

Verificarea formală a protoaleelor cu Tamarin

ADRIAN MANEA, 510 SLA

19 decembrie 2019

Cuprins

1	Introducere	2
2	Fundamente teoretice	4
2.1	Teoria urmelor	4
2.1.1	Obiecte de lucru și notații	4
2.1.2	Dependență și urme	6
2.2	Sisteme de rescriere	7
2.3	Sisteme etichetate de tranziții	8
2.4	Analiza automată a protocoalelor	9
3	Exemple	10
3.1	Sintaxă și elemente specifice	10
3.2	FirstExample.spthy	12
	Bibliografie	17

1 INTRODUCERE

Tamarin este un produs software care este folosit în modelarea simbolică și analiza protocoalelor de securitate. Modul de operare este dat de

- introducerea unui model de protocol de securitate, ca date de intrare, prin specificarea acțiunilor pe care agenții participanți la protocol le iau;
- o specificație a adversarului;
- o specificație a proprietăților dorite ale protocolului.

Tamarin oferă o modalitate de a raționa asupra protocoalelor de securitate prin specificații, atât ale protocolului, cât și ale adversarului, într-un limbaj expresiv bazat pe reguli de rescriere asupra unor multi-seturi (i.e. într-o logică multi-sortată). Prin aceste reguli, se definește un sistem de tranziții etichetat, ale cărui stări sînt reprezentate de cunoștințele adversarului, de mesajele existente pe rețea, informații despre valorile proaspăt generate (variabilele *fresh*) și generarea de noi mesaje.

Semantica operațională specifică analizei formale a protocoalelor este detaliată în [Cremers și Mauw, 2012, Cap. 3].

Totodată, putem specifica și ecuațional unii operatori criptografici, precum logaritmul (sau exponențierea) discret(ă) Diffie-Hellman.

Se oferă două moduri de a construi demonstrații:

- (1) Un mod *complet automat*, în care se combină reguli de deducție și raționamente ecuaționale, împreună cu argumente euristice pentru a ghida demonstrația. Dacă operația se finalizează, se returnează fie o demonstrație a corectitudinii, fie un contraexemplu, care reprezintă un atac ce încalcă proprietățile dorite.
- (2) Cum problemele de corectitudine sînt, în general, indecidabile, se oferă și un mod *asistat*, interactiv, în care utilizatorul ghidează demonstrația pas cu pas, inspectînd graful stărilor și luînd decizii combinate cu abordarea automată.

Descrierea formală a metodelor de funcționare a Tamarin este dată în [Schmidt, 2012] și [Meier, 2013], din care vom prelua cîteva elemente de bază în capitolul următor. Ideea fundamentală este următoarea: fie E o teorie cu egalitate care definește operatori criptografici,

un sistem de rescriere R asupra unui multi-set, care definește un protocol și ϕ o formulă care definește o proprietate trasabilă. Tamarin poate să verifice validitatea sau satisfiabilitatea lui ϕ pentru urmele lui R modulo E .

2 FUNDAMENTE TEORETICE

2.1 Teoria urmelor

Vom prelua din prezentarea [Mazurkiewicz, 1995], autorul fiind fondatorul teoriei. Carl Petri este citat ca avînd o influență foarte importantă în dezvoltarea teoriei urmelor, prin introducerea (la vîrsta de 13 ani!) a *rețelelor Petri*, utilizate în primă fază pentru reacții chimice, dar dezvoltate apoi pentru modelarea sistemelor computaționale concurente.

Teoria urmelor a venit ca o încercare de a înțelege sistemele concurente folosind limbaje formale. În primă fază, la apariția teoriei în 1970, ea a fost folosită pentru a muta accentul de pe concurența în execuție pe non-determinism. Principalele dificultăți de depășit se legau de:

- *intercalarea proceselor* (eng. *interleaving*);
- *rafinarea* comportamentului unui sistem, prin rafinarea proceselor concurente ce se execută;
- *inevitabilitatea* unui anume comportament — aspect care intra în contradicție cu non-determinismul;
- *serializarea impusă* a tranzacțiilor.

2.1.1 Obiecte de lucru și notații

Prezentăm acum cîteva elemente de bază din teoria urmelor. Pornim cu o mulțime X , o relație binară R definită pe X care va avea anumite proprietăți (e.g. ordine totală). Vom nota mulțimea numerelor naturale $\{0, 1, 2, \dots, n, \dots\}$ cu ω . Printr-un *alfabet* vom înțelege o colecție finită de simboluri, iar alfabetele se vor nota cu majuscule grecești.

Dacă Σ este un alfabet, șirurile de simboluri cu număr finit de elemente se vor nota cu Σ^* , iar șirul de lungime zero se va nota ϵ .

Mulțimea șirurilor peste un alfabet, împreună cu operația de concatenare formează un monoid, care se va numi *monoidul liber* generat de (alfabetul) Σ .

Dacă w este un șir, vom nota cu $\text{Alph}(w)$ mulțimea simbolurilor din w (*alfabetul* lui w), iar prin $w(a)$ vom înțelege numărul de apariții ale simbolului a în șirul w .

Fie Σ un alfabet și w un șir, posibil definit peste un alt alfabet. Se definește *proiecția* lui w peste Σ funcția:

$$\pi_{\Sigma}(w) = \begin{cases} \epsilon, & w = \epsilon \\ \pi_{\Sigma}(u), & w = ua, a \notin \Sigma \\ \pi_{\Sigma}(u)a, & w = ua, a \in \Sigma \end{cases}$$

În esență, proiecția pe Σ șterge dintr-un șir toate simbolurile care nu se găsesc în Σ .

Reducerea la dreapta a simbolului a din șirul w este un șir notat cu $w \div a$ și definit prin:

$$\begin{aligned} \epsilon \div a &= \epsilon \\ (wb) \div a &= \begin{cases} w, & a = b \\ (w \div a)b, & \text{altfel} \end{cases} \end{aligned}$$

definiție care are sens pentru orice șir w și simboluri a, b .

Se poate verifica ușor că proiecția și reducerea comută, adică avem:

$$\pi_{\Sigma}(w) \div a = \pi_{\Sigma}(w \div a),$$

pentru orice șir w și simbol a .

Vom defini un *limbaj* printr-un alfabet Σ și o mulțime de șiruri definite peste Σ , adică o submulțime a Σ^* . Formal, vom deosebi două limbaje care au aceeași mulțime de șiruri, dar sînt definite peste alfabet diferite. În particular, două limbaje care conțin doar simbolurile vide, dar definite peste alfabet diferite, vor fi considerate diferite.

Limbajele se vor nota cu majuscule din alfabetul latin.

Dacă A, B sînt două limbaje definite peste același alfabet, se poate defini *concatenarea* lor AB , care conține $\{uv \mid u \in A, v \in B\}$, peste același alfabet.

Puterea unui limbaj A se definește inductiv:

$$A^0 = \{\epsilon\}, \quad A^{n+1} = A^n A, \forall n \in \mathbb{N},$$

iar iterația A^* se definește prin:

$$A^* = \bigcup_{n \geq 0} A^n.$$

În general, putem extinde funcția de proiecție de la șiruri la limbaje. Fie A un limbaj, Σ un alfabet. Se definește proiecția lui A pe Σ ca fiind limbajul $\pi_{\Sigma}(A)$ dat de:

$$\pi_{\Sigma}(A) = \{\pi_{\Sigma}(u) \mid u \in A\}.$$

Dacă w este un șir, mulțimea:

$$\text{Pref}(w) = \{u \mid \exists v, uv = w\}$$

se va numi *mulțimea prefixelor* lui w . Evident, $\epsilon, w \in \text{Pref}(w)$, iar dacă A este un limbaj peste Σ , definim:

$$\text{Pref}(A) = \bigcup_{w \in A} \text{Pref}(w).$$

În general, are loc $A \subseteq \text{Pref}(A)$, iar dacă și incluziunea reciprocă este adevărată, limbajul se numește *închis la prefixe*. Rezultă că, pentru orice limbaj A , $\text{Pref}(A)$ este un limbaj închis la prefixe.

Se definește *relația de prefix* ca fiind o relație binară $\sqsubseteq \in \Sigma^* \times \Sigma^*$ definită prin:

$$u \sqsubseteq w \iff u \in \text{Pref}(w),$$

Se poate arăta simplu că relația de prefix este o relație de ordine pentru orice mulțime de șiruri.

2.1.2 Dependență și urme

Vom numi o *dependență* orice relație finită, reflexivă și simetrică, adică o mulțime finită de perechi ordonate D , astfel încât, dacă $(a, b) \in D$, atunci $(b, a) \in D$ și $(a, a) \in D$.

Fie acum D o dependență. Domeniul lui D se va nota Σ_D și se va numi *alfabetul* lui D . Dacă D este o dependență, atunci relația $I_D = (\Sigma_D \times \Sigma_D) - D$ se numește *independența* indusă de D . Evident, această relație este simetrică și nereflexivă. În particular, relația vidă, relația de identitate și relația completă pe Σ (aceasta din urmă fiind $\Sigma \times \Sigma$) sînt dependențe. Prima are alfabetul vid, a doua este cea mai mică dependență în Σ , iar ultima este cea mai mare dependență în Σ .

De exemplu, fie relația:

$$D = \{a, b\}^2 \cup \{a, c\}^2.$$

Aceasta este o dependență și avem:

$$\Sigma_D = \{a, b, c\}, \quad I_D = \{(b, c), (c, b)\}.$$

Teoria urmelor va avea dependențele ca noțiuni primare. De asemenea, se mai pot folosi și *alfabete concurente* ca noțiuni primare, alcătuite din orice pereche (Σ, D) , unde Σ este un alfabet, iar D este o dependență sau *alfabete suport* (eng. *reliance alphabet*), alcătuite dintr-un triplet format dintr-un alfabet Σ , o dependență D și independența I indusă de D .

Ajungem în fine la definiția principală.

Definiție 2.1: Fie D o dependență. Se definește *echivalența de urmă* (eng. *trace equivalence*) pentru D ca fiind cea mai mică congruență \equiv_D în monoidul Σ_D^* astfel încât, pentru orice $a, b \in \Sigma$ să avem:

$$(a, b) \in I_D \implies ab \equiv_D ba.$$

Clasele de echivalență din Σ_D^* / \equiv_D se numesc *urme* (eng. *traces*) peste D .

Urma reprezentată de șirul w se va nota $[w]_D$, iar prin $[\Sigma^*]_D$ vom nota mulțimea factor, respectiv $[\Sigma]_D$ va nota mulțimea claselor care include și clasa vidă ($\{[a]_D \mid a \in \Sigma_D\}$).

Reluând exemplul de mai sus, pentru dependența:

$$D = \{a, b\}^2 \cup \{a, c\}^2,$$

urma peste D dată de șirul $abbca$ este:

$$[abbca]_D = \{abbca, abcba, acbba\}.$$

În general, din definiție, toate șirurile care diferă numai prin ordinea a două simboluri consecutive determină o singură urmă.

Se definește *monoidul urmelor* $M(D)$ ca avînd mulțimea suport dată de mulțimea factor Σ_D^*/\equiv_D , iar operația definită în mod natural. Amintim că există aplicația de proiecție canonică de la mulțimea Σ^* la mulțimea factor, care de fapt induce un morfism natural de monoizi:

$$\varphi_D : \Sigma^* \rightarrow M(D), \quad \varphi_D(w) = [w]_D.$$

Presupunem în continuare că lucrăm cu o dependență fixată D , astfel că vom omite indicele în cele ce urmează. De asemenea, I va fi independența indusă de D (fixată), Σ va fi domeniul lui D , toate simbolurilor vor fi din Σ_D , toate șirurile vor fi peste Σ_D , iar toate urmele vor fi peste D , dacă nu se specifică altfel.

Implicația $u \equiv v \Rightarrow \text{Alph}(u) = \text{Alph}(v)$ este clară. Rezultă că definiția $\text{Alph}([w]) = \text{Alph}(w)$ este corectă.

Fie $\sim : \Sigma^* \times \Sigma^*$ o relație binară definită astfel încît $u \sim v$ dacă și numai dacă există $x, y \in \Sigma^*$ și $(a, b) \in I$ cu $u = xaby$ și $v = xbay$. Cu această definiție, se poate vedea că \equiv este închiderea simetrică, reflexivă și tranzitivă a \sim . Cu alte cuvinte, obținem:

$$u \equiv v \iff \exists (w_0, \dots, w_n), w_0 = u, w_n = v,$$

iar pentru toți $0 < i \leq n$ are loc $w_{i-1} \sim w_i$.

Fie $[u], [v]$ două urme peste aceeași dependență. Urma $[u]$ se numește *prefix* al urmei $[v]$ (iar $[v]$ se numește un *dominant* al lui $[u]$) dacă există o urmă $[x]$ astfel încît $[ux] = [v]$.

E important de știut că:

Propoziție 2.1: Fie $[w]$ o urmă, iar $[u], [v]$ prefixe ale lui $[w]$. Atunci există cel mai mare prefix comun și cel mai mic dominant comun al lui $[u]$, respectiv $[v]$.

Demonstrație. [Mazurkiewicz, 1995, Prop. 1.3.5]. □

2.2 Sisteme de rescriere

Preluăm o prezentare sumară a subiectului din [Genet și Klay, 2000], indicînd pentru mult mai multe detalii Capitolul 3 din [van Leeuwen (ed.), 1990].

Fie \mathcal{F} o mulțime finită de simboluri, care conține și o funcție de aritate definită pe \mathcal{F} . Fie \mathcal{X} o mulțime numărabilă de variabile, $\mathcal{T}(\mathcal{F}, \mathcal{X})$ o mulțime de termeni, iar $\mathcal{T}(\mathcal{F})$ mulțimea termenilor fără variabile. Mulțimea pozițiilor dintr-un termen t , notată $\text{Pos}(t)$, este ordonată lexicografic. Șirul vid ϵ ține poziția cea mai de sus. Dacă $p \in \text{Pos}(t)$, atunci $t|_p$ este subtermenul lui t de la poziția p , iar $t[s]_p$ este termenul obținut prin înlocuirea subtermenului $t|_p$ la poziția p cu termenul s . Mulțimea variabilelor din termenul t se va nota cu $\text{Var}(t)$.

O *substituție* este o aplicație

$$\sigma : \mathcal{X} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X}),$$

care poate fi extinsă în mod unic la un endomorfism al mulțimii termenilor (cu concatenarea). Domeniul său se definește prin:

$$\text{Dom}(\sigma) = \{x \in \mathcal{X} \mid x\sigma \neq x\}.$$

Un *sistem de rescriere a termenilor* \mathcal{R} este o mulțime de reguli de rescriere de forma $l \rightarrow r$, unde $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, iar $\text{Var}(l) \supseteq \text{Var}(r)$.

Relația $\rightarrow_{\mathcal{R}}$ indusă de un sistem de rescriere \mathcal{R} pe o mulțime de termeni $\mathcal{T}(\mathcal{F}, \mathcal{X})$ se definește astfel:

$$s \rightarrow_{\mathcal{R}} t \iff (\exists l \rightarrow r \in \mathcal{R}, \exists p \in \text{Pos}(s), \exists \sigma \text{ subst. a.î. } (l\sigma = s|_p \wedge t = s[r\sigma]_p)).$$

Închiderea tranzitivă a relației se va nota $\rightarrow_{\mathcal{R}}^*$.

2.3 Sisteme etichetate de tranziții

Un sistem etichetat de tranziții (eng. *labelled transition system*, *LTS*) este un cvadruplu notat (S, L, \rightarrow, s_0) , care conține:

- o mulțime de stări S ;
- o mulțime de etichete L ;
- $\rightarrow : S \times L \times S$ este o operație de tranziție de aritate 3;
- $s_0 \in S$ este o stare inițială, fixată.

Tranziția (p, α, q) se va nota pe scurt $p \xrightarrow{\alpha} q$.

Păstrînd notațiile de mai sus, se numește *execuție finită* a unui sistem de tranziții etichetate P un șir alternant σ de stări și etichete, care începe cu s_0 și se termină într-o stare s_n , astfel încît:

$$\sigma = [s_0, \alpha_1, s_1, \alpha_2, \dots, \alpha_n, s_n] \implies s_i \xrightarrow{\alpha_{i+1}} s_{i+1}, \forall 0 \leq i < n.$$

Pentru o execuție finită σ ca mai sus, șirul fără primul termen s_0 (starea inițială) se numește *o urmă finită* a lui P .

Un sistem de tranziții etichetate poate fi definit printr-o serie de reguli de tranziție. Aceasta înseamnă un număr de premise Q_1, \dots, Q_n , cu $n \geq 0$, care au loc pentru ca o concluzie de forma $p \xrightarrow{\alpha} q$ să poată fi trasă, notația fiind ca în cazul unei derivări logice:

$$\frac{Q_1 \quad Q_2 \quad \dots \quad Q_n}{p \xrightarrow{\alpha} q}$$

Un exemplu simplu este al unui sistem etichetat de tranziții pentru a controla un contor. Fie $S = \mathbb{B} \times \mathbb{N}$, unde $\mathbb{B} = \{\top, \perp\}$ este mulțimea booleană. Dacă într-o stare, prima poziție este \perp , spunem că a avut loc o eroare. Starea inițială este $s_0 = (\top, 0)$. Mulțimea etichetelor se definește:

$$L = \{\text{inc}, \text{dec}, \text{err}, \text{reset}\},$$

iar relația de tranziție este definită prin regulile:

$$\begin{array}{c} \frac{b = \top}{(b, n) \xrightarrow{\text{inc}} (b, n + 1)} \quad \frac{b = \top \quad n > 0}{(b, n) \xrightarrow{\text{dec}} (b, n - 1)} \\[10pt] \frac{}{(b, n) \xrightarrow{\text{err}} (\perp, n)} \quad \frac{}{(b, n) \xrightarrow{\text{reset}} (\top, 0)} \end{array}$$

Semnificația relației este clară. Un exemplu de execuție în acest sistem ar fi:

$$\begin{array}{c} (\top, 0) \xrightarrow{\text{inc}} (\top, 1) \xrightarrow{\text{inc}} (\top, 2) \xrightarrow{\text{dec}} (\top, 1) \\ \xrightarrow{\text{err}} (\perp, 1) \xrightarrow{\text{reset}} (\top, 0) \xrightarrow{\text{inc}} (\top, 1), \end{array}$$

care conduce la urma:

$$[\text{inc}, \text{inc}, \text{dec}, \text{err}, \text{reset}, \text{inc}].$$

2.4 Analiza automată a protocoalelor

Preluăm acum din [Schmidt, 2012, Cap. 3] pentru un exemplu general de analiză formală a protocoalelor de criptare care folosesc exponențierea Diffie-Hellman.

3 EXAMPLE

3.1 Sintaxă și elemente specifice

Preluăm din articolul [Basin et al., 2017] câteva elemente introductive privitoare la funcționarea Tamarin.

Studiul de caz prezentat este acela al schimbului de chei Diffie-Hellman (DH).

Observație 3.1: Sintaxa folosită în cele de mai jos este imprecisă, deoarece am preferat lizibilitatea și claritatea. Sintaxa exactă este dată în fișierele-sursă conținute în directorul proiectului, în interiorul directorului `src`.

Sursa de cod va fi un fișier cu extensia `spthy` (abreviere de la *Security Protocol TheorY*), care reprezintă o teorie pe care compilatorul o va testa. Fișierul de teorie conține următoarele elemente:

Date de intrare: Teoria ecuațională: Specificăm mesajele pentru protocol sub forma:

```
builtins: diffie-hellman
functions: mac/2, g/0, shk/0 [private]
```

Astfel, elementele specifice protocolului DH (care este inclus în biblioteca standard, lucru menționat prin sintaxa `builtins`) sînt încărcate. În particular, avem definit operatorul \wedge , folosit pentru exponențiere.

Se mai definește o funcție de 2 variabile `mac`, care modelează un cod de autentificare a mesajelor (eng. *Message Authentication Code*) și două constante, `g` pentru a modela generatorul grupului în care are loc problema, respectiv `shk` pentru o cheie secretă partajată (eng. *shared secret key*). Acestea sînt declarate `private`, deci nu pot fi obținute direct de către adversar.

De asemenea, cum Tamarin este implementat în Haskell, avem suport inclus pentru funcțiile specifice perechilor, `<_>`, `fst`, `snd`.

Date de intrare: Protocolul: Pentru modelarea protocolului vom utiliza 3 reguli de scriere într-un sistem bazat pe un multiset. Fiecare regulă este un triplet, alcătuit din șiruri de fapte în membrul stîng, etichete și membrul drept. Faptele sînt de forma $F(t_1, \dots, t_k)$, pentru un simbol specific F și termenii t_i . Regulele protocolului folosesc simboluri unare de fapte `Fr` și `In` pentru a obține constante noi (eng. *fresh names*) și mesajele de pe rețea.

Un mesaj este trimis pe rețea folosind un simbol unar de fapte `Out` în membrul drept.

Prima regulă creează un fir de execuție pentru protocol `tid` care alege un exponent nou `x` și trimite participanților g^x , concatenat cu `mac` aplicat acestei valori:

```
rule Step1: [ Fr(tid: fresh), Fr(x: fresh) ] -[ ]->
  [ Out(<g^(x: fresh), mac(shk, <g^(x: fresh), A: pub, B: pub>>>)
    , Step1(tid: fresh, A: pub, B: pub, x: fresh) ]
```

În aceste linii de sintaxă se specifică explicit ca variabilele utilizate să fie instanțiate doar cu sorturile `fresh`, respectiv `pub`, acolo unde este cazul.

Spunem că o instanță a regulii `Step1` *consumă* două fapte `Fr` pentru a obține constantele noi `tid` și `x` și *generează* un fapt `Out` care conține mesajul trimis și un fapt `Step1` care arată că firul de execuție a finalizat primul pas cu parametrii dați.

Remarcăm că regula nu are etichete, deci ea nu va afecta urma sistemului de rescriere.

A doua regulă este pasul al doilea din protocol:

```
rule Step2: [ Step1(tid, A, B, x: fresh), In(<Y, mac(Y, B, A)>>) ]
  -[ Accept(tid, Y^(x: fresh)) ]-> []
```

Aici se folosește faptul din pasul 1 și se verifică MAC-ul. Eticheta din final, `Accept(tid, Y^(x: fresh))`, arată că firul de execuție `tid` a acceptat cheia de sesiune Y^x .

Regula a treia arată cheia secretă adversarului:

```
rule RevealKey: [] -[ Reveal() ]-> [ Out(shk) ]
```

Eticheta `Reveal()` asigură că se va vedea pe urma sistemului dacă și când a avut loc dezvăluirea.

Date de intrare: Proprietăți: În continuare, trebuie specificate proprietățile pe care să le satisfacă sistemul, sub forma unor leme. Acestea vor fi verificate și admise sau respinse de către Tamarin. Un exemplu este:

```
lemma Accept_Secret:
  ∀ i j tid key. Accept(tid, key)@i & K(key)@j =>
    ∃ n. Reveal()@n & n < i
```

Această leamnă cuantifică peste momentele de timp i , j , n și mesajele `tid`, `key`. Predicatul folosit este de forma $F @ i$, care arată că faptul F are loc la momentul i . În total, lema arată că dacă un fir de execuție `tid` a acceptat o cheie `key` la momentul i , iar cheia este cunoscută și adversarului, atunci trebuie să existe un moment de timp n anterior lui i când a avut loc dezvăluirea (`Reveal()`).

Date de ieșire În fine, dacă rulăm Tamarin pe informațiile de mai sus, salvate în fișierul sursă `example.spthy`, invocat în linia de comandă cu:

```
$ tamarin-prover example.spthy
```

rezultă următorul mesaj:

```
analyzed example.spthy: Accept_Secret (all-traces) verified (9 steps)
```

Acest lucru arată că Tamarin a analizat cu succes că toate urmele protocolului satisfac proprietatea din lema `Accept_Secret`.

3.2 FirstExample.spthy

Preluăm acum prezentarea unui exemplu ceva mai sofisticat, din manualul oficial Tamarin ([Tamarin, 2019]). O notație generică pe care o vom folosi pentru schimbul de mesaje este de felul următor:

```
C -> S: aenc(k, pkS)
C <- S: h(k)
```

Această notație arată că în protocolul considerat, clientul `C` generează o cheie simetrică nouă (*fresh*) `k`, o criptează cu o cheie publică `pkS` a unui server `S` (notația `aenc` înseamnă *criptare asimetrică*) și o trimite serverului `S`. Serverul confirmă primirea cheii și trimite un hash al acesteia înapoi la client.

Modelul adversarului este de tip Dolev-Yao, adică este omniprezent în rețea și poate șterge, modifica și intercepta mesaje, fiind limitat doar de criptografia folosită.

În continuare, detaliem sursa care se distribuie cu manualul Tamarin, în fișierul `FirstExample.spthy`.

După declararea teoriei ce se va defini, se începe codul propriu-zis. Astfel, primele linii de sintaxă sînt, în acest caz:

```
theory FirstExample
begin
```

Apoi se declară primitivele (variabilele, funcțiile și orice alte entități prezente deja în biblioteca standard) ce se vor folosi, iar la final, proprietățile ce vrem să fie respectate, adică lemele.

Obiectele folosite din biblioteca standard se declară astfel:

```
builtins: hashing, asymmetric-encryption
```

Aceste variabile aduc cu sine următoarele funcții utile:

- o funcție unară `h`, care este funcție de hashing;
- o funcție binară `aenc`, pentru criptare asimetrică;

- o funcție binară adec , pentru decriptare asimetrică;
- o funcție unară pk , care reprezintă cheia publică, asociată unei chei private.

Totodată, modulele respective conțin și reguli de simplificare potrivite, adică o teorie ecuațională care conține, de exemplu, ecuația care arată că funcțiile de criptare și decriptare sînt inverse una celeilalte:

$$\text{adec}(\text{aenc}(m, \text{pk}(\text{sk})), \text{sk}) = m.$$

În continuare, avem de specificat regulile de funcționare. Protocolul și mediul în care acesta se desfășoară sînt modelate folosind reguli de rescriere pe multiseturi. Stările sistemului sînt multiseturi (eng. „bags”) de fapte. Aceste fapte sînt predicate care arată informațiile relevante privitoare la starea sistemului. De exemplu, un fapt de forma $\text{Out}(h(k))$ arată că protocolul a trimis mesajul $h(k)$ pe un canal public, iar faptul $\text{In}(x)$ arată că protocolul a primit mesajul x pe canalul public.¹

Prima regulă arată cum se înregistrează o cheie publică:

rule Register_pk:

[Fr(~ltk)] --> [~Ltk(\$A, ~ltk), ~Pk(\$A, pk(~ltk))]

Singura premisă este să existe un fapt de tip Fr, care în sintaxa standard înseamnă că este vorba despre o variabilă nouă (*fresh*). De asemenea, se folosesc următoarele prefixe în sintaxa standard, pentru diversele tipuri de variabile:

- x înseamnă variabilă nouă, scrisă pe larg $x:\text{fresh}$;
- $\$x$ notează o variabilă publică, $x:\text{pub}$;
- $\#i$ notează o variabilă temporală (un moment de timp din evoluția stărilor sistemului), adică $i:\text{temporal}$;
- m simplu notează un mesaj, adică $m:\text{msg}$.

Vom mai folosi, ca o convenție, numele c pentru o constantă publică, globală. Practic, avem sorturile msg , fresh , pub , temporal , cu proprietatea că fresh , $\text{pub} \subseteq \text{msg}$, iar elementele sortului temporal sînt incomparabile cu celelalte.

Regula definită mai sus arată că: se generează un nume nou ltk , deci de sort fresh , care este noua cheie privată. Se alege (în mod nedeterminist) un nume public A pentru agentul căruia i se generează perechea de chei.

Faptul care urmează, $\text{!Ltk}(\$A, \text{ltk})$ arată că se definește o asociere între agentul A și cheia sa privată ltk , iar semnul de exclamare din fața faptului arată că acesta se poate genera de oricîte ori este nevoie (este *persistent*).

Propagarea cheii publice în rețea se face cu regula următoare:

¹Implicit, Tamarin folosește doar canale de comunicație publice, dar se pot defini de către utilizator și canale private. În acest exemplu simplu, însă, vom presupune că folosim doar canale publice.

```
rule Get_pk:
    [ !Pk(A, pubkey) ] --> [ Out(pubkey) ]
```

Compromiterea cheii, adică descoperirea ei de către adversar, este formalizată în regula următoare, unde apare și un *fapt în acțiune* (eng. *actionfact*), care nu este prezent în stările sistemului, ci doar pe urmele sale:

```
rule Reveal_ltk:
    [ !Ltk(A, ltk) ] --[ LtkReveal(A) ]-> [ Out(ltk) ]
```

Funcționarea propriu-zisă a protocolului este acum modelată de următoarele trei reguli:

```
// Începe un nou fir de execuție pentru client,
// prin alegerea unui server în mod non-determinist.
rule Client_1:
    [ Fr(~k), !Pk($S, pkS) ] -->
    [ Client_1 ($S, ~k), Out( aenc(~k, pkS) ) ]
```

Se generează o cheie nouă ($Fr(k)$) și se caută cheia publică pentru sever, iar apoi se stochează serverul și cheia pentru pasul următor și se trimite cheia de sesiune criptată către server.

```
rule Client_2:
    [ Client_1(S, k), In( h(k) ) ] --[ SessKeyC( S, k ) ]-> []
```

În acest pas, se apelează la sesiunea anterioară pentru server și cheie, se primește cheia căreia i s-a aplicat funcția hash și se menționează (pe urmă, deci ca fapt în acțiune) că s-a stabilit o cheie de sesiune k cu serverul S .

```
// Răspunsul unui server într-un fir de execuție la cererea
// din partea unui client.
rule Serv_1:
    [ !Ltk($S, ~ltkS), In( request ) ]
    --[ AnswerRequest($S, adec(request, ~ltkS)) ]->
    [ Out( h(adec(request, ~ltkS)) ) ]
```

Practic, se caută cheia privată după primirea unei cereri, se înregistrează (ca fapt în acțiune) răspunsul la această cerere și se returnează hash-ul cererii decriptate.

Regulile pe care trebuie să le satisfacă protocolul sînt descrise prin următoarele leme.

lemma Client_session_key_secrecy:

```
" not( /* nu este cazul ca */
  Ex S k #i #j.
    /* clientul a stabilit o cheie de sesiune k cu serverul S */
    SessKeyC(S, k) @ #i
    /* iar adversarul să cunoască cheie */
    & K(k) @ #j
    /* fără a fi aplicat o dezvăluire pe S */
    & not(Ex #r. LtkReveal(S) @ r)
  )
"
```

lemma Client_auth:

```
" /* Pentru toate cheile de sesiune 'k'
   stabilite de clienți cu serverul 'S' */
( All S k #i. SessKeyC(S, k) @ #i
  ==>
    /* există un server care a răspuns la cerere */
    ( (Ex #a. AnswerRequest(S, k) @ a)
    /* sau adversarul a aflat cheia înainte de a fi stabilită */
    | (Ex #r. LtkReveal(S) @ r & r < i)
  )
)
"
```

Putem întări autorizarea, cerînd ca aceasta să fie injectivă, adică să nu existe doi clienți care au avut aceeași cerere la server:

lemma Client_auth_injective:

```
" /* Pentru toate cheile de sesiune 'k' stabilite cu serverul 'S' */
( All S k #i. SessKeyC(S, k) @ #i
  ==>
    /* există un server care a răspuns */
    ( (Ex #a. AnswerRequest(S, k) @ a
    /* și niciun alt client n-a avut exact aceeași cerere */
    & (All #j. SessKeyC(S, k) @ #j ==> #i = #j)
    ) /* sau adversarul a aflat cheia înainte de a fi stabilită */
    | (Ex #r. LtkReveal(S) @ r & r < i)
  )
)
"
```


Mai putem adăuga încă o proprietate pentru a ne asigura că proprietățile nu se execută pe un model vid:

```
lemma Client_session_key_honest_setup:
  exists-trace
    " Ex S k #i.
      SessKeyC(S, k) @ #i
      & not(Ex #r. LtkReveal(S) @ r)
    "
```

Întregul exemplu este conținut în fișierul sursă din proiect.

Evaluarea fișierului se face din linia de comandă sau se poate apela la unelte cu interfață grafică, detaliate în [[Tamarin, 2019](#), pp. 20–26].

BIBLIOGRAFIE

- [Basin et al., 2017] Basin, D. A., Cremers, C., Dreier, J., și Sasse, R. (2017). Symbolically analyzing security protocols using tamarin. *SIGLOG News*, 4(4):19–30.
- [Cremers și Mauw, 2012] Cremers, C. și Mauw, S. (2012). *Operational Semantics and Verification of Security Protocols*. Springer.
- [Diekert și Métivier, 1997] Diekert, V. și Métivier, Y. (1997). Partial commutation and traces. In *Handbook of Formal Languages*.
- [Genet și Klay, 2000] Genet, T. și Klay, F. (2000). Rewriting for cryptographic protocol verification. In McAllester, D., editor, *Automated Deduction - CADE-17*, pages 271–290, Berlin, Heidelberg. Springer Berlin Heidelberg.
- [Mazurkiewicz, 1995] Mazurkiewicz, A. (1995). Introduction to trace theory. In Diekert, V. și Rozenberg, G., editors, *The Book of Traces*, chapter 1, pages 3–41. World Scientific.
- [Meier, 2013] Meier, S. (2013). *Advancing automated security protocol verification*. PhD thesis, ETH Zürich.
- [Schmidt, 2012] Schmidt, B. (2012). *Formal analysis of key exchange protocols and physical protocols*. PhD thesis, ETH Zürich.
- [Tamarin, 2019] Tamarin, T. (2019). Tamarin-prover manual. tamarin-prover.github.io.
- [van Leeuwen (ed.), 1990] van Leeuwen (ed.), J. (1990). *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*. Elsevier; MIT Press.