UNIVERSITATEA ALEXANDRU IOAN CUZA IAŞI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENŢĂ

Security Protocols Checker

propusă de

Ștefan Stan

Sesiunea: *Iulie, 2015*

Coordonator ştiinţific

Lect. dr. Cosmin-Nicolae Vârlan

UNIVERSITATEA ALEXANDRU IOAN CUZA IAȘI

FACULTATEA DE INFORMATICĂ

Security Protocols Checker

Ștefan Stan

Sesiunea: *Iulie, 2015*

Coordonator ştiinţific

Lect. dr. Cosmin-Nicolae Vârlan

DECLARAŢIE PRIVIND ORIGINALITATEA ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul "Security Protocols Checker" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imagini etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iași,

Absolvent *Ștefan Stan*

_____

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul "Security Protocols Checker", codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.
De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea Alexandru Ioan Cuza Iaşi să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iași,

Absolvent *Ștefan Stan*

_____

Cuprins

Introducere

Ca societate, devenim din ce în ce mai dependenţi de accesul rapid la informaţie. Pe măsură ce această solicitare creşte, tot mai multă informaţie este stocată şi transmisă electronic. Inevitabil, apare problema protejării datelor transmise în rețea. Această problemă îmbracă diverse forme, de la păstrarea corectitudinii datelor și menținerea în siguranță a informației cu caracter sensibil, până la ascunderea identității celor implicați în comunicare.

Securitatea informației caută în mod continuu soluții la astfel de probleme, ele fiind reprezentate prin prisma următoarelor proprietăți de securitate: *integritate*, *confidențialitate*, *anonimitate*, *autenticitate* și *nerepudiere*. Astfel, se pot introduce protocoalele de securitate, ce sunt secvențe de pași, care specifică cu exactitate acțiunile a două sau mai multe entități pentru a realiza un anumit obiectiv de securitate.

În contextul aplicației prezente, corectitudinea este definită ca fiind proprietatea protocoalelor de securitate de a rezista la atacuri provocate de o entitate, atacator, asigurând *integritatea* și *confidențialitatea* informațiilor transmise în acea instanță de protocol. Acesta poate schimba părți din mesaje cu anumite informații din altă instanță de protocol, sau cu informații false generate de către el. De asemenea, mai poate să se dea drept unul din agenții unei instanțe de protocol, sau poate observa instanțe ce rulează la un moment dat, învățând anumite mesaje cu caracter sensibil și înlocuindu-le cu altele, eventual.

Aplicația *„Security Protocols Checker"* este un utilitar ce verifică atât proprietatea de integritate (un exemplu de caz detectat ar fi acela în care un atacator poate distinge între un agent A ce trimite un mesaj unui agent B_1 în instanța de protocol 1 și același agent A, dar care participă la instanța de protocol 2 și trimite un alt mesaj unui agent B_2, mesaje ce pot fi ușor interschimbate/modificate de atacator pentru a obține avantaje ulterioare), precum și cea de confidențialitate (aplicația detectează posibile modalități prin care atacatorul poate schimba mesajele astfel încât, la un moment dat, acesta reușește să obțină o informație cu caracter sensibil a unui agent).

Un utilitar similar ce este folosit pentru verificarea proprietății de confidențialitate este Scyther, construit de către Cas Cremers în cadrul tezei sale de doctorat, „Scyther - Semantics and Verification of Security Protocols" [3]. Acestă aplicație poate verifica confidențialitatea unei instanțe de protocol, oferind atacurile posibile într-o manieră grafică, mai apropiată de utilizator, în cazul unei instanțe de protocol vulnerabile.

În cele ce urmează, vor fi trecute în revistă câteva detalii legate de anonimitate în *Capitolul 1*, precum și o descriere a modelului teoretic ce stă la baza aplicației, model împrumutat din [1] și [2], în *Capitolul 2*. Lucrarea continuă cu o prezentare a arhitecturii aplicației în *Capitoul 3*, urmând ca implementarea modelului teoretic, împreună cu descrierea algoritmului de verificare a integrității și confidențialității să fie discutate în *Capitolul 4*. *Capitolul 5* se concentrează asupra prezentării interfeței grafice și a diverselor aspecte ale acesteia. Un scurt ghid al aplicației este trecut în revistă în *Capitolul 6*. Câteva exemple de protocoale de securitate, unele vulnerabile, altele sigure, sunt prezentate în *Capitolul 7*, urmând ca direcțiile de dezvoltare viitoare să fie surprinse în *Capitolul 8*.

Contribuții

În crearea aplicației s-a folosit ca sursă de inspirație modelul teoretic descris în [1] și [2]. Acesta a fost transpus în paradigma orientată obiect prin *modelarea, proiectarea și implementarea* claselor ce descriu elementele componente ale sistemului, precum și modul prin care interacționează între ele. Acest prim pas de modelare și proiectare arhitecturală a fost unul important, deoarece o aplicație de succes nu este caracterizată doar de algoritmii folosiți, ci și de factori precum cuplaj, coeziune etc., vizibili încă de la acest moment. Procesul este surprins în detaliu în *Capitolul 4.1.*

Funcționalitatea de încărcare a descrierii protocolului a fost realizată pornind de la fișiere de specificație în format text. Acestea respectă un anumit limbaj și de aceea a fost necesară crearea unei *gramatici de tip LL(k)* pentru a le descrie. Gramatica a fost folosită pentru a genera un parser top-down ce are ca scop verficarea sintactică a specificației descrise de utilizator. Aceasta este surprinsă în detaliu în *Capitolul 4.2.*

S-a implementat un algoritm (prezentat în detaliu în *Capitolul 4.4*) care verifică integritatea și confidențialitatea informațiilor împărtășite în desfășurarea protocolului, având în vedere un mediu cu atacator activ. Prin atacator activ înțelegem un agent extern sau intern protocolului (dându-se drept un agent dintr-o instanță de protocol). Acesta poate analiza fluxul de date și poate lua decizii de modificare a unor mesaje cu altele convenabile pentru a reuși să înțeleagă anumite mesaje cu „caracter sensibil".

De asemenea, a fost implementată și o interfață grafică pentru interacționarea cu modelul de date. Aceasta este împărțită în două submodule: un editor de specificații și un panou de control, de unde se poate folosi utilitarul de verificare. Modulul înglobează funcționalități specifice unui IDE (mediu de dezvoltare), precum colorarea sintactică a specificației. Mai multe detalii legate de interfața grafică sunt prezentate în *Capitolul 5.*

Anonimitatea în contextul ascunderii de informații

În cele ce urmează vor fi prezentate două metode pentru a obține anonimitatea în sistemele de calcul: Rețele de mix-uri și Rețele DC. Sursa de inspirație pentru conținutul prezentat în acest capitol este [2], ambele modele fiind introduse de către David Chaum în [4] și [5].

1.1 Rețele de mix-uri

D. Chaum introduce în [4] o metodă ce oferă comunicare anonimă folosind mix-uri. În mod intuitiv, un mix este un agent ce primește o mulțime de mesaje, le amestecă și le trimite la destinație într-o ordine diferită de cea în care le-a primit. Cu scopul de a se obține un grad mai mare de anonimitate procesul se iterează, mesajul trecând printr-o rețea de mix-uri.

În modelul rețelelor de mix-uri se folosesc chei publice. Pentru ca un agent să devină membru al rețelei este de ajuns ca acesta să genereze o pereche cheie publică - cheie privată, să o trimită pe cea publică celorlalți agenți din rețea și să anunțe faptul că dorește să se comporte ca un mix.

Metoda lui Chaum pentru a oferi anonimitate nu necesită o autoritate centrală. Folosind această metodă, sigura posibilitate de a compromite anonimitatea este aceea în care se corup toți agenții mix.

Una din problemele tratate în [4] este cazul în care, având un mesaj m_1 criptat cu cheia publică a unui agent A, rezultând $\{m_1\}_{K_A^e}$, un intrus ce deține un mesaj $\{m_2\}$ poate să îl cripteze tot cu cheia publică a lui A, iar dacă $\{m_1\}_{K_A^e} = \{m_2\}_{K_A^e}$ atunci intrusul poate deduce $\{m_1\}$=$\{m_2\}$. Pentru ca atacatorul să poată decide acest lucru, trebuie să cripteze toate mesajele posibile ce le poate primi A. Dacă mulțimea este mică, atunci atacatorul poate găsi mesajul $\{m_1\}$ chiar dacă nu cunoaște cheia privată a lui A. Pentru a rezolva această problemă, agentul care criptează mesajul $\{m_1\}$ poate adăuga un text aleator R_1. În acest caz, atacatorul poate genera un text R_2, dar este foarte puțin probabil ca $R_1 = R_2$.

În prezent s-au luat în considerare diferite metode de verificare dacă cheia privată a fost creată de către un agent ce semnează, de exemplu: trimițând mesajul necriptat și o semnătură a lui sau, mai bine, trimițând mesajul necriptat și semnând

doar un rezumat hash al acestuia.

Metoda din [4] se bazează pe două presupuneri:

- Nimeni nu poate să facă corespondența dintre mesajul criptat și versiunea sa necriptată, sau să creeze copii fără să aibă șirul aleator inițial sau cheia privată;
- Un atacator poate vedea expeditorul și destinatarul mesajului, poate să injecteze noi mesaje, sau să șteargă din cele existente.

Trimiterea anonimă a unui mesaj. Pentru a obține anonimitatea comunicării, mesajul este trimis la destinatar prin intermediul unei rețele de mixuri. Fiecare nod din rețea va procesa informația destinată lui și va trimite mai departe o parte din mesaj următorului nod din rețea. Acest proces continuă până când mesajul ajunge la destinație.

De exemplu, agentul A va fi considerat expeditor, agentul B va fi considerat destinatar, iar mix-urile din rețea vor fi numere naturale *1, 2,* ... etc.

Astfel, A trimite un mesaj anonim m agentului B folosind doar un singur mix: A criptează mesajul cu cheia publică a lui B, adăugând un șir de caractere R_0. Pentru ca mix-ul să știe unde să trimită mai departe mesajul $\{R_0, m\}_{K_{\mathrm{B}}^e}$, agentul A trebuie să adauge adresa agentului B. Rezultatul este criptat cu cheia publică a primului mix, K_1^e, adăugând un șir de caractere aleator, R_1. Mesajul trimis de A mix-ului 1 este:

$$\{R_1, \{R_0, m\}_{K_{\mathrm{B}}^e}, \mathrm{B}\}_{K_1^e}$$

Mix-ul 1 decriptează folosind cheia sa privată și obține R_1, mesajul $\{R_0, m\}_{K_{\mathrm{B}}^e}$ și identitatea celui care trebuie să îl primească.

Sunt identificate câteva probleme ale acestei abordări. Ele sunt prezentate în detaliu în [4]:

- Mix-ul nu trebuie să transmită mai departe doar un singur mesaj;
- Mix-ul nu trebuie să transmită mai departe mesaje clonate;
- Nu trebuie să existe doar un singur mix.

Răspunsul la un mesaj anonim. Tehnica trebuie să permită agentului B să raspundă mesajului primit de la agentul A, chiar dacă acesta nu îi cunoaște identitatea.

Pentru ca acest lucru să fie posibil, expeditorul, agentul A, va adăuga în mesajul m o adresă nedetectabilă de întoarcere și o cheie de criptare K_b, ce va fi folosită de B doar în această ocazie. Identitatea primului mix ce poate înțelege acea adresă nedetectabilă de întoarcere va fi de asemenea trimisă lui B. Adresa va conține chei pentru fiecare mix din drumul de întoarcere pentru ca mesajul să fie criptat în continuare.

Concluzii. Propunerea lui Chaum de a folosi rețele de mix-uri pentru a amesteca mesajele oferă anonimitate, dar adaugă și latență comunicării. Un alt model interesant construit pe aceeași idee este *Crowds*. În Crowds, fiecare mix cunoaște destinația mesajului, dar va decide în mod probabilistic dacă mesajul se trimite la destinație sau la un alt mix. Adresa nedetectabilă de întoarcere este adaugată la mesaj, iar modalitatea de răspuns este similară celei din rețele de mix-uri.

Un utilitar implementat folosind retele de mix-uri este TOR[1]. TOR este un program open source ce permite utilizatorilor să evite anumite forme de supraveghere a traficului pe rețea. Deși proiectul este despre asigurarea anonimității pe Internet, aceasta este adeseori înțeleasă greșit. Acesta este cazul *Deep Web-ului* sau *Darknet-ului* ce poate fi accesat printr-o simplă pagină de Wikipedia[2], oferind astfel servicii anonime, prin intermediul cărora pot fi angajați asasini plătiți sau pot fi accesate pagini web conținând pornografie infantilă.

[1] https://www.torproject.org/download/download.html.en
[2] kpvz7ki2v5agwt35.onion/wiki/index.php/Main Page

1.2 Rețele DC

Rețelele de mix-uri oferă anonimitate cu costul vitezei. După cum a fost explicat în secțiunea anterioară, fiecare mix adaugă o latență, în mare deoarece trebuie să se aștepte până ce se atinge un anumit prag pentru a se trimite toate mesajele. Procesarea fiecarui mesaj adaugă, de asemenea, latență.

Retelele DC[3] au fost introduse printr-o simplă poveste. Existau trei prieteni criptografi care luau cina la restaurantul lor preferat. Ospătarul îi anunță că meniul a fost plătit într-o manieră anonimă. Aceștia doresc să vadă dacă a plătit unul dintre ei sau agenția la care lucrează. Deoarece se respectă unii pe alții, în cazul în care a plătit unul din ei doresc să îi respecte decizia de a rămâne anonim. Pentru a face asta, vor juca jocul descris în cele ce urmează.

Fiecare criptograf va întoarce o monedă în mod secret și va împărți rezultatul cu colegul aflat în stânga lui. Astfel, fiecare va cunoaște rezultatul a două din cele trei monede și va ști dacă acestea au picat pe aceeași parte sau nu. Dacă unul dintre ei a plătit pentru cină, va spune inversul a ceea ce vede: dacă monedele erau pe aceeași parte va spune că sunt diferite, altfel va spune că erau la fel.

Pentru a înțelege cum este obținută anonimitatea, să presupunem că stema are valoarea 1, iar capul valoarea 0. Fiecare criptograf C_i are un secret: știe dacă a plătit sau nu. Secretul este reprezentat de o variabilă s_i, care are valoarea 1 dacă criptograful a plătit pentru cină, iar 0 dacă nu a plătit.

Sistemul poate fi modelat printr-o algebră booleană. La sfârșitul jocului fiecare criptograf va compune și va spune celorlalți o *disjuncție exclusivă* între cele două valori ale monedelor pe care le știe și valoarea secretului său. De exemplu, al doilea criptograf va cunoaște valoarea propriei monede, pe cea a primului criptograf, precum și valoarea secretului pe care vrea să îl împartă în mod anonim cu prietenii săi (s_2). Astfel, el îi va informa pe ceilalți doi colegi de valoarea expresiei $coin_1 \oplus coin_2 \oplus s_2$.

Cele trei monede pot pica în opt moduri diferite ($(coin_1, coin_2, coin_3) \in \{0,1\}^3$), dar doar două din aceste cazuri sunt relevante: când toate pică pe aceeași parte, sau când una dintre ele pică pe o parte diferită de celelate două. Astfel că se obțin următoarele expresii:

- C_1 va anunța $coin_3 \oplus coin_1 \oplus s_1$
- C_2 va anunța $coin_1 \oplus coin_2 \oplus s_2$
- C_3 va anunța $coin_2 \oplus coin_3 \oplus s_3$

[3]DC provine de la "Dining Cryptographers"

Fiecare criptograf va putea să calculeze valoarea:

$S = (coin_3 \oplus coin_1 \oplus s_1) \oplus (coin_1 \oplus coin_2 \oplus s_2) \oplus (coin_2 \oplus coin_3 \oplus s_3) = s_1 \oplus s_2 \oplus s_3$

Doar un singur criptograf ar fi putut plăti pentru cină, astfel că maxim un secret ar putea fi 1. Daca $S = 0$ asta înseamnă că niciunul dintre criptografi nu a plătit pentru cină, iar dacă $S = 1$ înseamnă că plata a fost făcută de unul dintre ei.

Verificare. Din punctul de vedere al lui C_2, dacă $S = 1$ el știe că unul dintre prietenii lui a plătit cina, dar nu știe exact care.

La finalul protocolului C_2 va cunoaște:

- $coin_1$
- $coin_2$
- s_2
- $coin_3 \oplus coin_1 \oplus s_1$
- $coin_2 \oplus coin_3 \oplus s_3$

Având atât $coin_1$ cât și $coin_3 \oplus coin_1 \oplus s_1$ și folosind disjuncție exclusivă criptograful C_2 poate afla valoarea expresiei $coin_3 \oplus s_1$. Folosind același raționament, poate afla și $coin_3 \oplus s_3$. Dar, deoarece nu știe valoarea lui $coin_3$, acesta nu îl poate elimina pentru a afla s_1 sau s_3.

De exemplu, dacă C_2 obține $coin_3 \oplus s_1 = 1$ și $coin_3 \oplus s_3 = 0$ atunci, din punctul său de vedere, valorile ar putea fi obținute din unul din cele două cazuri: $coin_3 = 0, s_1 = 1, s_3 = 0$ sau $coin_3 = 1, s_1 = 0, s_3 = 1$.

Generalizarea problemei DC. În cazul clasic, protocolul a fost jucat de doar trei participanți. Fiecare monedă avea rolul unui bit cheie împărțit între doi agenți. Secretul ce trebuie împărțit în mod anonim între criptografi era și el modelat printr-un bit. Protocolul trebuie extins la rețele cu mai mulți participanți și cu mesaje de lungime mai mare. Astfel că au fost aduse niște îmbunătățiri modelului.

Pentru ca agenții să poată trimite în mod anonim mesaje mai mari, protocolul trebuie să fie jucat în mai multe runde. Fiecare agent poate împărți chei de lungime mai mare cu partenerii lor. Modelul poate fi extins și în privința numărului de participanți. O abordare simplă în acest sens este descrisă în [6], unde generalizarea pentru rețele DC este compusă din n agenți ce stau la o masă rotundă.

În rețele DC există problemele enumerate mai jos. Acestea, precum și posibile soluții la unele dintre ele, sunt prezentate mai în detaliu în [2].

- Dacă un criptograf a plătit pentru cină, este necesar doar ca vecinii lui să colaboreze pentru a afla identitatea acestuia;
- Pot apărea coliziuni dacă nu se stabilește un mecanism pentru ordinea în care agenții trimit mesajele;
- Poate exista problema înțelegerilor secrete. Mai mulți agenți își pot împărtăși cunoștințele pentru a înlătura anonimitatea.

Fundamente teoretice

Modelul aplicației este similar cu cel din [1] fiind descris inițial în [7]. În acest capitol vor fi prezentate acele elemente teoretice care au fost implementate în cadrul aplicaţiei *„Security Protocols Checker”*.

2.1 Formatul unui protocol

Formatul unui protocol de securitate este reprezentat de un triplet $\mathcal{S} = (\mathcal{A}, \mathcal{K}, \mathcal{N})$, constând dintr-o mulţime finită $\mathcal{A}$ de *agenţi* şi două mulţimi cel mult numărabile $\mathcal{K}$ - *chei* şi $\mathcal{N}$ - *nonce*-uri.

Se presupune că:

- Mulţimea $\mathcal{A}$ conţine un element special notat cu I, numit *intrus*. Toate celelalte elemente ale mulţimii se numesc agenţi oneşti, iar submulţimea reprezentată de aceştia este notată cu H_0;

- $\mathcal{K} = \mathcal{K}_0 \cup \mathcal{K}_1$, unde $\mathcal{K}_0$ este mulţimea de chei pe termen scurt, iar $\mathcal{K}_1$ e o mulţime finită de chei pe termen lung. Elementele din $\mathcal{K}_1$ sunt *chei publice* (K_A^e - cheia publică a agentului A), *chei private* (K_A^d - cheia privată a agentului A), sau *chei partajate* (K_{AB} – cheia partajată de agenţii A şi B).

  Avantajul criptării folosind cheia publică a unui agent A intervine atunci când un alt agent doreşte să îi trimită un mesaj care să nu fie înţeles din exterior. Cum A este singurul care cunoaşte cheia privată corespunzătoare cheii sale publice, doar el poate decripta şi înțelege mesajul.

  Pe de altă parte, criptarea folosind cheia privată a unui agent A se realizează de către acestea în momentul în care doreşte să autentifice un mesaj transmis altui agent. Destinatarul va primi mesajul şi semnătura lui A asupra mesajului, va decripta semnătura folosind cheia publică a lui A şi va compara rezultatul cu mesajul primit. Cum A este singurul care cunoaşte cheia sa privată, el este singurul care ar fi putut crea o semnătură validă şi, deci, mesajul primit este autentic.

- Unii agenţi oneşti pot primi de la început o serie de informaţii numite cunoştinţe, $Secret_A \subset \mathcal{K}_0 \cup \mathcal{N}$, ce nu sunt cunoscute de către intrus.

2.2 Termi

Mulţimea de termi de bază este $\mathcal{T}_0 = \mathcal{A} \cup \mathcal{K} \cup \mathcal{N}$. Mulţimea $\mathcal{T}$ a termilor este definită inductiv astfel:

- Fiecare term de bază este term;
- Dacă t_1 şi t_2 sunt termi, atunci (t_1, t_2) este term;
- (t_1, t_2) este extins la $(t_1, \ldots, t_n)$ considerând că $(t_1, \ldots, t_n) = ((t_1, \ldots, t_{n-1})\ t_n)$, pentru orice n ≥ 3 şi omiţând parantezele;
- Dacă t este term şi K este o cheie, atunci $\{t\}_K$ este term.

Dacă t este un term, *Sub(t)* reprezintă mulţimea *subtermilor* lui t, iar $Sub(\{t_1, \ldots, t_n\}) = Sub(t_1) \cup \ldots \cup Sub(t_n)$.

Se adoptă o *presupunere de criptare perfectă*, care susţine că un mesaj criptat cu o cheie K poate fi decriptat doar de un agent care cunoaşte cheia inversă corespunzătoare lui K, notată K^{-1}, iar singurul mod de a crea mesajul $\{t\}_K$ este prin criptarea lui t cu cheia K. Cheia inversă a unei chei simetrice este chiar ea, cheia inversă a unei chei publice este cheia privată, iar cheia inversă a unei chei private este cheia publică.

2.3 Acţiuni

Există două tipuri de acţiuni, de trimitere şi recepţionare. O acţiune de trimitere are forma $A!B{:}(M)t$, iar una de recepţionare are forma $A?B{:}t$, unde A este un agent onest care realizează acţiunea, $A \neq B$, $t \in \mathcal{T}$, iar $M \subseteq \text{Sub}(t) \cap (\mathcal{N} \cup \mathcal{K}_0)$ este mulţimea noilor termi ai acţiunii (termi care au fost generaţi pentru a compune mesajul trimis).

2.4 Protocoale

Un *protocol de securitate* este definit printr-un triplet $\mathcal{P} = (\mathcal{S}, \mathcal{C}, \omega)$, unde $\mathcal{S}$ este *formatul protocolului*, $\mathcal{C}$ este o submulţime a lui $\mathcal{T}_0$ (numită *mulţimea de constante* a lui $\mathcal{P}$), iar ω este o mulţime nevidă de acţiuni, numită *corpul protocolului*, astfel încât nicio acţiune din ω nu implică intrusul.

$$A \,!\, B \;\; : (\{N_{1_A}\})\ \{N_{1_A}\}K^e_B$$
$$B \,?\, A : \{N_{1_A}\}K^e_B$$
$$B \,!\, A \;\; : \{N_{1_A}\}K^e_A$$
$$A \,?\, B : \{N_{1_A}\}K^e_A$$

Protocol 1: Vulnerabil la confidențialitate

Rezultat 1: Modalitatea de atac asupra *Protocolului 1*

Agents commons to both protocol instances: B

———Instance 1———	———Instance 2———
$A \,!\, B \;\; : (\{N_{1_A}\})\ \{N_{1_A}\}K^e_B$	$I!B{:}(\{N_{1_I}\})\{N_{1_A}\}K^e_B$
A generated $\{N_{1_A}\}$ for B	I sends the same message that was sended in the same action of the previous protocol instance
A sent $\{N_{1_A}\}K^e_B$ to B	
I received $\{N_{1_A}\}K^e_B$	All this because later will discover $[N_{1_A}]$ from the other protocol instance
$B \,?\, A : \{N_{1_A}\}K^e_B$	
B received $\{N_{1_A}\}K^e_B$ from A	$B?I{:}\{N_{1_A}\}K^e_B$
B received N_{1_A} from A	B received $\{N_{1_A}\}K^e_B$ from I
	B received N_{1_A} from I
$B \,!\, A \;\; : \{N_{1_A}\}K^e_A$	
B sent $\{N_{1_A}\}K^e_A$ to A	$B!I{:}\{N_{1_A}\}K^e_I$
I received $\{N_{1_A}\}K^e_A$	B sent $\{N_{1_A}\}K^e_I$ to I
	I received $\{N_{1_A}\}K^e_I$
$A \,?\, B : \{N_{1_A}\}K^e_A$	
A received $\{N_{1_A}\}K^e_A$ from B	$I?B{:}\{N_{1_I}\}K^e_I$
A received N_{1_A} from B	I received $\{N_{1_A}\}K^e_I$ from B
	I received N_{1_A} from B

În *Protocolul de securitate 1*, agentul A generează un *nonce*, îl trimite agentului B și apoi așteaptă confirmarea de la acesta. Problema intervine atunci când există două instanțe ale protocolului, iar agentul B este același în ambele.

Un atacator poate să se dea drept agentul A în instanța a doua. Acesta poate să copieze mesajul din prima acțiune a primei instanțe de protocol si să îl trimită

la rândul său în prima acțiune a celei de-a doua instanțe. Astfel pune în pericol proprietatea de *confidențialitate*, reușind să găsească *nonce*-ul generat de agentul A în prima instanță. Acest lucru se datorează faptului că agentul A nu își criptează și identitatea pe lângă mesaj în prima acțiune. Atacatorul, înainte să trimită mesajul copiat, schimbă identitatea expeditorului inițial cu a lui, până la nivelul la care are acces. Adăugarea criptată a identității este suficientă pentru a repara breșa de securitate.

Arhitectura aplicaţiei

Aplicaţia *„Security Protocols Checker”* a fost realizată folosind limbajul *Java 1.8* împreună cu tehnologia *JavaFX* și cu modelul arhitectural *MVC*[8] (model-view-controller). Acesta are rolul de a separa modelul aplicației de interfața grafică, oferind astfel posibilitatea dezvoltării în paralel a modulelor.

În *Figura 1* se pot evidenția cele două module mari, *interfaţa grafică* şi *modelul de date*, fiecare conţinând la rândul său diverse submodule. *Panoul de control* al interfeței grafice comunică cu controller-ul aplicației în timp ce view-ul, principala modalitate de vizualizare a specificațiilor de protocol și a rezultatelor verificării acestora, este reprezentat de *Editor*.

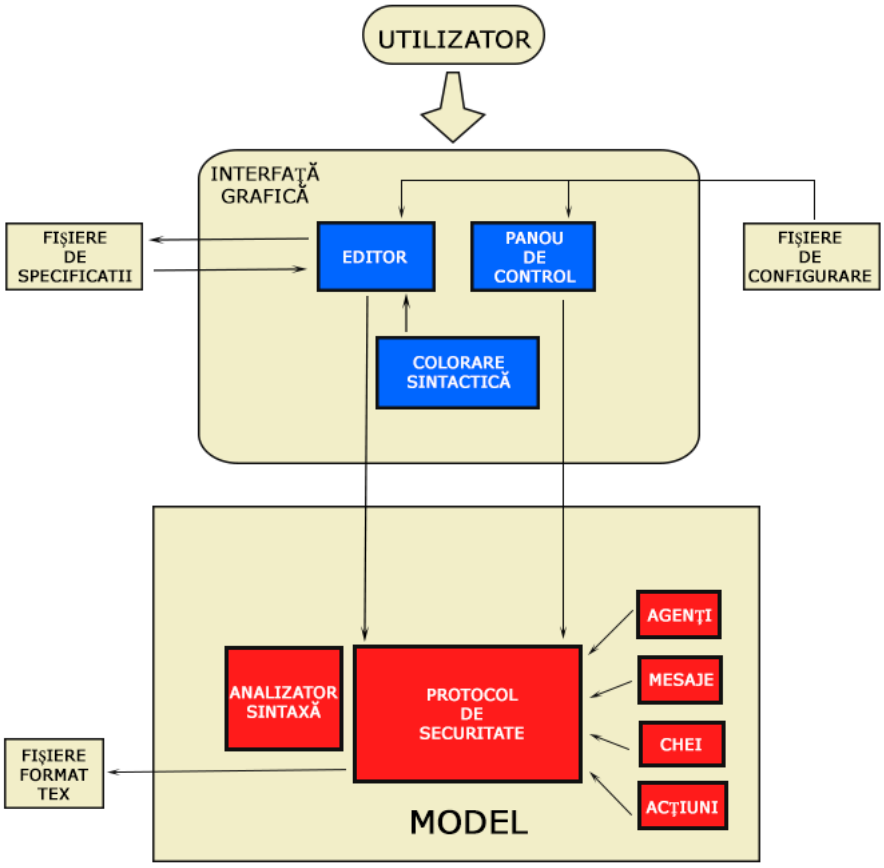

Figura 1: Arhitectura aplicaţiei

Interfaţa grafică este principala modalitate prin care utilizatorul poate interacţiona cu aplicaţia. Aceasta este formată din două submodule principale: un *editor* şi un *panou de control*. Editorul este folosit pentru a crea, deschide, modifica şi verifica sintactic şi semantic fişiere de specificaţii oferite de utilizator ca date de intrare

aplicației. *Panoul de control* are rolul de a oferi utilizatorului acces la acțiunile prezentate mai jos. Acestea sunt valabile odată cu încărcarea cu succes a unui protocol (ceea ce se întamplă dacă acesta este *valid din punct de vedere sintactic și semantic*).

- *Să ruleze* protocolul de securitate, observând dacă acesta este rezistent la atacuri produse de un intrus activ. Poate afla de asemenea și modalitatea prin care aceasta a reușit să compromită securitatea protocolului, acțiune cu acțiune, în cazul negativ. Altfel, dacă protocolul este sigur se afișează execuția acestuia pas cu pas;

- *Să genereze automat cod latex* pentru a transforma specificația protocolului, și/sau rezultatul verificării, în funcție de preferințele utilizatorului, din sintaxa folosită intern de aplicație în sintaxa folosită în articolele de specialitate, precum [1]. Scopul acestei funcționalități este de a oferi imediat codul latex necesar pentru a fi inclus într-o lucrare, fară a fi necesar sa fie scris manual (exemplele de protocoale din această lucrare au codul latex generat astfel);

- *Să genereze automat și să deschidă un fișier în format .pdf* ce conține protocolul/rezultatul execuției pentru un preview, folosind codul latex generat prin metoda prezentată anterior. Generarea fișierului cu extensia pdf necesită utilitarul *MiKTeX*[1] instalat.

De asemenea, modulul *Interfața grafică* mai conţine şi o componentă care realizează colorarea sintactică a conţinutului fişierelor de specificaţii din editor. Modalitatea de colorare a textului va fi prezentată în detaliu în *Capitolul 5.4*.

Modelul de date conține implementarea protocolului de securitate, împreună cu submodule auxiliare, precum cel responsabil de *verificarea sintactică* a specificaţiei unui protocol. De asemenea, tot aici putem aminti și submodulul ce se ocupă de *verificarea semantică* a specificaţiei unui protocol. Acestea sunt folosite la momentul încărcării unui protocol în panoul de control, pentru a informa utilizatorul în legătură cu validitatea specificaţiilor protocolului ce se încearcă a fi încărcat.

Un submodul de o importanță majoră al modelului de date este cel responsabil cu implementarea strategiei de verificare a corectitudinii protocolului de securitate în circumstanțele rulării într-un mediu ce conține un atacator activ.

[1] MiKTeX, utilitar folosit pentru generarea fișierelor *.pdf* - `http://miktex.org/`

Funcționalitatea de încărcare în *panoul de control* a unui protocol, pornind de la specificaţia descrisă de catre utilizator, este precedată de o etapă de *interpretare a textului* şi *instanţiere* a anumitor clase pentru a obține elementele componente ale acestuia. Etapa de încărcare a protocolului este sintetizată în metoda statică `loadProtocol()` a clasei `Protocol`.

În capitolele următoare vor fi prezentate detaliile implementării modulelor principale şi ale submodulelor acestora. Pentru unele dintre ele se vor trece în revistă și câteva fundamente teoretice cu scopul de a clarifica explicațiile.

Transpunerea modelului în implementare

În acest capitol se vor trece în revistă detalii despre clasele cele mai importante ce intră în componența modelului de date, punând accentul pe proiectarea funcţionalităţilor principale și pe relaţiile dintre acestea, având în vedere şabloanele de proiectare folosite. Diagramele de clase au fost create folosind ArgoUML[1]. Scopul lor este acela de a oferi o imagine mai clară asupra aspectelor ce vor fi discutate.

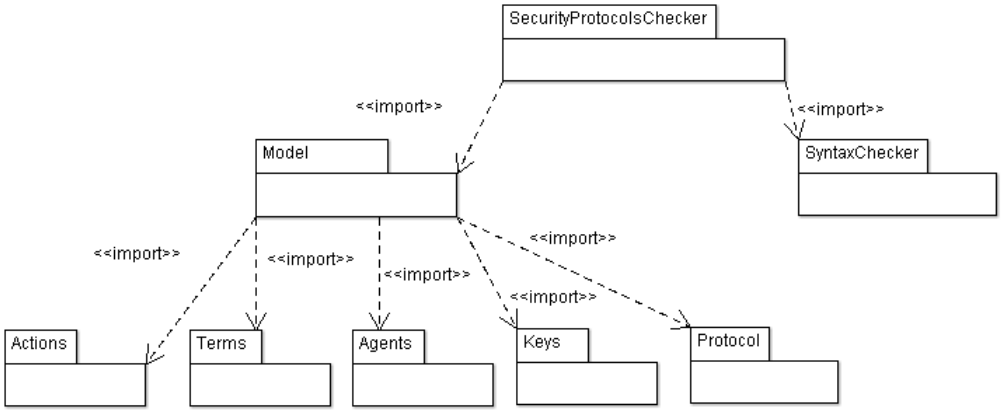

Figura 2: Ierarhia pachetelor modelului de date

Figura 2 prezintă ierarhia pachetelor ce conţin implementarea modelului teoretic. Pachetul `AnonymityChecker.Model` este compus din clasele reprezentând obiectele agregate de protocolul de securitate, iar pachetul `AnonymityChecker.SyntaxChecker` conține clasele folosite în verificarea sintactică a specificației oferite de utilizator.

4.1 Modelare şi proiectare

4.1.1 Şablonul de proiectare *Composite*. Abstractizarea termilor

În *Capitolul 2.2* s-a definit inductiv mulţimea termilor modelului teoretic ca fiind formată din termi de bază (agenţi, chei, nonce-uri), termi compuşi şi termi criptaţi. În *Figura 3* se prezintă ierarhia de clase ce reprezintă această mulțime în programarea orientată obiect.

Având în vedere modelul teoretic ce impune faptul că orice term de bază este term şi orice term compus este term, rezultă că termii compuşi ar trebui reprezentați prin intermediul unor structuri arborescente, recursive. Accesul la termi trebuie să se poată face prin intermediul unui layer de abstractizare.

Soluţia la această problemă este utilizarea şablonului de proiectare *Composite*.

[1]ArgoUML, utilitar open source de modelare a diagramelor UML `http://argouml.tigris.org/`

Scopul acestui şablon structural este de a reprezenta ierarhii parte-întreg, oferind clientului posibilitatea de a trata obiectele individuale şi ierarhiile de obiecte în mod uniform[9]. Clasele `CompoundTerm` şi `EncryptedTerm` au fost implementate folosind șablonul de proiectare Composite.

- `CompoundTerm` reţine într-o listă referinţe la instanţe ale clasei `Term`, reprezentând termii componenţi (care pot fi la rândul lor termi de bază sau termi compuşi);
- `EncryptedTerm` reţine o referinţă la o cheie şi o referinţă la o instanţă a clasei `Term`, reprezentând termul criptat cu cheia respectivă.

Relaţia de compunere dintre clasele `CompoundTerm` şi `Term` este de tip 1 la n, iar cea dintre `EncryptedTerm` şi `Term` este de tip 1 la 1.

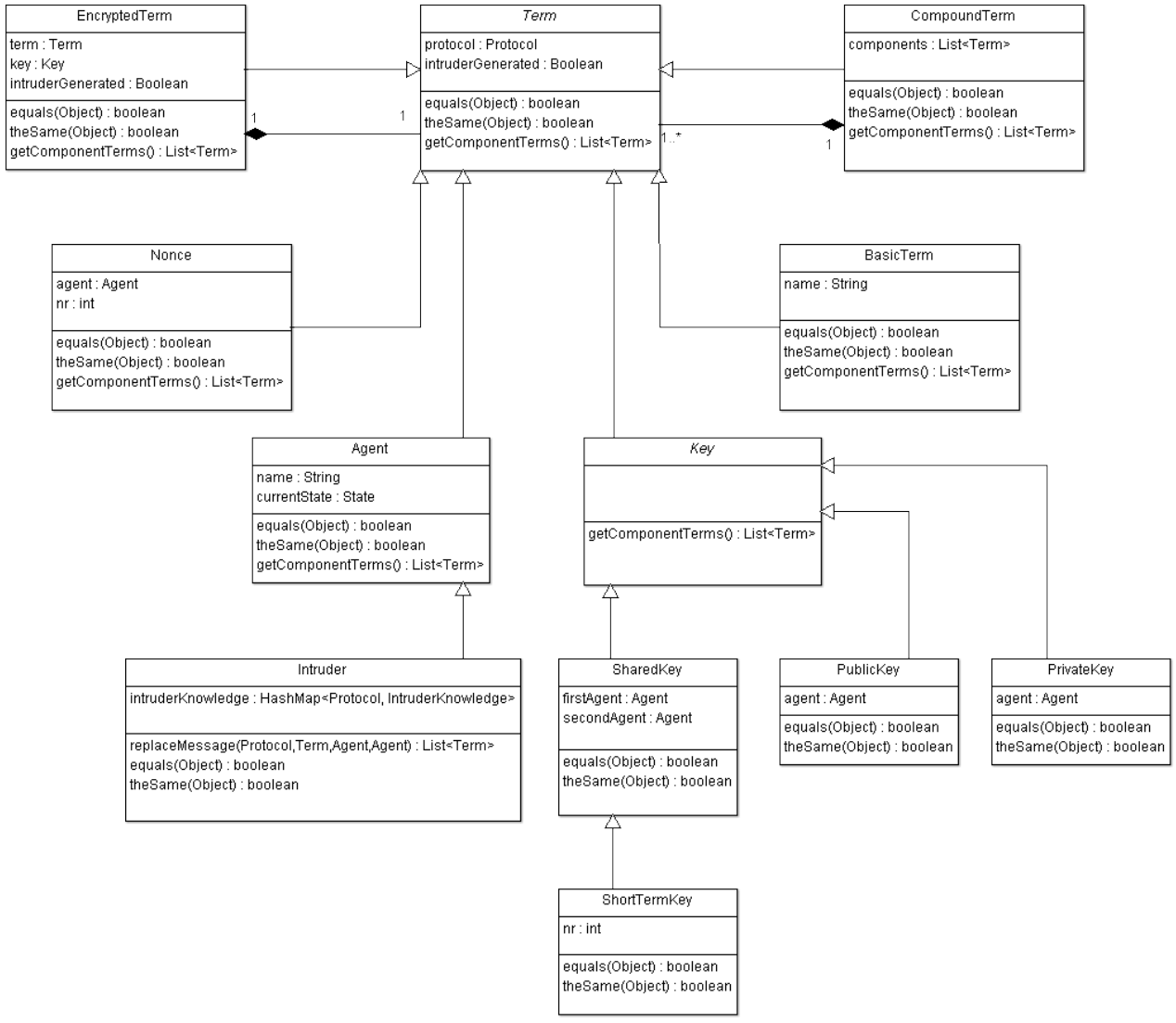

Figura 3: Ierarhia claselor ce reprezintă termii

4.1.2 Şablonul de proiectare *Singleton*. Atacatorul

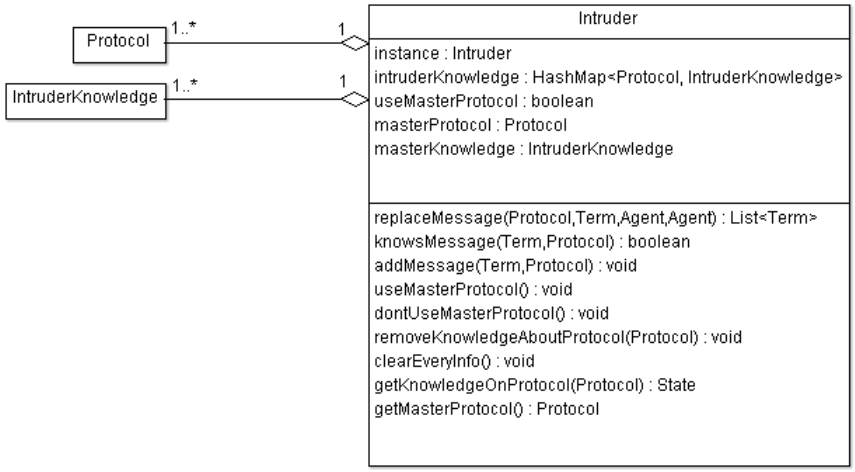

Figura 4: Diagrama clasei `Intruder`

Clasa `Intruder`, ilustrata in *Figura 4*, este responsabilă cu stabilirea posibililor termi cu care se poate inlocui un term la un moment al execuției unui protocol, funcționalitate implementată în metoda `replaceMessage()`.

Aceasta agregă obiecte de tip `Protocol` și `IntruderKnowledge` deoarece reține referințe precum `masterProtocol`, `masterKnowledge` și `intruderKnowledge` (o colecție de tip `HashMap<Protocol, IntruderKnowledge>` ce asociază obiectelor din clasa `Protocol` obiecte din clasa `IntruderKnowledge`) reprezentând cunoștințele pe care le are atactorul despre fiecare instanță a clasei `Protocol` din colecție. Se va reveni la acestea și la modul în care interacționează la momentul analizei algoritmului rulării protocolului într-un mediu cu atacator activ.

Șablonul de proiectare *Singleton* a fost utilizat din necesitatea existenței unui singur atacator, care să fie accesibil de oriunde, pentru a putea influența instanțele de protocol ce rulează la un anumit moment. Acest şablon creaţional asigură faptul că o clasă poate avea o singură instanţă şi expune un punct de acces global către instanţa respectivă[9].

4.2 Definirea specificaţiei. Validarea sintactică şi semantică

Protocoalele sunt definite folosind fișiere de specificație, ce vor fi încărcate în cadrul aplicației. Structura unui astfel de fișier este următoarea:

- se enumeră *agenții* ce participă la rularea protocolului împreună cu câte o listă de *cunoștințe inițiale* pentru fiecare agent;
- se descriu *acțiunile* ce vor avea loc în cadrul protocolului.

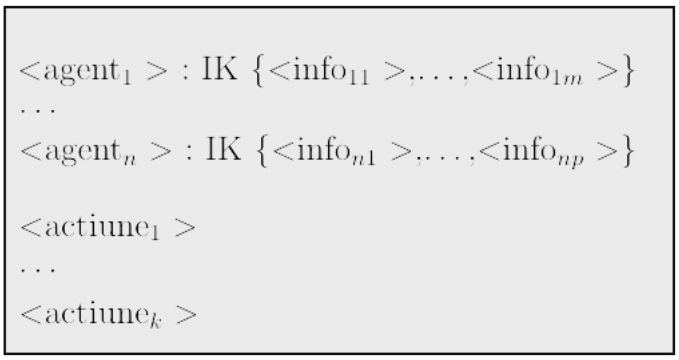

Figura 5: Specificaţia unui protocol

Elementele de tip <*info*> pot fi nonce-uri, chei sau termi de bază (tichete), reprezentaţi doar prin nume (vor fi instanţe ale clasei `BasicTerm`). Elementele de tip <*acțiune*> reprezinta acțiuni de trimitere sau recepționare și pot avea urmatoarele forme, în funcție de tipul lor:

<*agent1*> ! <*agent2*> : (<*termiGeneraţi*>) <*mesaj*>

<*agent1*> ? <*agent2*> : <*mesaj*>

Încărcarea protocolului trebuie precedată de o verificare a corectitudinii sintactice precum și a celei semantice.

4.2.1 Verificarea sintactică

Pentru validarea sintactică, am analizat structura specificaţiei, am observat regulile după care este construită și am decis să creez o gramatică pentru a o descrie şi să generez un analizor de sintaxă pornind de la aceasta.

În acest scop, am folosit *Java Compiler Compiler*[2] (JavaCC), un utilitar scris în limbajul Java şi care, pornind de la o gramatică, generează un program Java care recunoaşte dacă un anumit text aparţine limbajului generat de acea gramatică. Acesta creează clase ce au fost incluse în aplicaţie în pachetul `AnonymityChecker.SyntaxChecker`, împreună cu fișierul ce conține descrierea gramaticii, `Checker.jj`. JavaCC, spre deosebire de alte instrumente de tipul YACC[3], generează analizoare sintactice des-

[2]Java Compiler Compiler tm (JavaCC^{tm}) - The Java Parser Generator - `http://javacc.java.net/`
[3]Yet Another Compiler-Compiler - `http://dinosaur.compilertools.net/yacc/`

cendente (top-down). Un avantaj ar fi acela că permite definirea unor gramatici mai generale, însă recursia stângă este interzisă. JavaCC generează programe ce analizează gramatici de tip LL(1). Însă, pot fi cazuri în care porţiuni ale gramaticii să nu fie LL(1), atunci când, de exemplu, avem elemente pentru a căror diferenţiere este necesară analizarea a mai mult de un token. În astfel de situaţii, JavaCC permite adăugarea unui lookahead[4] sintactic sau semantic pentru rezolvarea conflictelor de tip FIRST – FIRST[5]. Analizorul generat va fi LL(k) doar pentru acele porţiuni care conţin lookahead şi LL(1) în rest, pentru o mai bună performanţă.

Terminalii gramaticii sunt agenţi, nonce-uri, chei, termi de baza, precum și alte elemente referitoare la sintaxă. Pentru identificarea lor am scris următoarele expresii regulate:

```
TOKEN: {
        <INIT_KNOWLEDGE: "IK">                  |
        <AGENT: ["A"-"H"] | "J" | ["L"-"Z"]>    |
        <INTRUDER: "I">                         |
        <BASIC_TERM: (["a"-"z"])+>              |
        <NONCE: "N_"<AGENT>"_"<NUMBER>>         |
        <PUBLIC_KEY: "KE("<AGENT>")">           |
        <PRIVATE_KEY: "KD("<AGENT>")">          |
        <SHARED_KEY: "K("<AGENT>","<AGENT>")">  |
        <SHORT_TERM_KEY: "K(" <NUMBER> ")">     |
        <SEND: "!">                             |
        <RECEIVE: "?">                          |
        <NUMBER: ["1"-"9"](["0"-"9"])*>
}
```

Primele două secțiuni ale gramaticii stabilesc faptul că aceasta impune cel puțin două declarații de agenți și cel puțin o acțiune.

```
void START(): {} {PROTOCOL()<EOF> }

void PROTOCOL(): {}{
        DECLARATION()
        (LOOKAHEAD(2) DECLARATION())+
        (ACTION())+
}
```

[4]Lookahead - https://javacc.java.net/doc/lookahead.html
[5]Conflicte LL(1) - https://parasol.tamu.edu/~rwerger/Courses/434/lec10.pdf

Secțiunea `DECLARATION()` are scopul de a stabili strucutura primei părți a protocolului, în care se descriu agenții, precum și cunoștințele lor inițiale, atacatorul fiind privit de asemenea ca un agent.

```
void DECLARATION(): {}{
        <AGENT> ":" <INIT_KNOWLEDGE> KNOWLEDGES() LINES()    |
        <INTRUDER> ":" <INIT_KNOWLEDGE> KNOWLEDGES() LINES()
}
```

Secțiunea `ACTION()` descrie componența celor două tipuri de acțiuni, *send* și *receive*. A fost necesară folosirea unui `Lookahead(2)`, deoarece acțiunile se diferențiază abia după elementul de pe a doua poziție. Aici gramatica nu mai este de tip LL(1), ci de tip LL(2).

```
void ACTION():{}{
        LOOKAHEAD(2) <AGENT> <SEND> <AGENT> ":"(GENERATED())? MESSAGE() LINES() |
        LOOKAHEAD(2) <AGENT> <RECEIVE> <AGENT> ":" MESSAGE() LINES() }
```

Următoarele secțiuni vor cuprinde descrierile elementelor componente ale acțiunilor, împreună cu modul de formare a acestora. De exemplu, un mesaj descris de către `MESSAGE()` este format dintr-un mesaj compus urmat sau nu de o cheie, semnificând posibilitatea de a fi criptat sau nu. Acesta se poate identifica printr-o listă de termi, care, la rândul lor, pot fi terminali ai gramaticii (agenți, chei, nonce-uri, termi de bază sau term compus urmat sau nu de o cheie). Se evidențiază șablonul de proiectare Composite prezentat în *Capitolul 4.1.1.*

```
void GENERATED():{}{"({"GENERATED_TERM()(","GENERATED_TERM())*"})"}
void GENERATED_TERM(): {}{ <BASIC_TERM> | <NONCE> | <SHORT_TERM_KEY> }

void MESSAGE():{}{ COMPOUND_TERM() (KEY())? }
void COMPOUND_TERM():{}{ "{" TERM() ("," TERM())* "}"}
void TERM() :{}{ <AGENT> | KEY() | <NONCE> | <BASIC_TERM> |
                COMPOUND_TERM() (KEY())? }
void KEY(): {}{ <PUBLIC_KEY>|<PRIVATE_KEY>|<SHARED_KEY>|<SHORT_TERM_KEY> }
void LINES():{}{ ("\n")* }
```

4.2.2 Verificarea semantică

Responsabilă cu validarea semnatică a specificației este metoda `validate()` din cadrul clasei `Protocol`. Aceasta efectuează teste precum:

- pentru fiecare informație deținută de fiecare agent se verifică:
  - dacă este cheie/nonce ce aparține unui agent nedefinit în protocol (de exemplu, agentul A cunoște cheia publică a agentului P, dar in protocol nu a fost definit agentul P);
  - dacă este un agent nedefinit în protocol ;
  - dacă este cheie privată, dar nu aparține agentului ce o deține;

- dacă este cheie partajată dar agentul ce o deține nu este niciunul din cei referiți de către cheie.

- pentru fiecare cheie se verifică:
  - dacă este cheie publică trebuie ca toți agenții să o cunoască;
  - dacă este cheie privată atunci agentul referit de către cheie trebuie să o cunoască însă toți ceilalți nu trebuie;
  - dacă este cheie partajată verificăm dacă toți agenții referiți de către cheie o cunosc.

- pentru fiecare acțiune din protocol:
  - dacă este o acțiune de trimitere
    - verificăm dacă agenții sunt definiți;
    - verificăm dacă mesajul generat conține termi care nu pot exista;
    - verificăm dacă mesajul trimis conține termi care nu pot exista;
    - verificăm dacă cel care trimite mesajul știe toate componentele acestuia.
  - dacă este o acțiune de recepționare
    - verificăm dacă agenții sunt definiți;
    - verificăm dacă mesajul primit conține termi care nu pot exista.

4.3 Încărcarea unui protocol pornind de la o specificaţie

Verificarea sintactică şi semantică se realizează înainte de transforma protocolul descris în fișierul de specificație în instanțe ale claselor prezentate anterior. Dacă cel puțin una din cele două verificări eșuează, atunci încărcarea nu va mai avea loc, iar utilizatorul va fi informat prin intermediul unor mesaje de eroare de motivele pentru care specificația nu este corectă.

Pe de altă parte, dacă specificaţia este validă sintactic şi semantic, va fi apelată metoda statică `loadProtocol()` din cadrul clasei `Protocol`. Există posibilitatea obținerii mai multor instanțe de protocol, plecând de la aceeași specificație, folosind metoda statică `loadNProtocols()`. Interpretarea specificației se realizează cu ajutorul unor expresii regulate ce sunt membri statici ai claselor a caror obiecte le descriu. Acestea se folosesc cu scopul de a extrage treptat informații din specificație. În acest scop, sunt utilizate clasele `Pattern` şi `Matcher` din cadrul pachetului `java.util.regex`.

4.4 Algoritmul de verificare a rezistenței la un intrus activ

În acest capitol se va prezenta algoritmul de detecție a posibilelor atacuri produse de un intrus activ. Acesta va fi împărțit în trei subcapitole ce vor trece în revistă pașii preliminari algoritmului, algoritmul propriu-zis, precum și subrutina de tratare a deciziei conform căreia, având un mesaj, trebuie găsite mesajele cu care poate fi înlocuit pentru a afecta proprietățile de securitate ale protocolului.

4.4.1 Pașii preliminari algoritmului

Presupunând că există o specificație ce a fost declarată validă din punct de vedere sintactic și semantic prin procedeele descrise în *Capitolul 4.2*, trebuie urmăriți anumiți pași bine definiți:

- Atacatorul trebuie să își reseteze variabilele membru (trebuie să șteargă informațiile din colecții). Procedeul este necesar în cazul în care aceasta nu e prima rulare a utilitarului, lucru care se face prin apelarea metodei `clearEveryInfo()` din clasa `Intruder`;

- Trebuie creată o instanță de protocol. Înainte de încărcarea oricărei instanțe de protocol, atacatorul trebuie să apeleze metoda `dontUseMasterProtocol()`. Apelul are rolul de a seta comportamenul atacatorului pentru a nu interveni în procesul de încărcare a protocolului. Apoi se încarcă instanța efectivă prin apelul metodei statice `loadProtocol()` din clasa `Protocol`.

  Această instanță va fi înregistrată de atacator la momentul rulării ca `masterProtocol`, iar cu ea se vor schimba mesajele luate de la instanța victimă. În acest scop, următorul pas este acela de setare a comportamentului prin metoda `useMasterProtocol()`.

  Astfel, instanța rulată va fi inregistrată ca `masterProtocol` și, din acest motiv, apelăm metoda `run()` din clasa `Protocol` fără niciun parametru, ceea ce semnifică faptul că nu se aplică algoritmul ce va fi prezent mai jos pe această instanță. Altfel spus, ea va rula de la prima la ultima acțiune fără intervenții externe.

  Se încarcă instanța căreia i se va aplica algoritmul de verificare prin procedeul descris mai sus (cele două apeluri de metode `dontUseMasterProtocol()` și `loadProtocol()`). În acest moment, sistemul este pregătit pentru testarea protocolului, folosind instanța creată anterior.

În *Codul sursă 1* este prezentată metoda `isProtocolSafe()`. Aceasta primește specificația unui protocol de securitate valid și aplică algoritmul de verificare a proprietăților de securitate. Metoda întoarce *true* dacă protocolul este sigur, iar *false* în caz contrar.

```
1     private boolean isProtocolSafe(String protocolDesc) {
2
3         //  pașii preliminari algortmului de rulare
4         Intruder intruder = Intruder.getInstance();
5
6         intruder.clearEveryInfo();
7
8         intruder.dontUseMasterProtocol();
9         Protocol masterProtocol = Protocol.loadProtocol(protocolDesc, null);
10
11        intruder.useMasterProtocol();
12        masterProtocol.run();
13
14        intruder.dontUseMasterProtocol();
15        Protocol myProtocol = Protocol.loadProtocol(protocolDesc, null);
16
17
18        //  rularea efectivă a instanței de protocol
19        myProtocol.run(1);
20        if(myProtocol.getLog().isCorrect()){
21
22            myProtocol.reset();
23            myProtocol.run(2);
24            if (myProtocol.getLog().isCorrect()){
25
26                return true;
27            }
28        }
29
30        return false;
31    }
```

Cod sursă 1: `isProtocolSafe()`

4.4.2 Algoritmul de verificare a rezistenței la un intrus activ

Verificarea unei instanțe din clasa `Protocol` presupune aplicarea a două teste asupra acesteia folosindu-se apelurile de metode `run(1)`, `reset()` și `run(2)`. Desigur că ultimele două apeluri au sens doar dacă nu s-au găsit probleme la primul test, lucru ce se verifică printr-un test asupra rezultatului expresiei `instanțaTest.getLog()` `.isCorrect()`. Rularea metodei `run()` cu cei doi parametri, `1` și `2`, acoperă două strategii de atac (prima mai generală surprinzând o arie largă de atacuri, iar a doua particulară). Acestea vor fi trecute în revistă în urmatoarele paragrafe.

Procesul de rulare al protocolului de securitate implică parcurgerea fiecărei acțiuni, de la prima la ultima, precum și luarea anumitor decizii. Cu acest scop au fost create în clasa `Protocol` metodele `runSendAction()` și `runReceiveAction()`. În consecință, algoritmul de detectare intervine atunci când se primește un mesaj, deci în cazul metodei `runReceiveAction()`, aceasta fiind diferită în cele două cazuri, pe când metoda `runSendAction()` este aceeași.
Metoda `runSendAction()` este compusă din mai mulți pași:

- Veficarea posibilității transmiterii mesajului (se cunosc toate componentele acestuia, lucru ce poate să nu fie valabil dacă cineva a intervenit într-un anumit mod într-una din acțiunile precedente);
- Adăugarea de către emițător, în lista acestuia cu informații, a mesajului trimis;
- Adăugarea de către intrus, în lista acestuia cu informații, a mesajului trimis precum și a componentelor sale (până la nivelul la care are acces). Pasul acesta simulează observarea pasivă a fluxului de informații.

În timp ce ambele strategii intervin la momentul în care un agent primește un mesaj, diferența fundamentală dintre ele este aceea că, în cazul celei dintâi, atacatorul, chiar dacă este unul activ ce schimbă mesaje între instanțele de protocol, nu poate lua parte la vreuna din ele, dându-se drept unul dintre agenți, caz acoperit de cel de-al doilea test.

Metoda `runReceiveAction()`, prima strategie de atac:

- Se apelează metoda `replaceMessage()` din clasa `Intruder`, care, pentru un mesaj trimis de un agent catre alt agent, întoarce o listă de mesaje cu care atacatorul ar putea să înlocuiască mesajul inițial (algoritmul prin care se aleg mesajele în acest scop, va fi discutat mai tarziu);

- Pentru fiecare mesaj din lista primită la pasul anterior se creează două copii ale instanței de protocol, semnificând cazurile în care intrusul a intervenit și a schimbat mesajul în acțiunea *x*, iar de la acțiunea *x+1* va schimba, sau nu, mesajele în continuare. Agentul care primește mesajul modificat face o serie de verificări pentru a vedea daca poate să observe acest lucru. În caz afirmativ se oprește rularea acelei instanțe de protocol, deoarece atacatorul a fost detectat;

- Instanța curentă (aflată încă la acțiunea *x*, deoarece simulările precedente au rulat pe copii ale ei) va rula cu mesajul original, simulând situația în care intrusul alege să nu intervină la această acțiune. De fapt, destinatarul își adaugă mesajul și componentele sale (la care are acces) în lista cu informații și rulează o procedură de detecție a posibilelor schimbări făcute de intrus. Verificarea simulează situația în care, chiar dacă intrusul nu a schimbat nimic la această acțiune, poate a schimbat ceva la o acțiune precedentă, iar acum, deoarece nu a avut grijă să schimbe la loc unele parți ale mesajului, a fost dat de gol.

A doua strategie, ce se bazează pe abilitatea atacatorului de a se da drept un agent al unei instanțe de protocol, are nevoie de niște presetări. Apelul `run(2)` funcționează ca un wrapper, pornind secvențial un numar de instanțe de protocol egal cu numărul de agenți (în fiecare instanță atacatorul se dă drept un anumit agent). Desigur că, dacă se gasește un mod de atac, acesta este raportat, oprindu-se execuția.
Metoda `runReceiveAction()`, a doua strategie de atac:

- dacă agentul care a trimis mesajul este atacatorul:
  - dacă agentul care primește mesajul este comun celor două instanțe de protocol:
    - creez o copie a instanței actuale de protocol;
    - la instanța copie schimb mesajul inițial din această acțiune cu exact același mesaj, din instanța de protocol `masterProtocol` reținută de intrus;
    - rulez instanța copie din acest punct.
  - dacă agentul care primește mesajul nu este comun celor două instanțe de protocol:

        * creez trei copii ale instanței actuale de protocol ce vor coincide cazurilor (*adaug+schimb*, *adaug+nuSchimb*, *nuAdaug+schimb*; cazul *nuAdaug+nuSchimb* coincide cu instanța inițială, care rulează fără nicio intervenție a atacatorului);
        * la primele două copii adaug agentul care primește mesajul în lista agenților comuni cu instanța `masterProtocol`;
        * la prima și a treia instanță copie schimb mesajul inițial din această acțiune cu exact același mesaj, din instanța de protocol `masterProtocol` reținută de atacator;
        * rulez cele trei instanțe copii din acest punct.

- dacă agentul care a trimis mesajul nu este atacatorul (în acest caz nu se mai pot face schimbări precum cea anterioară):
    - dacă agentul care primește mesajul este comun celor două instanțe de protocol:
        * trebuie creată o copie a instanței curente și tratat cazul în care se face o schimbare normală, folosind cunoștințele pe care le are atacatorul la acest moment (ca în cazul `run(1)` când se utilizează apelul metodei `replaceMessage()` din clasa `Intruder`). Acest lucru va fi reluat alături de altele similare în *Capitolul 8*.
    - dacă agentul care primește mesajul nu este comun celor două instanțe de protocol:
        * creez trei copii ale instanței actuale de protocol, ce vor coincide cazurilor (*adaug+schimbareNormală*, *adaug+nuSchimb*, *nuAdaug+schimbareNormală*; cazul *nuAdaug+nuSchimb* coincide cu instanța inițială, care rulează fără nicio intervenție a atacatorului);
        * rulez cele trei instanțe copii din acest punct.

4.4.3 Aflarea termilor cu care poate fi înlocuit un mesaj

În acest capitol se va discuta algoritmul prin care atacatorul găsește, pornind de la un mesaj, protocolul căruia îi aparține, agenții (destinatarul și receptorul) și o listă cu mesajele ce pot fi înlocuite astfel încât să se compromită securitatea protocolului. Algoritmul este implementat în metoda `replaceMessage()` din clasa `Intruder`. Pentru o mai bună înțelegere a algoritmului, trebuie precedat de explicarea metodei de diferențiere între același mesaj aparținând a două instanțe de protocol diferite, dar care sunt create folosind același fișier de specificație.

Orice mesaj descris prin clasa concretă poate fi, de asemenea, privit și prin prisma clasei de baza pe care o derivă, `Term`. Aceasta agregă o referință din clasa `Protocol`, semnificând instanța de protocol în cadrul căreia a fost creat acel term. Orice clasă concretă are obligația de a oferi o implementare metodelor abstracte `equals()` și `theSame()` din clasa de baza, `Term`. Cele două metode primesc câte o instanță a clasei `Object` și returnează o valoare booleană, diferența fiind aceea că `theSame()` returnează *true* dacă apelantul și parametrul au aceeași formă ca și reprezentare textuală, pe cand `equals()` ține cont și de protocolul de securitate de care aparțin.

Metoda `replaceMessage()` din clasa `Intruder` funcționează ca un wrapper, apelând metoda privată `replaceTerm()` cu parametrii primiți. Metoda `replaceTerm()` este supraîncărcată pentru fiecare clasă concretă ce extinde clasa abstractă `Term`, deoarece atacatorul are diverse strategii pentru diverse tipuri de mesaje (tichete, nonce-uri, chei, nume de agenți etc.).

- pentru termii necriptați:
  - se caută exact același term, dar din instanța de protocol `masterProtocol`, deoarece, încercând să se adauge cât mai puțin posibil pe lângă ceea ce este obligatoriu vor scădea riscurile ca atacatorul sa fie detectat;
  - în cazul termilor ce se pot obține prin generare (tickete și nonce-uri), atacatorul adaugă și același mesaj generat de acesta, având atributul `intruderGenerated` setat cu valoarea *true*.
- pentru termii criptați:
  - când se știe modalitatea de criptare, se modifică conținutul și se recriptează;
  - când nu se știe modalitatea de criptare, se alege fiecare element criptat pe care îl avem.

În *Codul sursă 2* este prezentată metoda `replaceMessage()`. Având în vedere tipul mesajului primit, aceasta distribuie responsabilitatea metodei `replaceTerm()` corespunzătoare. Se trec în revistă toate semnăturile metodei `replaceTerm()` și se prezintă implementarea celei care găsește mesajele cu care se poate schimba un *nonce* primit.

```
1  public ArrayList<Term> replaceMessage(Protocol protocol, Term message, Agent to, Agent from){
2
3      ArrayList<Term> possibleChoices = new ArrayList<>();
4      if(message instanceof Agent){
5
6          possibleChoices.addAll(replaceTerm(protocol, (Agent) message));
7          return possibleChoices;
8      }
9      if(message instanceof Key){
10
11         possibleChoices.addAll(replaceTerm(protocol, (Key) message));
12         return possibleChoices;
13     }
14     if(message instanceof BasicTerm){
15
16         possibleChoices.addAll(replaceTerm(protocol, (BasicTerm) message));
17         return possibleChoices;
18     }
19     if(message instanceof Nonce){
20
21         possibleChoices.addAll(replaceTerm(protocol, (Nonce) message));
22         return possibleChoices;
23     }
24     if(message instanceof CompoundTerm){
25
26         possibleChoices.addAll(replaceTerm(protocol, (CompoundTerm) message, to, from));
27         return possibleChoices;
28
29     }
30     if(message instanceof EncryptedTerm) {
31
32         possibleChoices.addAll(replaceTerm(protocol, (EncryptedTerm) message, to, from));
33         return possibleChoices;
34     }
35     return possibleChoices;
36 }
37
38 private ArrayList<Term> replaceTerm(Protocol prot, Agent message);
39 private ArrayList<Term> replaceTerm(Protocol prot, Key message);
40 private ArrayList<Term> replaceTerm(Protocol prot, BasicTerm message);
41 private ArrayList<Term> replaceTerm(Protocol prot, Nonce message);
42 private ArrayList<Term> replaceTerm(Protocol prot, CompoundTerm message, Agent to, Agent from);
43 private ArrayList<Term> replaceTerm(Protocol prot, EncryptedTerm message, Agent to, Agent from);
44
45 private ArrayList<Term> replaceTerm(Protocol prot, Nonce message){
46
47     ArrayList<Term> result = new ArrayList<>();
48
49     Nonce generated = new Nonce(message.getAgent(), message.getNr(), message.getProtocol());
50     generated.setIntruderGenerated(true);
51     result.add(generated);
52
53     for(Term auxTerm : getMassiveKnowledge(prot)){
54
55         if(!message.equals(auxTerm) && message.theSame(auxTerm)){
56
57             result.add(auxTerm);
58             break;
59         }
60     }
61     return result;
62 }
```

Cod sursă 2: `replaceMessage()`

Interfaţa grafică

Interfaţa grafică din cadrul aplicaţiei *„Security Protocols Checker”* a fost realizată folosind tehnologia *JavaFx*[1]. Aceasta este formată atât din elemente grafice create la runtime, instanțiind anumite clase și apelând metodele corespunzătoare, cât și din ferestre statice realizate în design mode folosind utilitarul SceneBuilder[2] ce crează fișiere cu extensia `.fxml`. Toate acestea se găsesc în cadrul pachetului `SecurityProtocolsChecker.GUI`, a cărui componenţă este descrisă în *Figura 6*.

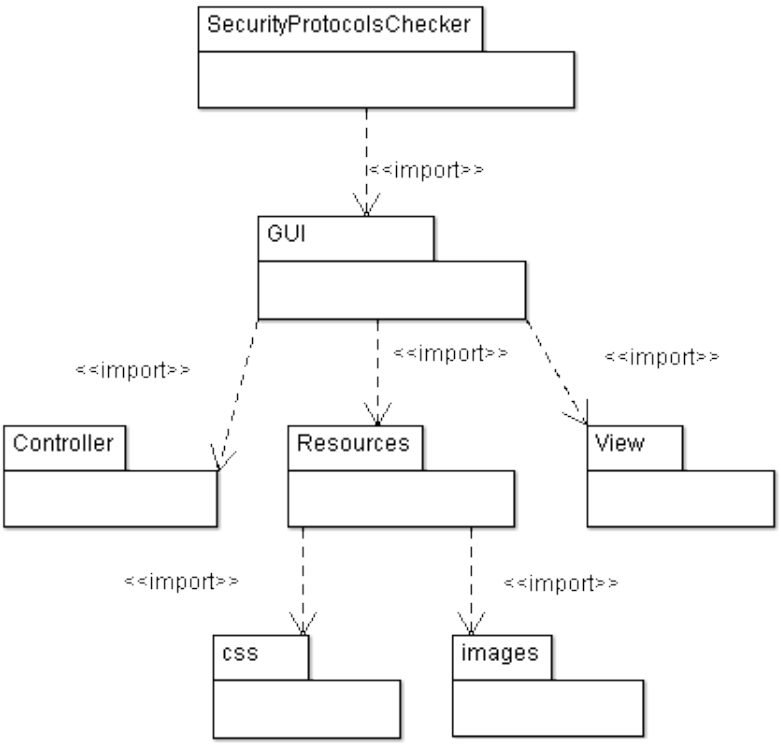

Figura 6: Diagrama pachetelor ce compun interfața grafică

În continuare, se vor prezenta componentele principale ale interfeței grafice, împreună cu clasele ce le descriu, precum și modulele răspunzătoare cu descrierea fișierelor de configurare și cu colorarea sintactică.

[1] *JavaFx*, distribuit de către *Oracle* în pachetul standard al *Java*, de la versiunea *1.8* - `http://docs.oracle.com/javafx/`

[2] SceneBuilder, utilitar oferit de Oracle sub licența Oracle BSD `http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html`

5.1 Editorul de specificații

Elementul principal al interfeței grafice, fiind, de fapt, unul din componentele de bază al oricărui *IDE* (mediu de dezvoltare), este editorul de specificații. Dintre funcționalitățile sale putem aminti: crearea şi deschiderea unui număr variabil de fișire de specificație, precum și modificarea, salvarea şi verificarea corectitudinii lor sintactice şi semantice.

La nivel atomic, editorul este format dintr-un număr variabil de taburi, ce se obțin din instanțe ale clasei `MyTab`, intrând în alcătuirea unui container (`TabPane`), definit alături de celelalte componente statice ale interfeței în fișierul `mainWindow.fxml`. Clasa `MyTab` extinde clasa `Tab` din pachetul `javafx.scene.control` și agregă două obiecte din clasele `MyCodeArea` și `TextArea`, semnificând cele două porțiuni unde se scrie codul sursă și respectiv unde se primește rezultatul verificărilor sintactice și semantice. Clasa `MyCodeArea` este un wrapper peste un obiect de tip `CodeArea`, din libraria open source *RichTextFX*[3].

Figura 7: Editorul de specificaţii

În *Figura 7* se pot observa cele două componente ale tab-ului, instanțe ale claselor `MyCodeArea` și `TextArea`, separate de un obiect din clasa `SplitPane`. În exemplul din figură este încărcată specificația unui protocol de securitate, care a fost analizat din punct de vedere sintactic și semantic. De asemenea, componenta de introducere a codului sursă are inclusă o bară de numerotare a liniilor, aceasta fiind de ajutor atunci când validarea sintactică eșuază, deoarece se indică la ce linie sunt probleme.

[3]RichTextFX, librarie open source distrbuită de Tomas Mikula `https://github.com/TomasMikula/RichTextFX`

5.2 Panoul de control

Un alt element important al interfeței grafice cu utilizatorul este *panoul de control*. Scopul acestuia este de a oferi o organizare a butoanelor ce permit interacționarea cu diferitele funcționalități ale aplicației.

Din punct de vedere structural, panoul de control este divizat în trei părți ce separă butoanele cu care utilizatorul poate interacționa după funcționalitățile pe care le oferă. Cele trei secțiuni, precum și elementele din interiorul acestora au fost aliniate folosind instanțe ale claselor `VBox` și `HBox` din pachetul `javafx.scene.layout`. Elementele componente ale secțiunilor panoului de control vor fi prezentate în cele ce urmează:

- butoane folosite pentru interacționarea cu sistemul de fișiere (fișier nou, deschidere fișier, salvare fișier);
- o imagine cu două stări (*„X”* roșu și bifă verde) ce se modifică dinamic, indicând dacă protocolul a fost sau nu încărcat cu succes pentru a fi rulat, precum și două butoane, pentru încărcarea și rularea protocolului;
- butoane folosite pentru a genera un `.pdf` și respectiv codul latex necesar acestuia, pornind de la specificația protocolului încărcat (cele din stânga) sau pornind de la rezultatul verificării protocolului (cele din dreapta).

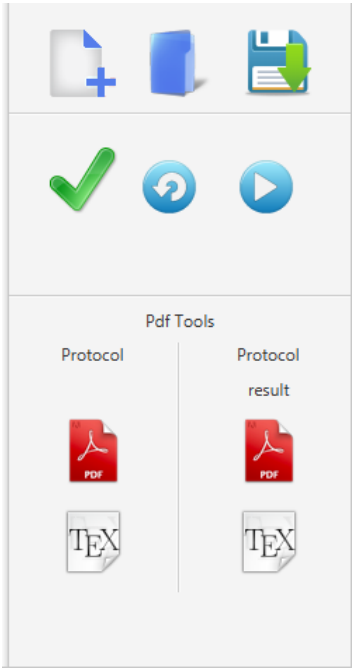

Figura 8: Panoul de control

5.3 Fişierele de configurare

În aplicația *„Security Protocols Checker”* au fost utilizate două tipuri de fișiere (`.properties`[4] și .css[5]) pentru a încărca date ce pot fi personalizate de utilizator. Ele sunt utilizate de clasa `ConfigurationAssistant` și descriu urmatoarele:

- *labels.properties* are o structură de tip cheie-valoare și reține numele meniurilor și etichetelor ce apar în cadrul aplicației;
- *colors.css* se folosește pentru a încărca culorile diverselor elemente din cadrul gramaticii ce descrie limbajul fișierelor de specificație.

[4]Extensia .properties - `https://docs.oracle.com/javase/tutorial/essential/environment/properties.html`
[5]Extensia `.css` - `http://www.w3schools.com/css/css_howto.asp`

5.4 Colorarea sintactică

Modulul de colorare sintactică a fost creat cu scopul de a îmbunătăți interacțiunea utilizatorului cu aplicația. Acesta poate înțelege acum mai ușor fișierul de specificație al unui protocol, unde termeni distincți(chei, agenți etc.) sunt colorați diferit.

Metoda prezentată *Codul sursă 3* este responsabilă cu colorarea sintactică a specificației. Dacă fișierul de configurare *colors.css* este prezent și este scris corect, atunci acesta se interpretează, culorile fiind citite de acolo. Altfel, este încărcată o schemă de culori implicită. Fișierul este foarte ușor de editat de către utilizator, care îl primește odată cu aplicația, într-un format implicit, iar tot ce trebuie să facă este să modifice culorile pentru anumite clase *css* predefinite în fișier (cu nume ușor de înțeles: `keys`, `agents` etc.). Culorile pot fi introduse în plain text (`red`, `green`, `blue` etc.), sau în cod hexadecimal (`#0000FF`). Metoda `computeHighlighting()` primește textul ce trebuie colorat sintactic și folosește clasele `Pattern` și `Matcher` pentru a îl împărți în token-uri (utilizând expresiile regulate pentru fiecare tip de element: cheie, agent etc.). Apoi le asignează câte o clasă din fișierul cu extensia `.css`. Apelul metodei este înregistrat la modificări asupra textului din `codeArea`, precum este prezentat în *Codul sursă 4*.

```
1     private static StyleSpans<Collection<String>> computeHighlighting(String text) {
2
3         Matcher matcher = PATTERN.matcher(text);
4         int lastKwEnd = 0;
5         StyleSpansBuilder<Collection<String>> spansBuilder
6                 = new StyleSpansBuilder<>();
7         while(matcher.find()) {
8             //        asignam o clasa din fisierul css in fct de exp regulata
9             String styleClass =
10                    matcher.group("KEYS") != null ? "keys" :
11                    matcher.group("BASICTERMS") != null ? "basicTerms" :
12                    matcher.group("NONCES") != null ? "nonces" :
13                    matcher.group("AGENTS") != null ? "agents" :
14                    matcher.group("BRACES") != null ? "braces" :
15                    matcher.group("PARAN") != null ? "paran" :
16                    null; /* never happens */ assert styleClass != null;
17            spansBuilder.add(Collections.emptyList(), matcher.start() - lastKwEnd);
18            spansBuilder.add(Collections.singleton(styleClass), matcher.end() - matcher.start());
19            lastKwEnd = matcher.end();
20        }
21        spansBuilder.add(Collections.emptyList(), text.length() - lastKwEnd);
22        return spansBuilder.create();
23    }
```

Cod sursă 3: Metoda de colorare `computeHighlighting()`

```
1     codeArea.textProperty().addListener((obs, oldText, newText) -> {
2
3             codeArea.setStyleSpans(0, computeHighlighting(newText));
4     });
```

Cod sursă 4: Inregistrarea metodei de colorare

Ghid de utilizare a aplicaţiei

Lucru cu fișiere

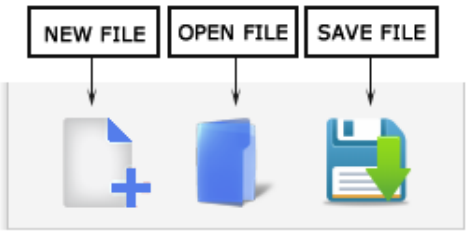

Figura 9: Butoanele editorului

Pentru a crea un fișier nou se va apăsa butonul *New*, ce va adăuga un tab nou și îl va activa. Acesta va avea numele predefinit *„Untitled”*. Al doilea buton din *Figura 9* este folosit cu scopul de a deschide fișiere de specificație din sistemul de fișiere. Salvarea fișierului de specificație se va realiza prin acționarea ultimului buton din *Figura 9*. În cazul în care fișierul nu a mai fost salvat, se va invoca dialogul de salvare al sistemului de operare. În caz contrar, conținutul fișierului va fi stocat la destinația reținută anterior.

Incărcarea și rularea protocolului

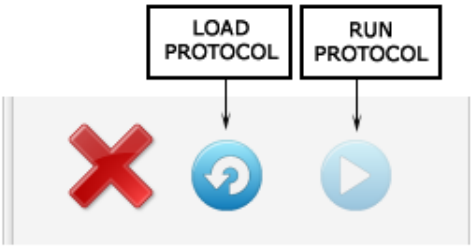

Figura 10: Incărcarea și rularea protocolului

Pentru a încărca un protocol în modulul de verificare al aplicației, este necesară apăsarea butonului de reîncărcare din *Figura 10*. În cazul în care specificația este validă, se va activa butonul de rulare al algoritmului de verificare și se va schimba imaginea, dintr-un ”X” roșu într-o bifă verde. La apăsarea ultimului buton, se va porni algoritmul de verificare a corectitudinii, lucru semnalat de apariția unui *progress circle* (cerc ce indică progresul task-ului curent). Algoritmul va rula într-un fir de execuție diferit de cel al interfeței grafice, utilizatorul având ocazia să interacționeze cu specificațiile deschise până se termină rularea, în cazul unui protocol complex. La finalul rulării protocolului, se va afișa rezultatul într-un panou.

Utilitarul de conversie a protocolului

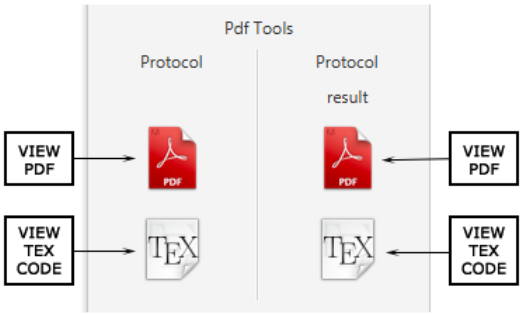

Figura 11: Utilitar de conversie

S-a creat un utilitar ce poate genera fișiere .pdf pornind de la protocol sau de la rezultatul verificării acestuia, transpunând textul din gramatica folosită de aplicație în gramatica de specialitate. Această funcționalitate se obține apăsând butonul roșu din prima sau din a doua coloană a *Figurii 11.* Este necesară instalarea în prealabil a utilitarului MikTex[1].

Același utilitar pentru lucrul cu fișiere .pdf prezentat mai sus are funcționalitatea de a oferi codul Latex necesar generării acestora, pornind tot de la protocolul încărcat, sau de la rezultatul verificării sale. Serviciul este util în cazul scrierii unei lucrări de specialitate.

Închiderea aplicației

Pentru a închide aplicația este necesară apăsarea butonului roșu din colțul din dreapta sus. La această acțiune este înregistrată o procedură ce verfică dacă sunt tab-uri deschise, iar pentru fiecare tab nesalvat, utilizatorului i se vor prezenta trei opțiuni: *Yes, No sau Cancel.* La apăsarea butonului *Cancel*, se va consuma evenimentul, iar aplicația nu se va închide.

[1]MikTex, utilitar de generare a fișierelor pdf - `http://miktex.org/`

Protocoale de securitate

În acest capitol vor fi trecute în revistă câteva exemple de protocoale, precum și modalitatea de atac găsită de utilitarul *„Security Protocols Checker”*. Se vor prezenta mai întâi protocoale simple pentru a identifica problemele la nivel atomic cu scopul unei mai bune înțelegeri. Apoi vom trece în revistă și protocoale mai complexe. Vom observa diverse strategii ale atacatorului, precum generarea de mesaje, schimbarea de mesaje între două instanțe diferite de protocolol etc.

7.1 Un protocol vulnerabil la integritate și confidențialitate

În *Protocolul 2* atacatorul generează un nonce și schimbă mesajul inițial, astfel că agentul B primește nonce-ul de la atacator în loc de cel corect.

$$A \; ! \; B \; : (\{N_{1_A}\}) \; N_{1_A}$$
$$B \; ? \; A : N_{1_A}$$

Protocol 2: Vulnerabil la confidențialitate și integritate

Rezultat 2: Modalitatea de atac asupra *Protocolului 2*

———Instance 1———

$A \; ! \; B \; : (\{N_{1_A}\}) \; N_{1_A}$

A generated $\{N_{1_A}\}$ for B

A sent $\{N_{1_A}\}$ to B

I received N_{1_A}

$B \; ? \; A : N_{1_I}$

Intruder generated some terms and recomputed the message with those

B received N_{1_I} from A

7.2 Un protocol vulnerabil la integritate

Protocolul 2 a fost modificat precum se arată în *Protocolul 3*, criptând comunicarea cu cheia publică a destinatarului. Scopul a fost evidențierea faptului că, deși atacatorul nu mai are acces la nonce-ul trimis de A, el totuși poate să pună probleme protocolului, oferind lui B o informație falsă.

$$A \; ! \; B \; : (\{N_{1_A}\}) \; \{N_{1_A}\}K^e_B$$
$$B \; ? \; A : \{N_{1_A}\}K^e_B$$

Protocol 3: Vulnerabil integritate

Rezultat 3: Modalitatea de atac asupra *Protocolului 3*

———Instance 1———

$A \; ! \; B \; : (\{N_{1_A}\}) \; \{N_{1_A}\}K^e_B$

A generated $\{N_{1_A}\}$ for B

A sent $\{\{N_{1_A}\}K^e_B\}$ to B

I received $\{N_{1_A}\}K^e_B$

$B \; ? \; A : \{N_{1_I}\}K^e_B$

Intruder generated some terms and recomputed the message with those

B received $\{N_{1_I}\}K^e_B$ from A

B received N_{1_I} from A

7.3 Un protocol vulnerabil la confidențialitate

S-a modificat *Protocolul 3*, adăugând o acțiune de confirmare a mesajului primit de la B către A. Totuși, se pare că doar o simplă confirmare nu este suficientă, atacatorul reușind să obțină o informație nedestinată acestuia.

$$A \; ! \; B \; : (\{N_{1_A}\}) \; \{N_{1_A}\}K^e_B$$
$$B \; ? \; A : \{N_{1_A}\}K^e_B$$
$$B \; ! \; A \; : \{N_{1_A}\}K^e_A$$
$$A \; ? \; B : \{N_{1_A}\}K^e_A$$

Protocol 4: Vulnerabil confidențialitate

Rezultat 4: Modalitatea de atac asupra *Protocolului 4*

Agents commons to both protocol instances: B

Instance 1	Instance 2
$A \; ! \; B \; : (\{N_{1_A}\}) \; \{N_{1_A}\}K^e_B$	$I!B{:}(\{N_{1_I}\})\{N_{1_A}\}K^e_B$
A generated $\{N_{1_A}\}$ for B	I sends the same message that was sended in the same action of the previous protocol instance
A sent $\{N_{1_A}\}K^e_B$ to B	
I received $\{N_{1_A}\}K^e_B$	All this because later will discover $[N_{1_A}]$ from the other protocol instance
$B \; ? \; A : \{N_{1_A}\}K^e_B$	$B?I{:}\{N_{1_A}\}K^e_B$
B received $\{N_{1_A}\}K^e_B$ from A	B received $\{N_{1_A}\}K^e_B$ from I
B received N_{1_A} from A	B received N_{1_A} from I
$B \; ! \; A \; : \{N_{1_A}\}K^e_A$	$B!I{:}\{N_{1_A}\}K^e_I$
B sent $\{N_{1_A}\}K^e_A$ to A	B sent $\{N_{1_A}\}K^e_I$ to I
I received $\{N_{1_A}\}K^e_A$	I received $\{N_{1_A}\}K^e_I$
$A \; ? \; B : \{N_{1_A}\}K^e_A$	$I?B{:}\{N_{1_A}\}K^e_I$
A received $\{N_{1_A}\}K^e_A$ from B	I received $\{N_{1_A}\}K^e_I$ from B
A received N_{1_A} from B	I received N_{1_A} from B

7.4 Un protocol sigur

Protocolul 4 a fost îmbunătățit, adăugând în prima acțiune identitatea agentului A, alături de nonce. În protocolul anterior s-a observat că atacatorul copiază mesajul trimis în acțiunea 1 și profită de faptul că se retransmite exact același lucru drept confirmare în acțiunea 3. Prin adăugarea unei informații suplimentare, atacatorul nu poate să privească înauntrul mesajului și să rețina doar nonce-ul, prin urmare nu poate să recompună mesajul necesar acțiunii de confirmare, după cum o arată *Protocolul 5*.

$$
\begin{aligned}
&A\ !\ B\ \ : (\{N_{1_A}\})\ \{A,N_{1_A}\}K^e_B\\
&B\ ?\ A : \{A,N_{1_A}\}K^e_B\\
&B\ !\ A\ \ : \{N_{1_A}\}K^e_A\\
&A\ ?\ B : \{N_{1_A}\}K^e_A
\end{aligned}
$$

Protocol 5: Sigur

Rezultat 5: Rularea *Protocolului 5*

———Instance 1———

$A\ !\ B\ \ : (\{N_{1_A}\})\ \{A,N_{1_A}\}K^e_B$
A generated $\{N_{1_A}\}$ for B
A sent $\{A,N_{1_A}\}K^e_B$ to B
I received $\{A,N_{1_A}\}K^e_B$

$B\ ?\ A : \{A,N_{1_A}\}K^e_B$
B received $\{A,N_{1_A}\}K^e_B$ from A
B received A from A
B received N_{1_A} from A

$B\ !\ A\ \ : \{N_{1_A}\}K^e_A$
B sent $\{N_{1_A}\}K^e_A$ to A
I received $\{N_{1_A}\}K^e_A$

$A\ ?\ B : \{N_{1_A}\}K^e_A$
A received $\{N_{1_A}\}K^e_A$ from B
A received N_{1_A} from B

7.5 Protocolul Needham-Schroeder, vulnerabil la confidențialitate

Agentul J a greșit în prima acțiune pentru că nu și-a trimis identitatea sub formă criptată. Astfel, atunci când atacatorul va copia mesajul ce se trimite în acțiunea numărul 1 a primei instanțe, el îl va prelucra înainte de a îl trimite mai departe, schimbând identitatea lui J cu propria identitate.

$$
\begin{aligned}
&J \mathbin{!} R \ : (\{N_{1_J}\})\ J,R,\{R,N_{1_J}\}K^e_R \\
&R \mathbin{?} J : J,R,\{R,N_{1_J}\}K^e_R \\
&R \mathbin{!} J \ : (\{N_{1_R}\})\ R,J,\{N_{1_J},N_{1_R},R\}K^e_J \\
&J \mathbin{?} R : R,J,\{N_{1_J},N_{1_R},R\}K^e_J \\
&J \mathbin{!} R \ : J,R,\{N_{1_R}\}K^e_R \\
&R \mathbin{?} J : J,R,\{N_{1_R}\}K^e_R
\end{aligned}
$$

Protocol 6: Needham-Schroeder

Rezultat 6: Modalitatea de atac asupra *Protocolului 6*

Agents commons to both protocol instances: R

———Instance 1———	———Instance 2———
$J \mathbin{!} R \ : (\{N_{1_J}\})\ J,R,\{R,N_{1_J}\}K^e_R$	$I!R{:}(\{N_{1_I}\})\{I,R,\{R,N_{1_J}\}K^e_R\}$
J generated $\{N_{1_J}\}$ for R	I sends the same message that was sended in the same action of the previous protocol instance
J sent $\{J,R,\{R,N_{1_J}\}K^e_R\}$ to R	
I received J	
I received R	All this because later will discover $[N_{1_J}]$ from the other protocol instance
I received $\{R,N_{1_J}\}K^e_R$	
$R \mathbin{?} J : J,R,\{R,N_{1_J}\}K^e_R$	$R?I{:}\{I,R,\{R,N_{1_J}\}K^e_R\}$
R received J from J	R received I from I
R received R from J	R received R from I
R received $\{R,N_{1_J}\}K^e_R$ from J	R received $\{R,N_{1_J}\}K^e_R$ from I
R received R from J	R received R from I
R received N_{1_J} from J	R received N_{1_J} from I
$R \mathbin{!} J \ : (\{N_{1_R}\})\ R,J,\{N_{1_J},N_{1_R},R\}K^e_J$	$R!I{:}(\{N_{1_R}\})\{R,I,\{N_{1_J},N_{1_R},R\}K^e_I\}$
R generated $\{N_{1_R}\}$ for J	R generated $\{N_{1_R}\}$ for I

R sent $\{R,J,\{N_{1_J},N_{1_R},R\}K^e_J\}$ to J

I received R

I received J

I received $\{N_{1_J},N_{1_R},R\}K^e_J$

J ? R : $R,J,\{N_{1_J},N_{1_R},R\}K^e_J$

J received R from R

J received J from R

J received $\{N_{1_J},N_{1_R},R\}K^e_J$ from R

J received N_{1_J} from R

J received N_{1_R} from R

J received R from R

J ! R : $J,R,\{N_{1_R}\}K^e_R$

J sent $\{J,R,\{N_{1_R}\}K^e_R\}$ to R

I received J

I received R

I received $\{N_{1_R}\}K^e_R$

R ? J : $J,R,\{N_{1_R}\}K^e_R$

R received J from J

R received R from J

R received $\{N_{1_R}\}K^e_R$ from J

R received N_{1_R} from J

R sent $\{R,I,\{N_{1_J},N_{1_R},R\}K^e_I\}$ to I

I received R

I received I

I received $\{N_{1_J},N_{1_R},R\}K^e_I$

$I?R$:$\{R,I,\{N_{1_J},N_{1_R},R\}K^e_I\}$

I received R from R

I received I from R

I received $\{N_{1_J},N_{1_R},R\}K^e_I$ from R

I received N_{1_J} from R

I received N_{1_R} from R

I received R from R

$I!R$:$\{I,R,\{N_{1_R}\}K^e_R\}$

$R?I$:$\{I,R,\{N_{1_R}\}K^e_R\}$

7.6 Protocolul Needham-Schroeder-Lowe, sigur

Aceasta este versiunea modificată a *Protocolului 6* ce a fost corectată de catre Lowe pentru a rezolva breșa de securitate. Lowe a modificat prima acțiune a protocolului, agentul J trimițând acum propria identitate criptată, și nu pe cea a lui R.

J ! R : $(\{N_{1_J}\})$ J,R,$\{J,N_{1_J}\}K^e_R$
R ? J : J,R,$\{J,N_{1_J}\}K^e_R$
R ! J : $(\{N_{1_R}\})$ R,J,$\{N_{1_J},N_{1_R},R\}K^e_J$
J ? R : R,J,$\{N_{1_J},N_{1_R},R\}K^e_J$
J ! R : J,R,$\{N_{1_R}\}K^e_R$
R ? J : J,R,$\{N_{1_R}\}K^e_R$

Protocol 7: Needham-Schroeder-Lowe

Rezultat 7: Rularea *Protocolului 7*

———Instance 1———
J ! R : $(\{N_{1_J}\})$ J,R,$\{J,N_{1_J}\}K^e_R$
J generated $\{N_{1_J}\}$ for R
J sent $\{J,R,\{J,N_{1_J}\}K^e_R\}$ to R
I received J
I received R
I received $\{J,N_{1_J}\}K^e_R$

R ? J : J,R,$\{J,N_{1_J}\}K^e_R$
R received J from J
R received R from J
R received $\{J,N_{1_J}\}K^e_R$ from J
R received J from J
R received N_{1_J} from J

R ! J : $(\{N_{1_R}\})$ R,J,$\{N_{1_J},N_{1_R},R\}K^e_J$
R generated $\{N_{1_R}\}$ for J
R sent $\{R,J,\{N_{1_J},N_{1_R},R\}K^e_J\}$ to J
I received R
I received J
I received $\{N_{1_J},N_{1_R},R\}K^e_J$

J ? R : R,J,$\{N_{1_J},N_{1_R},R\}K^e_J$

J received R from R

J received J from R

J received $\{N_{1_J},N_{1_R},R\}K^e_J$ from R

J received N_{1_J} from R

J received N_{1_R} from R

J received R from R

J ! R : J,R,$\{N_{1_R}\}K^e_R$

J sent $\{J,R,\{N_{1_R}\}K^e_R\}$ to R

I received J

I received R

I received $\{N_{1_R}\}K^e_R$

R ? J : J,R,$\{N_{1_R}\}K^e_R$

R received J from J

R received R from J

R received $\{N_{1_R}\}K^e_R$ from J

R received N_{1_R} from J

7.7 Protocolul BAN modified Andrew Secure RPC, sigur

Protocolul 8 a fost modelat după specificația de la `http://www.lsv.ens-cachan.fr/spore/andrewBAN.html`. Conform SPORE[1] nu există atacuri cunoscute, ceea ce confirmă și utilitarul *„Security Protocols Checker”*.

$$
\begin{aligned}
&J\ !\ R\ : (\{N_{1_J}\})\ J,R,J,\{N_{1_J}\}K_{JR}\\
&R\ ?\ J\ : J,R,J,\{N_{1_J}\}K_{JR}\\
&R\ !\ J\ : (\{N_{1_R}\})\ R,J,\{N_{1_J},N_{1_R}\}K_{JR}\\
&J\ ?\ R\ : R,J,\{N_{1_J},N_{1_R}\}K_{JR}\\
&J\ !\ R\ : J,R,\{N_{1_R}\}K_{JR}\\
&R\ ?\ J\ : J,R,\{N_{1_R}\}K_{JR}\\
&R\ !\ J\ : (\{N_{2_R},K_I\})\ R,J,\{K_I,N_{2_R},N_{1_J}\}K_{JR}\\
&J\ ?\ R\ : R,J,\{N_{3_R},N_{2_R},N_{1_J}\}K_{JR}
\end{aligned}
$$

Protocol 8: BAN modified Andrew Secure RPC

Rezultat 8: Rularea *Protocolului 8*

———Instance 1———

$J\ !\ R\ : (\{N_{1_J}\})\ J,R,J,\{N_{1_J}\}K_{JR}$

J generated $\{N_{1_J}\}$ for R

J sent $\{J,R,J,\{N_{1_J}\}K_{JR}\}$ to R

I received J

I received R

I received J

I received $\{N_{1_J}\}K_{JR}$

$R\ ?\ J\ : J,R,J,\{N_{1_J}\}K_{JR}$

R received J from J

R received R from J

R received J from J

R received $\{N_{1_J}\}K_{JR}$ from J

R received N_{1_J} from J

$R\ !\ J\ : (\{N_{1_R}\})\ R,J,\{N_{1_J},N_{1_R}\}K_{JR}$

R generated $\{N_{1_R}\}$ for J

R sent $\{R,J,\{N_{1_J},N_{1_R}\}K_{JR}\}$ to J

[1]SPORE - Security Protocols Open Repository `http://www.lsv.ens-cachan.fr/Software/spore/`

I received R

I received J

I received $\{N_{1_J},N_{1_R}\}K_{JR}$

J ? R : $R,J,\{N_{1_J},N_{1_R}\}K_{JR}$

J received R from R

J received J from R

J received $\{N_{1_J},N_{1_R}\}K_{JR}$ from R

J received N_{1_J} from R

J received N_{1_R} from R

J ! R : $J,R,\{N_{1_R}\}K_{JR}$

J sent $\{J,R,\{N_{1_R}\}K_{JR}\}$ to R

I received J

I received R

I received $\{N_{1_R}\}K_{JR}$

R ? J : $J,R,\{N_{1_R}\}K_{JR}$

R received J from J

R received R from J

R received $\{N_{1_R}\}K_{JR}$ from J

R received N_{1_R} from J

R ! J : $(\{N_{2_R},K_1\})$ $R,J,\{K_1,N_{1_R},N_{1_J}\}K_{JR}$

R generated $\{N_{2_R},K_1\}$ for J

R sent $\{R,J,\{K_1,N_{1_R},N_{1_J}\}K_{JR}\}$ to J

I received R

I received J

I received $\{K_1,N_{1_R},N_{1_J}\}K_{JR}$

J ? R : $R,J,\{K_1,N_{1_R},N_{1_J}\}K_{JR}$

J received R from R

J received J from R

J received $\{K_1,N_{1_R},N_{1_J}\}K_{JR}$ from R

J received K_1 from R

J received N_{1_R} from R

J received N_{1_J} from R

Direcţii de dezvoltare viitoare

Modelul teoretic se poate extinde prin adăugarea unui modul ce verifică proprietatea de anonimitate. Acest lucru este posibil prin transpunerea sistemului de fapte prezentat în [1]. Se vor putea astfel decide diverse clase de anonimitate precum: anonimitatea minimală , anonimitatea de grup etc.

Având în vedere implementarea, am putea aplica un algoritm de decizie inteligent bazat pe reprezentarea arborescentă a mesajelor atunci când e nevoie să se schimbe un mesaj criptat despre care nu se știe modalitatea de criptare. Scopul este scădea numărului de rezultate întoarse de către metoda `replaceMessage()`, având în vedere faptul că, pentru fiecare rezultat, se creează o copie a instanței de protocol apelante, se schimbă acel mesaj și se rulează.

Modalitatea actuală de copiere a unei instanțe de protocol este aceea de serializare și deserializare, operație care este costisitoare. O rezolvare ar fi implementarea interfeței `Clonable` și suprascrierea metodei `Clone()`.

Un avantaj major în ceea ce privește performanța ca viteză de execuție a sistemului ar putea-o aduce paralelizarea task-urilor prin rularea instanțelor de protocol diferite în fire de execuție diferite. Va fi nevoie de implementarea unui sistem de comunicare între instanța părinte și protocoalele fii ce rulează în paralel, având în vedere problemele clasice ale programării concurente, precum interblocajul.

Cât despre interacțiunea cu utilizatorul, putem îmbunătăți aplicația prin schimbarea modului de reprezentare a rezultatului, din cel textual într-unul grafic, precum cel din cadrul utilitarului Scyther[1]. De asemenea, editorul de text ar putea fi îmbunătățit prin adăugarea unui modul de completare automată a specificației.

[1]Scyther, utilitar de verificare automata a protocoalelor de securitate - `http://www.cs.ox.ac.uk/people/cas.cremers/scyther/`

Concluzii

Am creat un utilitar bazat pe modelul teoretic trecut în revistă în *Capitolul 2*. Scopul său este de a verifica *integritatea* și *confidențialitatea* unui protocol de securitate care rulează într-un mediu cu atacator activ. Protocoalele sunt definite folosind fișiere de specificație, ce vor fi încărcate în cadrul aplicației. Rularea acestora este precedată de trecerea prin module de verificare sintactică și semantică, pentru a se garanta respectarea modelului teoretic folosit. Pentru validarea sintactică, am analizat structura specificaţiei, am observat regulile după care este construită și am decis să creez o gramatică pentru a o descrie şi să generez un analizator de sintaxă pornind de la aceasta. Definirea specificaţiei alături de validarea sintactică şi semantică sunt descrise în detaliu în *Capitolul 4.2*.

Principalul modul al aplicației conține implementarea algoritmului de *model checking* folosit pentru verificarea proprietăților de securitate. La nivel atomic, algoritmul este construit având în vedere posibilele intervenții ale atacatorului: generarea de mesaje, schimbarea mesajelor între instanțele de protocol, participarea la o instanță în locul unui agent etc. Acestea au fost iterate pe parcursul acțiunilor protocolului, luând în considerare și o serie de verificări. Unele protocoale reușesc să observe intervențiile atacatorului, în timp ce altele sunt vulnerabile. Algoritmul este surprins în detaliu în *Capitolul 4.4*.

Aplicația a fost construită în stilul unui *IDE*, mediu de dezvoltare, fiind constituită dintr-un editor de text și un panou de control. Editorul poate lucra cu mai multe fișiere în același timp, oferind de asemenea și funcționalitatea de colorare sintactică a specificației. Prin intermediul panoului de control se poate rula algoritmul de verificare a instanței de protocol încărcate.

În opinia mea, prin utilitarul *„Security Protocols Checker”* am reușit să realizez ceea ce mi-am propus pentru moment. Aplicația poate fi îmbunătățită prin adăugarea unui modul ce verifică proprietatea de anonimitate. În acest scop, modelul teoretic poate fi extins prin transpunerea sistemului de fapte prezentat în [1]. De asemenea, prin modificarea metodei `replaceMessage()` din clasa `Intruder`, se pot adăuga și alte strategii atacatorului, pentru a detecta o clasă mai largă de probleme în protocoalele de securitate. Acestea, precum și alte direcții de dezvoltare viitoare sunt prezentate în *Capitolul 8*.

Bibliografie

[1] F. L. Ţiplea, L. Vamanu, and C. Vârlan. Reasoning about minimal anonymity in security protocols, future generation computer systems. February 2012.

[2] V. Cosmin. *Anonymity in security Protocols*. Ph.D. dissertation, Faculty of Computer Science, Iași, April 2013.

[3] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.

[4] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.

[5] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, October 1985.

[6] T. Harreveld. *Dining Cryptographer Networks*. June, 2012.

[7] R. Ramanujam and S. P. Suresh. A decidable subclass of unbounded security protocols. 2003.

[8] Martin Fowler. *GUI Architectures*. 18 July 2006.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2005.