.3 Configurazione

Al momento non ci sono grandi margini di configurabilità: le opzioni di impostazione del sistema di regole di default e verbosità del logger sono stati descritti in precedenza (→ §.A.1), mentre le estensioni non richiedono configurazione: se si vuole aggiugere un modulo jar, è sufficiente metterlo nella directory `$PROTOK_HOME/plugins`; se si decide di aggiungere una sola classe o risorsa, è sufficente metterla in:

```
$PROTOK_HOME/plugins/it/unive/dsi/protok/plugins
```

Appendice B

ρ-*spi*

B.1 Grammatica

```
protocolNarration :=
    PROTOCOL_KW identifier COLON
    sequentialProcessDef+
    START_KW DOT processesInst
    EOF

sequentialProcessDef :=
    sequentialProcessHeader SP_ASSIGN sequentialProcess

sequentialProcessHeader
    identifier SEQ_PROC_ASSIGN identifier
    LPAR
    [identifier COLON seqProcHeaderTag
     (COMMA identifier COLON seqProcHeaderTag)*]
    RPAR

seqProcHeaderTag :=
    ID_KW | keyInstr

keyInstr :=
    SYM_KEY_KW LPAR identifier COMMA identifier RPAR |
    ASYM_KEY_KW LPAR identifier RPAR

sequentialProcess :=
    (instruction DOT)* nil
```

instruction :=
    [(IDENTIFIER | INTEGER) COLON]
    (input | output | restriction | decryption | encryption | run | commit)

nil :=
    NIL_KW

input :=
    IN_KW dataList

output :=
    OUT_KW dataList

restriction :=
    NEW_KW LPAR identifier taggedData RPAR

decryption :=
    DECRYPT_KW data AS_KW compositeData

encryption :=
    ENCRYPT_KW compositeData AS_KW identifier

run :=
    RUN_KW principalCouple

principalCouple :=
    LPAR identifier COMMA identifier [COMMA identifier] RPAR

commit :=
    COMMIT_KW principalCouple

dataList :=
    LPAR data (COMMA data)* RPAR

compositeData :=
    LBRACE data (COMMA data)* RBRACE [COLON key = identifier] |
    ASYM_LBRACE data (COMMA data)* ASYM_RBRACE
    [COLON (PUB_KW | PRIV_KW) LPAR identifier RPAR]

data :=
    identifier taggedData | compositeData

Tabella B.1: Valori dei token.

PROTOCOL_KW	Protocol	LET_KW	let
SYM_KEY_KW	sym-key	ASYM_KEY_KW	asym-key
START_KW	start	NIL_KW	0
IN_KW	in	OUT_KW	out
NEW_KW	new	DECRYPT_KW	decrypt
AS_KW	as	ENCRYPT_KW	encrypt
RUN_KW	run	COMMIT_KW	commit
PUB_KW	Pub	PRIV_KW	Priv
SELF_KW	Self	ID_KW	Id
SEQ_PROC_ASSIGN	>	REPL_SEQ_PROC_ASSIGN	!
LPAR	(	RPAR	)
LBRACE	{	RBRACE	}
ASYM_LBRACE	{|	ASYM_RBRACE	|}
DOT	.	COLON	:
COMMA	,	PIPE	|
ASSIGN	=	SP_ASSIGN	:=
INTEGER	[1-9] ([0-9])*		
IDENTIFIER	[A-Z, a-z, _] ([A-Z, a-z, 0-9, _])* (?)?		

taggedData :=
    [`COLON` identifier [dataList]]

identifier :=
    `IDENTIFIER`

processesInst :=
    (keyBinding `DOT`)*
    [`LPAR` sequentialProcessInst (`PIPE` sequentialProcessInst)+ `RPAR`]

keyBinding :=
    `LET_KW` identifier `ASSIGN` keyInstr

sequentialProcessInst :=
    identifier (`SEQ_PROC_ASSIGN` | `REPL_SEQ_PROC_ASSIGN`)
    identifier `LPAR` [identifier (`COMMA` identifier)*] `RPAR`

B.2 Codifiche dei protocolli

B.2.1 Amended Needham-Schroeder Shared Key

```
Protocol Amended_Needham_Schroeder_Shared_Key_Protocol:

A>InitiatorANS(B:Id, kAT:sym-key(A, T)):=
  out(A).
  in(x).
  new(nA).
  out(A, B, nA, x).
  in(xT).
  decrypt xT as {nA:owner, B:id, kAB:key, xB}:kAT.
  out(xB).
  in(yB).
  decrypt yB as {y}:kAB.
  commit(A, B).
  run(A, B).
  new(prec_y).
  encrypt {prec_y}:kAB as z.
  out(z).
  0

B>ResponderANS(A:Id, kBT:sym-key(B, T)):=
  in(A).
  new(nB).
  encrypt {A, nB}:kBT as x.
  out(x).
  in(xT).
  decrypt xT as {nB:owner, A:id, kAB:key}:kBT.
  new(a).
  run(B, A).
  encrypt {a}:kAB as y.
  out(y).
  in(zA).
  decrypt zA as {prec_a}:kAB.
  commit(B, A).
  0

T>ServerANS(A:Id, kAT:sym-key(A, T),
    B:Id, kBT:sym-key(B, T)):=
  in(A, B, xA, x).
  decrypt x as {A, xB}:kBT.
  new(kAB).
```

```
  encrypt {xB:owner, A:id, kAB:key}:kBT as yB.
  encrypt {xA:owner, B:id, kAB:key, yB}:kAT as y.
  out(y).
  0

start.
let kAT = sym-key(A, T).
let kBT = sym-key(B, T).
( A>InitiatorANS(B, kAT)
| B>ResponderANS(A, kBT)
| T>ServerANS(A,kAT, B, kBT))
```

B.2.2 Amended SPLICE/AS

```
Protocol Amended_SPLICE_AS_Protocol:

I>InitiatorS(J:Id, kI:asym-key(I), kJ:asym-key(J)):=
  new(nI:Un).
  out(I,nI).
  in(J,I,y).
  new(a:Un).
  decrypt y as {| I:id, nI:verif, z1:auth |}:Pub(kJ).
  commit(I,J,z1).
  decrypt z1 as {| J:id, x:verif?, M:auth |}:Priv(kI).
  run(I,J,M).
  encrypt {| I, x |}:Pub(kJ) as z2.
  out(z2).
  0

I>ResponderS(J:Id, kI:asym-key(I), kJ:asym-key(J)):=
  new(M:Un).
  new(nI:nonce(J,I,M)).
  in(J,x).
  run(I,J,M).
  encrypt {| I:id, nI:verif?, M:auth |}:Pub(kJ) as z.
  run(I,J,z).
  encrypt {| J:id, x:verif, z:auth |}:Priv(kI) as z1.
  out(I,J,z1).
  in(y).
  decrypt y as {| J, nI |}:Priv(kI).
  commit(I,J,M).
  0
```

```
start.
let kA = asym-key(A).
let kB = asym-key(B).
let kE = asym-key(E).
(A>!InitiatorS(B, kA, kB) | A>!ResponderS(E, kA, kE) |
 B>!InitiatorS(A, kB, kA) | B>!ResponderS(E, kB, kE) )
```

B.2.3 ISO Two-Steps Unilateral Authentication

```
Protocol ISO_TwoSteps_Unilateral_Authentication_Protocol:

A>InitiatorISO(B:Id, kAB:sym-key(A, B)):=
  new(m).
  in(x).
  run(A, B).
  encrypt {A:id, x:claim, m}:kAB as y.
  out(y).
  0

B>ResponderISO(A:Id, kAB:sym-key(A, B)):=
  new(nB).
  out(nB).
  in(y).
  decrypt y as {A:id, nB:claim, z}:kAB.
  commit(B, A).
  0

start.
let kAB = sym-key(A, B).
(A>InitiatorISO(B, kAB) | B>ResponderISO(A, kAB))
```

B.2.4 ISO Two-Steps (versione modificata)

```
Protocol ISO_TwoSteps_Unilateral_Authentication_Protocol:

A>InitiatorISO(B:Id, kAB:sym-key(A, B)):=
  new(m:Un).
  in(x).
  run(A, B, m).
1:encrypt {A, x, m}:kAB as y.
  out(y).
  0
```

```
B>ResponderISO(A:Id, kAB:sym-key(A, B)):=
  new(nB:Un).
  out(nB).
  in(y).
1:decrypt y as {A, nB, z}:kAB.
  commit(B, A, z).
  0

start.
let kAB = sym-key(A, B).
(A>InitiatorISO(B, kAB) | B>ResponderISO(A, kAB))
```

B.2.5 Nonce-Based Wide Mouthed Forg

```
Protocol Nonce_Based_Wide_Mouthed_Frog_Protocol:

A>InitiatorWMF(B:Id, kAT:sym-key(A, T)) :=
  new(kAB).
  out(A).
  in(xT).
  run(A, B).
  encrypt {B:id, kAB, xT:verif}:kAT as x.
  out(A, x).
  0

B>ResponderWMF(A:Id, kBT:sym-key(B, T)) :=
  new(nB).
  out(nB).
  in(x).
  decrypt x as {A:id, xAB, nB:claim}:kBT.
  commit(B, A).
  0

T>ServerWMF(A:Id, kAT:sym-key(A, T),
            B:Id, kBT:sym-key(B, T)) :=
  new(nT).
  in(A).
  out(nT).
  in(A, xA).
  decrypt xA as {B:id, xAB, nT:verif}:kAT.
  in(xB).
  encrypt {A:id, xAB, xB:claim}:kBT as x.
  out(x).
  0
```

```
start.
let kAT = sym-key(A, T).
let kBT = sym-key(B, T).
( A>InitiatorWMF(B, kAT)
| B>ResponderWMF(A, kBT)
| T>ServerWMF(A, kAT, B, kBT))
```

Appendice C

Validation Rules

C.1 Grammatica

validationRules :=
    identifierDecls
    ruleDefs
    `EOF`

identifierDecls :=
    `CONTAINERS_KW` `LBRACE` (containerDecl)* `RBRACE`
    `DECLARATIONS_KW` `LBRACE` (identifierDecl)* `RBRACE`
    [tagDecls]

containerDecl :=
    `IDENTIFIER` `IDENTIFIER` (`COMMA` `IDENTIFIER`)* `SEMICOLON`

identifierDecl :=
    typeName `IDENTIFIER` (`COMMA` `IDENTIFIER`)* `SEMICOLON`

typeName :=
    `PRINCIPAL_KW` | `TTP_KW` | `NAME_KW` | `VARIABLE_KW` | `KEY_KW`

tagDecls :=
    `TAGS_KW` `LBRACE` (tagDecl)* `RBRACE`

tagDecl :=
    (`TAG_KW` | `TAG_TYPE`) idOrKey (`COMMA` idOrKey)* `SEMICOLON`

ruleDefs :=
    `RULES_KW` `LBRACE` (rule)+ `RBRACE`

```
rule :=
    RULE_KW id LPAR id RPAR
    LBRACE
    instructionDecl
    [conditions]
    [actions]
    RBRACE

instructionDecl :=
    LBRACE (instruction | NIL_KW) RBRACE

instruction :=
    id LPAR [arguments] RPAR

arguments :=
    argument (COMMA argument)*

argument :=
    (abstractData | assign | instruction | message | taggedId | asymKeyInstance)

assign :=
    id EQ (value | instruction | message | keyInstr)

keyInstr :=
    KEY_INSTR_KW LPAR IDENTIFIER COMMA IDENTIFIER RPAR |
    ASYM_KEY_INSTR_KW LPAR IDENTIFIER RPAR

value :=
    STRING_VALUE

message :=
    (LBRACE messageData RBRACE |
     ASYM_LBRACE messageData ASYM_RBRACE)
    (key | LPAR key RPAR)

messageData :=
    (taggedId (COMMA taggedId)* [COMMA abstractData] | abstractData)

key :=
    COLON (id | asymKeyInstance)
```

asymKeyInstance :=
    (PUB_KW | PRIV_KW) LPAR id RPAR

taggedId :=
    id [COLON (idOrKey | ANY_TAG_KW)] [LPAR arguments RPAR]

conditions :=
    [operationSource] CONDITIONS_KW LBRACE (condition)+ RBRACE

operationSource :=
    (NATIVE_KW | EXTERNAL_KW)

condition :=
    [RESET_ABSTRACT] [(EBTDIR |UBTDIR)] [BOOL]
    [LPAR operationSource RPAR]
    [ipothesis]
    thesis

ipothesis :=
    IPOTHESIS_KW LPAR orCheck RPAR SEMICOLON

thesis :=
    THESIS_KW LPAR orCheck RPAR SEMICOLON

orCheck :=
    andCheck (PIPE andCheck)*

andCheck :=
    notCheck (AMP notCheck)*

notCheck :=
    [NOT] simpleCheck

simpleCheck:
    LPAR orCheck RPAR | operation

operation :=
    [LPAR operationSource RPAR] id DOT id LPAR arguments RPAR

actions :=
    [operationSource] ACTIONS_KW LBRACE (operation SEMICOLON)+ RBRACE

Tabella C.1: Valori dei token.

RULES_KW	rules
RULE_KW	rule
DECLARATIONS_KW	declarations
CONTAINERS_KW	containers
TAGS_KW	tags
IPOTHESIS_KW	ipothesis
THESIS_KW	thesis
CONDITIONS_KW	conditions
ACTIONS_KW	actions
NIL_KW	0
ABSTRACT_CONTENT_KW	...
ABSTRACT_TAGGED_CONTENT_KW	:::
ANY_TAG_KW	anyTag
NATIVE_KW	native
EXTERNAL_KW	external
KEY_INSTR_KW	sym-key
ASYM_KEY_INSTR_KW	asym-key
PUB_KW	Pub
PRIV_KW	Priv
CONTAINER_KW	Container
PRINCIPAL_KW	Principal
TTP_KW	TrustedThirdParty
NAME_KW	Name
VARIABLE_KW	Variable
KEY_KW	Key
TAG_KW	Tag
BOOL	true | false
STRING_VALUE	`"[^"]*"`
EBTDIR	@exist
UBTDIR	@univ
RESET_ABSTRACT	@ra

idOrKey :=
  (`IDENTIFIER` | `KEY_INSTR_KW`)

id :=
  `IDENTIFIER`

abstractData :=
  (`ABSTRACT_CONTENT_KW` | `ABSTRACT_TAGGED_CONTENT_KW`)

Tabella C.2: Valori dei token (continua).

<table>
<tr><td>NOT</td><td>!</td></tr>
<tr><td>LPAR</td><td>(</td></tr>
<tr><td>RPAR</td><td>)</td></tr>
<tr><td>LBRACE</td><td>{</td></tr>
<tr><td>RBRACE</td><td>}</td></tr>
<tr><td>ASYM_LBRACE</td><td>{|</td></tr>
<tr><td>ASYM_RBRACE</td><td>|}</td></tr>
<tr><td>DOT</td><td>.</td></tr>
<tr><td>COLON</td><td>:</td></tr>
<tr><td>COMMA</td><td>,</td></tr>
<tr><td>PIPE</td><td>|</td></tr>
<tr><td>EQ</td><td>=</td></tr>
<tr><td>SEMICOLON</td><td>;</td></tr>
<tr><td>QUOTE</td><td>"</td></tr>
<tr><td>AMP</td><td>&</td></tr>
<tr><td>IDENTIFIER</td><td>[A-Z, a-z, _] ([A-Z, a-z, 0-9, _])* (?)?</td></tr>
<tr><td>TAG_TYPE</td><td>KeyTag | TypeTag | NonceTag
MessageTag | IdentityTag | CiphertextTag</td></tr>
</table>

C.2 Sistemi di regole

C.2.1 Analisi sintattica

```
containers
{
  SetContainer gamma;
  OrdContainer pi;
}

declarations
{
  Name n, M;
  Principal I, A, B, T;
  Key k, d, yp, ks, kBT, kAT;
  Variable x, y, z;
}

tags
{
  TypeTag Un, nonce;
  KeyTag key;
  IdentityTag id;
  NonceTag verif, claim, owner;
}

// rule definitions
rules
{
  // Protocol correctness
  rule SYM_KEY(I)
  {
  { let(k = sym-key(A, B)) }
  conditions { thesis(!pi.contains(k=sym-key(A, B))); }
  actions { pi.add(k=sym-key(A, B)); }
  }

  rule ASYM_KEY(I)
  {
  { let(k = asym-key(A)) }
  conditions { thesis(!pi.contains(k = asym-key(A))); }
  actions { pi.add(k = asym-key(A)); }
  }
```

```
// Generic Principal Rules
rule NIL(I)
{
{ 0 }
}

rule NEW_NAME(I)
{
{ new(n) }
external conditions { thesis(gamma.fresh(n)); }
actions { gamma.add(n="unchecked"); }
}

rule INPUT(I)
{
{ in(...) }
}

rule OUTPUT(I)
{
{ out(...) }
}

rule OWNER(A)
{
{ encrypt(y={:::}:x) }

conditions
{
  @ra thesis(
  pi.contains(k=sym-key(A, T)) &
  pi.contains(any=dec({B:id, n:owner, x:key, ...}(:k))) &
  (external)pi.isLast(B=run())
  );
}

actions { pi.add(y=enc({:::}:x)); }
}

rule ENCRYPTION(I)
{
{ encrypt(y={...}:d) }

conditions
```

```
{
  ipothesis(pi.contains(any=dec({d:key, :::}(:k))));
  thesis(pi.contains(B=run()) &
     (external)pi.isNext(x=enc({...}:d)));
}

actions { pi.add(y=enc({...}:d)); }
}

rule DECRYPTION(I)
{
{ decrypt(y={:::}(:d)) }
actions { pi.add(y=dec({:::}(:d))); }
}

// TTP and Principal Rules - Claimant and Verifier Rules
rule AUTHENTICATE_CLAIM(A)
{
{ commit(A, B) }

conditions
{
  thesis(
  gamma.contains(n="unchecked") &
  pi.contains(any=dec({B:id, n:claim, ...}(:k))) &
  (pi.contains(k=sym-key(A, T)) |
   pi.contains(k=sym-key(A, B)))
  );
}

actions { gamma.set(n="checked"); }
}

rule AUTHENTICATE_VERIF(A)
{
{ commit(A, B) }

conditions
{
  thesis(
  gamma.contains(n="unchecked") &
  pi.contains(any=dec({A:id, n:verif, ...}(:k))) &
  pi.contains(k=sym-key(A, B))
  );
```

```
}

actions { gamma.set(n="checked"); }
}

rule AUTHENTICATE_OWNER(A)
{
{ commit(A, B) }

conditions
{
  @ra thesis(
  gamma.contains(n="unchecked") &
  pi.contains(k=sym-key(A, T)) &
  pi.contains(any=dec({B:id, n:owner, y:key, ...}(:k)))
  );

  @ra thesis(pi.contains(x=dec({:::}(:y))));

  @ra ipothesis(pi.contains(z=enc({:::}:yp)));
  thesis(!(external)x.contentEquals(pi, z));
}

actions { gamma.set(n="checked"); }
}

rule CLAIMANT(A)
{
{ encrypt(y={A:id, x:claim, ...}:k) }

conditions
{
  thesis(pi.contains(k=sym-key(A, B)) &
     (external)pi.isLast(B=run()));
}

actions { pi.add(y=enc({A:id, x:claim}:k)); }
}

rule VERIFIER(A)
{
{ encrypt(y={B:id, x:verif, ...}:k) }

conditions
```

```
{
  thesis((pi.contains(k=sym-key(A, T)) |
      pi.contains(k=sym-key(A, B))) &
     (external)pi.isLast(B=run()));
}

actions { pi.add(y=enc({B:id, x:verif}:k)); }
}

rule RUN(A)
{
{ run(A,B) }
actions { pi.add(B=run()); }
}

// Local correctness: TTP and Principal Rules - TTP Rules
rule TTP_FORWARD_AND_CHECK(T)
{
{ encrypt(y={A:id, x:claim, ...}:kBT) }

conditions
{
  @ra thesis(
  gamma.contains(n="unchecked") &
  pi.contains(kBT=sym-key(B, T)) &
  pi.contains(kAT=sym-key(A, T)) &
  pi.contains(any=dec({B:id, n:verif, ...}(:kAT)))
  );
}

actions
{
  gamma.set(n="checked");
  pi.add(y=enc({A:id, x:claim}:kBT));
}
}

rule TTP_FORWARD(T)
{
{ encrypt(y={A:id, x:claim, ...}:kBT) }

conditions
{
  @ra thesis(
```

```
    pi.contains(kBT=sym-key(B, T)) &
    pi.contains(kAT=sym-key(A, T)) &
    pi.contains(any=dec({B:id, x:verif, ...}(:kAT)))
    );
  }

  actions { pi.add(y=enc({A:id, x:claim}:kBT)); }
  }

  rule TTP_DISTRIBUTE(T)
  {
  { encrypt(y={A:id, x:owner, ks:key, ...}:k) }

  external conditions { thesis(gamma.match(n="unchecked")); }

  actions
  {
    gamma.set(n="checked");
    pi.add(y=enc({A:id, x:owner, ks:key}:k));
  }
  }
}
```

C.2.2 Sistema di tipi

```
containers
{
  MapContainer gamma;
  MultisetContainer e;
}

declarations
{
  Name n, M, N;
  Principal A, I, J;
  Key k;
  Variable x, z;
}

tags
{
  TypeTag Un, nonce;
  MessageTag auth;
  IdentityTag id;
  NonceTag verif, claim, verif?, claim?;
}

// rule definitions
rules
{
  // Typing process
  rule NIL(A)
  {
    { 0 }
  }

  rule IDENTITY(A)
  {
    { id(I) }

    actions { gamma.add(I, Un); }
  }

  rule SYMMETRIC_KEY(A)
  {
    { let(k = sym-key(I, J)) }
```

```
  actions { gamma.add(k, key_sym(I, J)); }
}

rule ASYMMETRIC_KEY(A)
{
  { let(k = asym-key(I)) }

  actions { gamma.add(k, key_asym(I)); }
}

rule RUN(A)
{
  { run(A,I,M) }

  actions { e.add(run(A,I,M)); }
}

rule INPUT(A)
{
  { in(:::) }

  external actions
  {
    gamma.addFreeVar(:::);
    e.addEach(in(), :::);
  }
}

rule OUTPUT(A)
{
  { out(...) }
  external
    conditions { thesis(gamma.checkType(..., Un)); }
}

rule NEW_NAME(A)
{
  { new(n:Un) }

  actions
  {
    gamma.add(n, Un);
    e.add(fresh(n));
  }
```

```
}

rule NEW_SECRET_NONCE(A)
{
  { new(n:nonce(I,A,M)) }

  actions
  {
    gamma.add(n, nonce(I,A,M));
    e.add(fresh(n,I,A,M));
  }
}

// Authentication rules: Encryption Rules
rule SYMMETRIC_ENCRYPTION(I)
{
  { encrypt(z={...}:k) }

  external conditions
  {
    thesis(gamma.checkType(..., Un) &
           (gamma.checkType(k, Un) |
            gamma.checkType(k, key_sym(I,J))));
  }

  actions { gamma.add(z, Un); }
}

rule PUB_ASYMMETRIC_ENCRYPTION(I)
{
  { encrypt(z={|...|}:Pub(k)) }

  external conditions
  {
    thesis(gamma.checkType(..., Un) &
           gamma.checkType(k, key_asym(J)));
  }

  actions { gamma.add(z, Un); }
}

rule PRIV_ASYMMETRIC_ENCRYPTION(I)
{
  { encrypt(z={|...|}:Priv(k)) }
```

```
  external conditions
  {
    thesis(gamma.checkType(..., Un) &
           gamma.checkType(k, key_asym(I)));
  }

  actions { gamma.add(z, Un); }
}

rule SC_POSH_REQUEST(I)
{
  { encrypt(z={I:id, N:claim, M:auth, ...}:k) }

  external conditions
  {
    thesis((native)e.contains(run(I,J,M)) &
           gamma.checkType(N, M, ..., Un) &
           gamma.checkType(k, key_sym(I,J)));
  }

  actions
  {
    gamma.add(z, Un);
    e.delete(run(I,J,M));
  }
}

rule SV_POSH_REQUEST(I)
{
  { encrypt(z={J:id, N:verif, M:auth, ...}:k) }

  external conditions
  {
    thesis((native)e.contains(run(I,J,M)) &
           gamma.checkType(N, M, ..., Un) &
           gamma.checkType(k, key_sym(I,J)));
  }

  actions
  {
    gamma.add(z, Un);
    e.delete(run(I,J,M));
  }
```

```
}

rule AV_POSH_REQUEST(I)
{
  { encrypt(z={|J:id, N:verif, M:auth, ...|}:Priv(k)) }

  external conditions
  {
    thesis((native)e.contains(run(I,J,M)) &
           gamma.checkType(N, M, ..., Un) &
           gamma.checkType(k, key_asym(I)));
  }

  actions
  {
    gamma.add(z, Un);
    e.delete(run(I,J,M));
  }
}

rule SC_SOPH_SOSH_INQUIRY(J)
{
  { encrypt(z={I:id, N:claim?, M:auth, ...}:k) }

  external conditions
  {
    thesis(gamma.checkType(M, ..., Un) &
           gamma.checkType(N, nonce(I,J,M)) &
           gamma.checkType(k, key_sym(I,J)));
  }

  actions { gamma.add(z, Un); }
}

rule SV_SOPH_SOSH_INQUIRY(J)
{
  { encrypt(z={J:id, N:verif?, M:auth, ...}:k) }

  external conditions
  {
    thesis(gamma.checkType(M, ..., Un) &
           gamma.checkType(N, nonce(I,J,M)) &
           gamma.checkType(k, key_sym(I,J)));
  }
```

```
  actions { gamma.add(z, Un); }
}

rule AV_SOPH_SOSH_INQUIRY(J)
{
  { encrypt(z={|J:id, N:verif?, M:auth, ...|}:Pub(k)) }

  external conditions
  {
    thesis(gamma.checkType(M, ..., Un) &
           gamma.checkType(N, nonce(I,J,M)) &
           gamma.checkType(k, key_asym(I)));
  }

  actions { gamma.add(z, Un); }
}

// Authentication rules: Secret Nonce Typing
rule SOPH_SOSH_CONFIRM(I)
{
  { run(I,J,M) }

  external conditions
    { thesis(gamma.checkType(z, nonce(I,J,M))); }

  actions { gamma.set(z, Un); }
}

// Authentication rules: Authentication
rule SC_POSH_COMMIT(J)
{
  { commit(J,I,M) }

  conditions
  {
    thesis(
      e.contains(fresh(n)) &
      e.contains(dec({I:id, n:claim, M:auth, ...}(:k))) &
      (external)gamma.checkType(n, M, ..., Un) &
      (external)gamma.checkType(k, key_sym(I,J))
    );
  }
```

```
  actions { e.delete(fresh(n)); }
}

rule SV_POSH_COMMIT(J)
{
  { commit(J,I,M) }

  conditions
  {
    thesis(
      e.contains(fresh(n)) &
      e.contains(dec({J:id, n:verif, M:auth, ...}(:k))) &
      (external)gamma.checkType(n, M, ..., Un) &
      (external)gamma.checkType(k, key_sym(I,J))
    );
  }

  actions { e.delete(fresh(n)); }
}

rule AV_POSH_COMMIT(J)
{
  { commit(J,I,M) }

  conditions
  {
    thesis(
      e.contains(fresh(n)) &
      e.contains(
        dec({|J:id, n:verif, M:auth, ...|}(:Priv(k)))) &
      (external)gamma.checkType(n, M, ..., Un) &
      (external)gamma.checkType(k, key_asym(I))
    );
  }

  actions { e.delete(fresh(n)); }
}

rule SOPH_SOSH_COMMIT(J)
{
  { commit(J,I,M) }

  conditions
  {
```

```
    thesis(
      e.contains(fresh(n,I,J,M)) &
      (e.contains(in(n)) |
       e.contains(dec({n, ...}(:k))) |
       e.contains(dec({|n, ...|}(:Priv(k)))) |
       e.contains(dec({|n, ...|}(:Pub(k))))) &
      (external)gamma.checkType(n, nonce(I,J,M))
    );
  }

  actions { e.delete(fresh(n,I,J,M)); }
}

// Authentication rules: Decryption Rules
rule SYMMETRIC_DECRYPTION(I)
{
  { decrypt(z={...}(:k)) }

  external conditions
    { thesis(gamma.checkType(k, key_sym(I,J))); }

  actions
  {
    (external)gamma.addFreeVar(...);
    e.add(dec({...}(:k)));
  }
}

rule PUB_ASYMMETRIC_DECRYPTION(I)
{
  { decrypt(z={|...|}(:Pub(k))) }

  external conditions
    { thesis(gamma.checkType(k, key_asym(J))); }

  actions
  {
    (external)gamma.addFreeVar(...);
    e.add(dec({|...|}(:Priv(k))));
  }
}

rule PRIV_ASYMMETRIC_DECRYPTION(I)
{
```

```
  { decrypt(z={|...|}(:Priv(k))) }

  external conditions
    { thesis(gamma.checkType(k, key_asym(I))); }

  actions
  {
    (external)gamma.addFreeVar(...);
    e.add(dec({|...|}(:Pub(k))));
  }
}

rule SC_POSH_DECRYPTION(J)
{
  { decrypt(z={I:id, n:claim, x:auth, ...}(:k)) }

  external conditions
  {
    thesis(gamma.checkType(k, key_sym(I,J)) &
           gamma.checkType(n, Un));
  }

  actions
  {
    (external)gamma.addFreeVar(...);
    gamma.add(x, Un);
    e.add(dec({I:id, n:claim, x:auth, ...}(:k)));
  }
}

rule SV_POSH_DECRYPTION(J)
{
  { decrypt(z={J:id, n:verif, x:auth, ...}(:k)) }

  external conditions
  {
    thesis(gamma.checkType(k, key_sym(I,J)) &
           gamma.checkType(n, Un));
  }

  actions
  {
    gamma.add(x, Un);
    (external)gamma.addFreeVar(...);
```

```
    e.add(dec({J:id, n:verif, x:auth, ...}(:k)));
  }
}

rule AV_POSH_DECRYPTION(J)
{
  { decrypt(z={|J:id, n:verif, x:auth, ...|}(:Pub(k))) }

  external conditions
  {
    thesis(gamma.checkType(k, key_asym(I)) &
           gamma.checkType(n, Un));
  }

  actions
  {
    gamma.add(x, Un);
    (external)gamma.addFreeVar(...);
    e.add(dec({J:id, n:verif, x:auth, ...}(:Priv(k))));
  }
}

rule SC_SOPH_SOSH_DECRYPTION(I)
{
  { decrypt(z={I:id, x:claim?, z:auth, ...}(:k)) }

  external conditions
    { thesis(gamma.checkType(k, key_sym(I,J))); }

  actions
  {
    gamma.add(x, nonce(I,J,z));
    gamma.add(z, Un);
    (external)gamma.addFreeVar(...);
    e.add(dec({I:id, x:claim?, z:auth, ...}(:k)));
  }
}

rule SV_SOPH_SOSH_DECRYPTION(I)
{
  { decrypt(z={J:id, x:verif?, z:auth, ...}(:k)) }

  external conditions
    { thesis(gamma.checkType(k, key_sym(I,J))); }
```

```
    actions
    {
      gamma.add(x, nonce(I,J,z));
      gamma.add(z, Un);
      (external)gamma.addFreeVar(...);
      e.add(dec({J:id, x:verif?, z:auth, ...}(:k)));
    }
  }

  rule AV_SOPH_SOSH_DECRYPTION(I)
  {
    { decrypt(z={|J:id, x:verif?, z:auth, ...|}(:Priv(k))) }

    external conditions
      { thesis(gamma.checkType(k, key_asym(I))); }

    actions
    {
      gamma.add(x, nonce(I,J,z));
      gamma.add(z, Un);
      (external)gamma.addFreeVar(...);
      e.add(dec({|J:id, x:verif?, z:auth, ...|}(:Pub(k))));
    }
  }
}
```

C.3 Specifica XML

```
<validationRules
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="validationRules.xsd">

  <identifiers>
    <containers>
      <container name="pi" type="orderedContainer"/>
      <container name="gamma" type="unorderedContainer"/>
    </containers>
    <data>
      <names>
        <identities>
          <identity name="I" type="principal"/>
          <identity name="A" type="principal"/>
          <identity name="B" type="principal"/>
          <identity name="T" type="trustedThirdParty"/>
        </identities>
        <name name="n"/>
        <name name="d"/>
        <name name="k"/>
        <name name="k1"/>
        <name name="k2"/>
        <name name="yp"/>
      </names>
      <variables>
        <variable name="x"/>
        <variable name="y"/>
        <variable name="z"/>
      </variables>
      <tags>
        <tag name="Claim"/>
        <tag name="Verif"/>
        <tag name="Owner"/>
        <tag name="Key"/>
        <tag name="Id"/>
      </tags>
    </data>
  </identifiers>
```

```
  <rules>
<!-- Local correctness: Generic Principal Rules -->
    <rule name="NIL" identity="I"/>

    <rule name="NEW" identity="I">
      <token value="new">
        <argument name="n"/>
      </token>
      <conditions>
        <condition>
          <thesis>
            <check op="fresh" container="gamma">
              <argument name="n" value="unchecked"/>
            </check>
          </thesis>
        </condition>
      </conditions>
      <actions>
        <action op="add" container="gamma">
          <argument name="n"/>
        </action>
      </actions>
    </rule>
  </rules>
</validationRules>
```

Appendice D

Interfacce per i plugin

D.1 Container

```
package it.unive.dsi.protok.core.parser;

import java.util.*;

public interface Container
  extends it.unive.dsi.protok.util.CloneableElement
{
  public class OperationException extends Exception
  {
    public OperationException(Throwable cause) { super(cause); }
    public OperationException(String msg) { super(msg); }
  }

  public boolean add(Data data)
    throws OperationException;
  public boolean add(Data key, Data data)
    throws OperationException;
  public boolean contains(Data data);
  public boolean delete(Data data);
  public Data get(Data data);
  public boolean autoPatternMatch(
    Data arg, Data data, Map substMap);

  public Collection content();
  public it.unive.dsi.protok.util.CloneableIterator iterator();
  public Object[] toArray();
  public void clear();
```

```
  public String getContentImage();
  public String getName();
  public void setName(String name);
}
```

D.2 ExternalOperation

```
package it.unive.dsi.protok.core;

import it.unive.dsi.protok.core.parser.Container;

public interface ExternalOperation
{
  public static final String METHOD_NAME = "perform";
  public static final Class[] ARG_CLASSES =
    new Class[]{ Container.class, Object[].class };

  public Object perform(Container container, Object[] args)
    throws Container.OperationException;

  public void setSymbolTable(java.util.Map symbolTable);
}
```

Bibliografia

[AG99] art-1 [2]

[BBD+03] art-1 [5]

[BFM04-1] articolo analisi sintattica

[BFM04-2] articolo type sistem

[Jam04] link a byacc

[DY83] art-1 [11]

[FM04] articolo esempi

[GJ04] link a cryptic

[GJ01] art-2 [11]

[GJ02] art-2 [12]

[jcc04] link a javacc

[jcc04-1] javacc, componenti generate

[jcc04-2] jjtree, componenti generate

[jcc04-3] javacc, errori

[MOV97] A.Menezes, P.van Oorschot, S.Vanstone, *Handbook Of Applied Cryptography*, CRC Press, 1997

[sun04] link a API java