ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Information Security Group
Prof. Dr. David Basin

Simulation von Sicherheitsprotokollen

Semesterarbeit

September 2003 – Dezember 2003

Tommi Lindgren und Thomas Tscherrig
{tolindgr,tschetho}@student.ethz.ch

Betreuung: Michael Näf, Paul Hankes Drielsma, Paul E. Sevinç

Inhalt

Zusammenfassung

In dieser Semesterarbeit wurde ein Sicherheitsprotokoll-Simulator konzeptionell ausgearbeitet und implementiert. Er vermag generische Protokolle einzulesen und diese zu simulieren, indem mittels Textausgabe der aktuelle Stand des Protokolls visualisiert wird.

Besonderes Augenmerk wurde auf ein ordentliches Softwaredesign gelegt, sodass spätere Erweiterungen ohne grosse Umstrukturierungen stattfinden können.

Ein besonderes Feature ist die Verteiltheit: Es spielt keine Rolle, ob die Simulation auf ein und derselben Maschine oder verteilt über mehrere Computer im lokalen Netzwerk stattfindet.

Abstract

In this semester thesis a security protocol simulator has been designed and implemented. The simulator reads generic protocols and simulates them by visualising the current state of the protocol through textual output.

Special attention has been paid to a well thought out software design, which supports further development.

A noteworthy feature of the simulator is its ability to execute in a distributed environment: it does not matter whether or not the simulation runs on one or several machines in the same local network simultaneously.

1 Aufgabenstellung & Motivation

Im Themengebiet Informationssicherheit gibt es viele Sicherheitsprotokolle, die den Austausch von Information unter Erhaltung gewisser Eigenschaften (Authentizität, Geheimhaltung, etc.) ermöglichen. Beispiele solcher Protokolle sind Diffie-Hellman, RSA oder SSL. Diese Protokolle sind z.T. nicht leicht verständlich und erfordern im Lehrbereich einigen Aufwand, um sie aufzubereiten.

Zum vereinfachten Studium von Sicherheitsprotokollen soll ein geeigneter Protokollsimulator entworfen und implementiert werden. Er soll ein beliebiges Protokoll als Input einlesen und dieses auf eine geeignete Weise darstellen. Die Lernenden sollen das Protokoll im Detail studieren und mit relevanten Parametern experimentieren können.

In einem weiteren Schritt kann die Simulation von Angriffen einbezogen werden, indem der Simulator auf den bestehenden AVISPA-Tools [1] aus der Information Security Group aufbaut und den Angriff visualisiert. Die Realisierung dieses Projektteils ist abhängig vom vorgängigen Projektverlauf und wird während des Projekts entschieden[1].

2 Der Simulator im Überblick

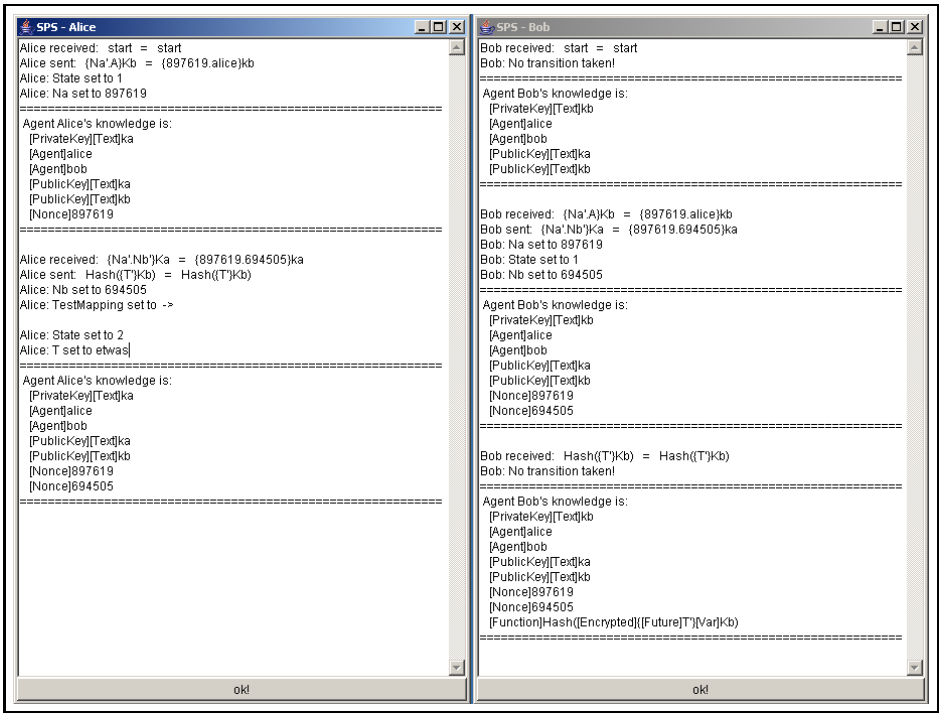

Abbildung 1: Screenshot des Simulators

[1] In Folge verstärkter Fokussierung auf andere Bereiche wurde dieser Teil nicht realisiert.

2.1 Einführung

In diesem Kapitel wird das fertige Ergebnis unserer Semesterarbeit – der Simulator – vorgestellt. Der Einfachheit halber reden wir von Alice und Bob als Protokollrollen, auch wenn diese beliebige andere Namen tragen könnten.

Der Simulator arbeitet mit der High Level Protocol Specification Language (im folgenden HLPSL genannt), in welcher die zu simulierenden Protokolle verfasst sein müssen. HLPSL [2] entstand im Zusammenhang mit dem AVISPA-Projekt [1], das die automatisierte Validation von Sicherheitsprotokollen verfolgt.

Nach dem Start liest der Simulator das als Argument mitgegebene Protokoll und öffnet zwei Outputfenster – eines für Alice, eines für Bob. Das Protokoll wird im Folgenden simuliert, d.h. abgearbeitet. Wie in HLPSL vorgesehen, schickt der Simulator zu Beginn allen beteiligten Rollen eine "start"-Nachricht. Darauf reagiert genau eine Rolle: Sie beginnt das Protokoll mit der ersten zu verschickenden Nachricht. Schickt z.B. Alice eine Nachricht, so wird eine entsprechende Mitteilung in ihrem Fenster angezeigt. In Bobs Fenster erscheint die angekommene Nachricht sowie die Mitteilung, ob damit eine HLPSL-Transition durchgeführt werden konnte. Falls ja, werden alle Variablenzuweisungen sowie sein aktuelles Wissen ausgegeben.

Dieser Ablauf wird wiederholt, bis keine Nachrichten mehr verschickt werden. Das ist dann der Fall, wenn eine ankommende Nachricht keinen Zustandsübergang mehr auslöst. Anhand der State-Variable (die Zustandsvariable der beiden endlichen Automaten Alice und Bob) kann nun abgelesen werden, ob die beiden in akzeptierende Zustände gelangt sind.

2.2 Beispiel

Als Beispiel führen wir das Needham-Schröder Public Key Authentifikationsprotokoll [3] auf. (Siehe Anhang)

In der Kommandozeile wird je Alice und Bob einzeln gestartet:

```
java -jar SPSimulator.jar -file=NSPK -name=alice -m -group=sps0
java -jar SPSimulator.jar -name=bob -s -group=sps0
```

Die genaue Bedeutung der Befehle wird im Abschnitt 6 erläutert.

Nachdem Alice die "start"-Nachricht vom Simulator gekriegt hat, quittiert der User dies mit einem Klick auf den "ok!"-Knopf. Dann führt Alice ihren ersten Zustandsübergang durch, gibt ihren aktuellen Wissensstand samt Variablenzuweisungen aus und schickt, wie vom Protokoll verlangt, die erste Nachricht (`{Na'.A}Kb`) an Bob. Die Nonce `Na'` wurde im Beispiel auf `897619` gesetzt.

Bob erhält die "start"-Nachricht ebenfalls, kann aber nichts damit anfangen und gibt "No transition taken!" aus. Sodann erreicht ihn die von Alice verfasste Nachricht, welche in seinem Fenster als ankommende Nachricht erscheint. Laut Protokoll erwartet Bob eine Nachricht der Form `{Na.A}Kb`: er führt eine Unifikation mit der ankommenden Nachricht durch und entscheidet, dass die beiden zusammenpassen (siehe auch Abschnitt 4.2).

Nun führt Bob einen Zustandsübergang durch: Sein `State` wird auf `1` gesetzt. `Nb'` nimmt den Wert `694505` an und wird, wie auch die empfangene Nonce `Na'`, zu Bobs Wissen hinzugefügt. Nun schickt er die Nachricht `{Na.Nb'}Ka` an Alice und gibt seinen Wissensstand aus.

Alice führt analog eine Unifikation des erwarteten Nachrichtenformats mit dem tatsächlich empfangenen durch, setzt die Variable `Nb'` auf `694505` und führt erneut einen Zustandsübergang durch: `State'=2`. Zum Schluss sendet sie die Bestätigungsnachricht `{Nb'}Kb` und gibt ihren Wissensstand aus.

Bob empfängt diese Nachricht und überprüft, ob sein `Nb` mit dem empfangenen übereinstimmt, wonach er seinen Zustand `State` auf `2` setzt, seinen Wissensstand ausgibt und nichts mehr sendet. Somit ist das Protokoll abgeschlossen.

3 Architektur

3.1 Statische Architektur

Hervorzuheben an der statischen Architektur ist das Schichtenmodell. Mit ihm wurde versucht, die jetztigen und auch die künftigen Anforderungen an den Simulator zu erfüllen. Die Schichten sind (von der obersten zur untersten): Role – Player – Simulator – Communication.

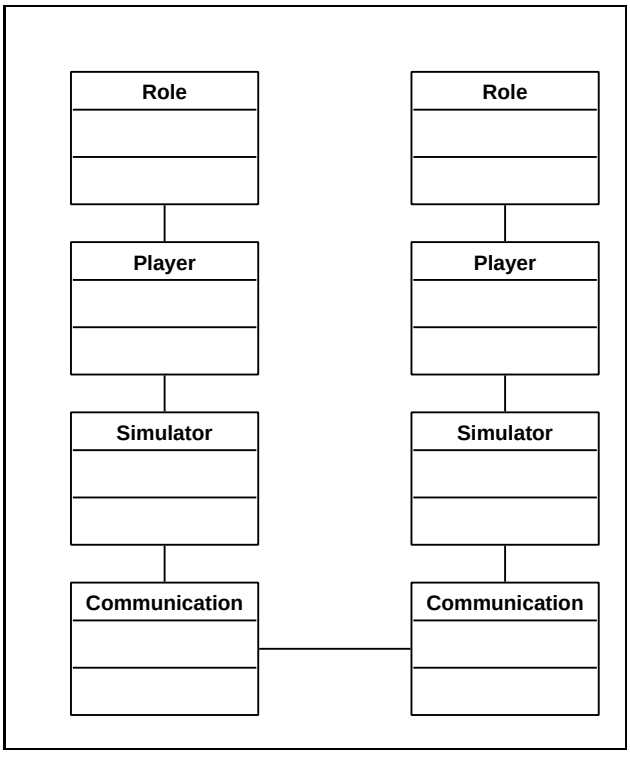

Abbildung 2: Eine (instanzierte) Visualisierung des Schichtenmodells

Die oberste Schicht bildet die Role-Klasse. Sie verschickt die durch das Protokoll spezifizierte, zu sendende Nachricht an die andere Rolle des Protokolls (unser Simulator beschränkt sich auf zwei Rollen, siehe Abschnitt 3.3). Die Rolle "weiss" aber nicht, *wo* sich die andere Rolle befindet, d.h. weder von welchem

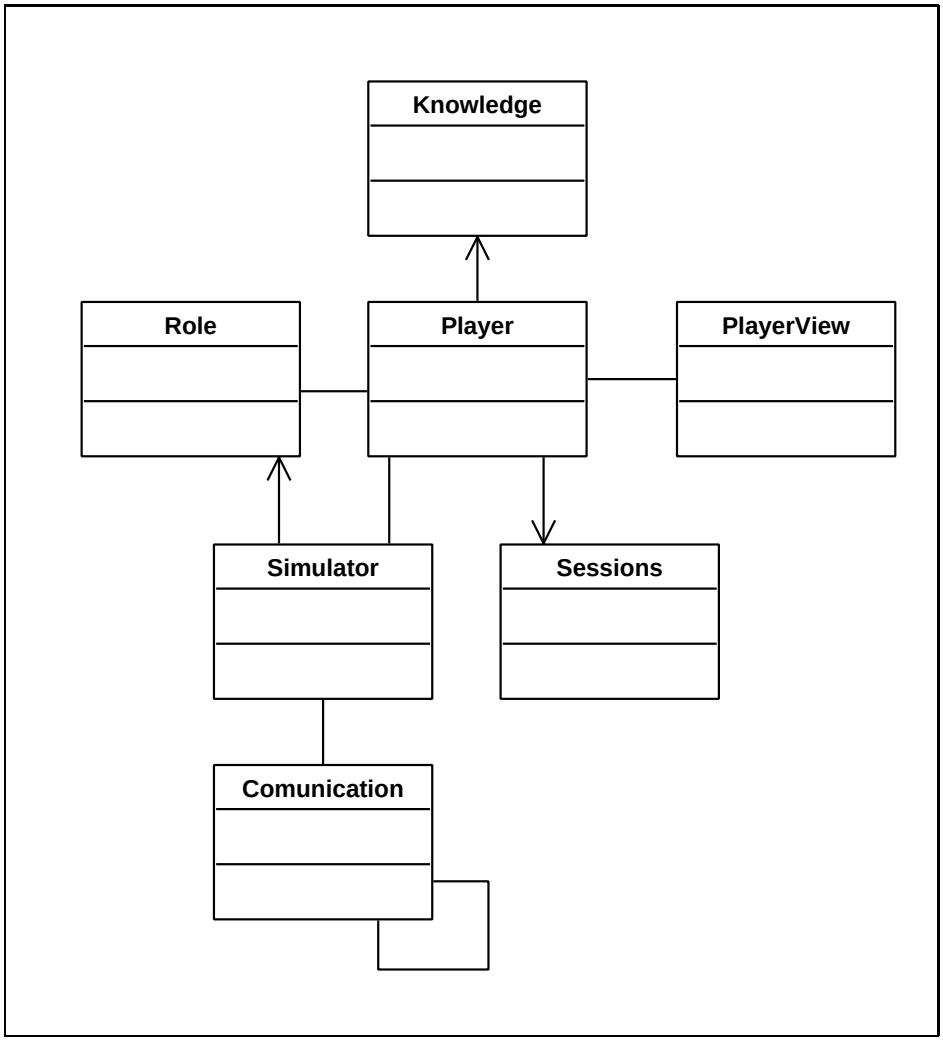

Abbildung 3: Die statische Architektur in UML

Player [2] sie gespielt, noch von welchem Simulator dieser Player simuliert wird. Sie reicht deshalb die zu versendende Nachricht an die unterliegende Schicht weiter – an den Player, von dem sie gespielt wird – in der Form (AbsenderRolle, (Nachricht)).

Der Player kennt die einzelnen Sessionen, in die er involiert ist (siehe Sessions, Abb. 3). An Hand dieser Sessionen kann er die Partnerrolle ausfindig machen und von dieser den Player, von welchem sie gespielt wird. Der Player verpackt die Mitteilung, die der empfangende Player von seiner Rolle erhalten hat, folgendermassen: (AbsenderAgent, EmpfängerAgent, EmpfängerRolle, (Nachricht)) und schickt sie dem Simulatoren, von dem er simuliert wird. Die Absenderrolle wird absichtlich nicht mitgeschickt, da diese Information redundant wäre: Der empfangende Player kann die Partnerrolle wiederum an Hand seiner Sessioneninformation ausfindig machen.

Der Simulator hat die Aufgabe, jede Nachricht zuerst an den Intruder zu schicken, sofern dieser existiert (muss nicht zwingend instanziert sein). Der Intruder ist ein Agent, der die Intruderrolle spielt. Der Simulator verpackt demnach die Nachricht weiter: (Intruder, (AbsenderAgent, EmpfängerAgent, EmpfängerRolle, (Nachricht))) und leitet sie an die Kommunikations-Klasse weiter.

Die Communication-Klasse ist die unterste Schicht. Sie ist einzig für den korrekten Versand der Nachrichten verantwortlich und kennt demnach die (Netzwerk-) Adressen der einzelnen Player. Sie schickt die Nachricht an den spezifizierten Empfänger, im obigen Falle an den Intruder. Der Intruder hat dann die Möglichkeit, die Nachricht an den ordnungsgemässigen Empfänger weiterzuleiten: (EmpfängerAgent, (AbsenderAgent, EmpfängerAgent, EmpfängerRolle,(Nachricht))).

Nicht nur der Intruder hat die Möglichkeit, Nachrichten zu fälschen, auch die beteiligten Player können die Nachrichten vor dem entgültigen Versand abändern. Hierzu dient das GUI (siehe PlayerView, Abb. 3): Dieses kann verschiedenartig implementiert werden, was die Manipulationsmöglichkeiten beeinflusst (siehe Abschnitt 8.1). Das GUI kriegt vom Player alle Events zugespielt, die sich ereignen (z.B. den Erhalt einer neuen Nachricht, den Versand einer Nachricht usw.).

Jeder Player besitzt ein Gesamtwissen (Knowledge), das rollenübergreifend ist. Spielt ein Player mehrere Rollen gleichzeitig, darf er stets auf das übergeordnete Wissen – jenes aller Rollen gemeinsam, sowie Vorwissen und Wissen aus früheren Sessionen – zugreifen. Dies ermöglicht das volle Spektrum für Angriffe.

3.2 Laufzeitarchitektur

Die Simulation wird von den Rollen angetrieben. Wir nutzen aus, dass in HLPSL die Rollen als eigenständige endliche Automaten programmiert werden: Durch Zustandsübergänge werden gewöhnlich Nachrichten verschickt. Diese kommen bei der anderen Rolle an, lösen wiederum einen Zustandsübergang aus, wonach erneut eine Nachricht verschickt wird. Das ist der "Motor" der Simulation.

[2]Player und Agent sind praktisch Synonyme: Wir unterscheiden die beiden nur auf Implementationsbasis. Der Player ist dabei jener, der die Funktionalitäten eines Agenten zur Verfügung stellt und in Objektbeziehungen eingebettet ist. Der Agent ist ein reiner Repräsentant des Players und wird beispielsweise in Nachrichten der Form $\{Na'.A\}Kb$ verschickt.

Durch die Art, *wie* die Nachrichten verschickt und *wo* sie durchgereicht werden, ergeben sich Möglichkeiten zur Manipulation des Protokolls (siehe Abschnitt 3.1).

Die Simulation wird immer mit genau einem Master-Simulator und beliebig vielen – in den meisten Fällen aber einem – Slave-Simulatoren instanziert. Der Master-Simulator regelt dabei alle administrativen Angelegenheiten wie: Welches Protokoll simuliert werden soll, welcher Agent welche Rolle spielt, das Senden der Startnachrichten und das Verschicken der Rollen an die beteiligten Slave-Simulatoren. Letzteres ist besonders interessant und bedarf einiger Erläuterungen: Der Master-Simulator initialisiert die gesamte Simulation, kennt also insbesondere alle teilnehmenden Agenten, alle zu vergebenden Rollen, wie auch welcher Agent welche Rolle spielen wird, sowie die Initialbelegung der Variablen. Er alleine ist im Stande, die Rollen korrekt zu initialisieren (Übergabeparameter setzen) und an die designierten Agenten zu versenden. Verschickt wird letztlich das effektive Rollenobjekt mittels Objektserialisierung.

Die Slave-Simulatoren verfolgen eine Sit-and-Wait Strategie und verhalten sich entsprechend den Anweisungen des Master-Simulators.

3.3 Besonderheiten

Es folgen einige allgemeine Überlegungen sowie erklärende Bemerkungen.

- Die Kanäle werden nach dem Modell von Dolev-Yao [4] als Angreifer aufgefasst. Dies bedeutet, dass jede verschickte Nachricht zuerst in die Hände des Angreifers gelangt. Ob er die Nachrichten weiterleitet, modifiziert oder gar blockiert, ist ihm überlassen. Falls der Angreifer nicht von einem User gespielt wird, leitet er die Nachrichten einfach weiter und stört damit das Protokoll nicht. Weiter werden alle Kanäle (bzw. Angreifer) zusammengefasst und als eine einzige Entität betrachtet[3].

- Die vom Simulator verschickten Nachrichten werden nie effektiv verschlüsselt – selbst wenn das Protokoll dies vorsieht. Stattdessen sind die Nachrichten, wo erforderlich, als verschlüsselt (bzw. signiert) markiert. Der Simulator sorgt dann dafür, dass die Nachricht nur geöffnet wird, wenn der Empfänger den passenden Schlüssel besitzt. Hierbei wird unterschieden, ob es sich um symmetrische oder asymmetrische Schlüssel handelt. Im ersten Fall kennt eine Entität genau dann einen Schlüssel, wenn sie alle Grundkomponenten kennt (Beispiel: wird als Schlüssel h(f(X),g(Y)) gewählt, wobei h, f und g Funktionen sind, reicht es, X und Y zu kennen). Bei asymmetrischen Schlüsseln macht diese Handhabung keinen Sinn, müsste man doch das Inverse von X kennen, um eine mit X verschlüsselte Nachricht zu entschlüsseln. Das Inverse eines Werts versteht sich jedoch immer bezüglich einer konkreten Operation in einer Algebra, die in unserem idealen Modell nicht spezifiziert ist.

- Agenten werden entweder von einem User oder aber automatisiert vom Simulator übernommen. Diese Entscheidung kann bei jeder Protokollinstanzierung neu getroffen werden. Sie ermöglicht zwei Dinge: 1. Auch ein

[3]Hierfür besteht lediglich die Architektur: Ein Intruder ist in der vorliegenden Arbeit nicht implementiert.

einzelner User kann Protokolle durchspielen, ohne zwischendurch die Seiten zu wechseln. 2. Ein Angreifer kann unbemerkt Entitäten kontaktieren (z.B. für parallele Sessionen).

- Dadurch, dass beinahe beliebige HLPSL-Protokolle vom Simulator akzeptiert werden, besteht bezüglich Simulationsmöglichkeiten eine grosse Freiheit.

4 Implementation

Um den Simulator nicht an eine Plattform zu binden, wählten wir Java als Implementationssprache. Eine gut geeignete Entwicklungsumgebung fanden wir im lizenzfreien Eclipse [8], das zugleich CVS [9] unterstützt und ein bequemes Arbeiten zu zweit ermöglicht.

4.1 Parser

Vor jeder Protokollsimulation wird ein HLPSL-Protokoll geparst und internalisiert. Das Ziel des Parsens ist es, fertige Rollenobjekte zu erhalten. Später werden diese Rollenobjekte instanziert (d.h. die Übergabeparameter werden gesetzt und der init-Befehl wird ausgeführt), und an die zugehörigen Player verschickt.

4.1.1 Parsen & Internalisation

Der HLPSL-Parser wurde mit der Hilfe des Parsergenerator Javacup [5] und mit dem Scanner Jflex [6] erzeugt, basierend auf einer in EBNF geschriebenen HLPSL-Grammatik. Liest der HLPSL-Parser ein HLPSL-File ein, baut er gleichzeitig die Rollenobjekte auf. Findet der Parser eine neue Rolle (HLPSL: role), erzeugt er sofort ein Rollenobjekt (Role). Ein Rollenobjekt besitzt die notwendigen Datenfelder, um alle zugehörigen Elemente zu speichern, Variablen, Transitionen, Init und Vorwissen (HLPSL: knowledge), siehe Abb. 4.

Die Transitionen bestehen aus einer linken und einer rechten Seite (left hand side [LHS], bzw. right hand side [RHS]). Beide Seiten sind nach dem Parsen als Baumstruktur gespeichert und werden bei Bedarf (d.h. bei der Evaluation einer Seite, siehe Abschnitt 4.2) ausgewertet.

4.1.2 Einschränkungen gegenüber HLPSL

Es folgt eine Liste der Vereinfachungen, die wir gegenüber Standard-HLPSL trafen.

- Es werden nur sofortige Transitionen (= |>) akzeptiert. Das sind solche, die deterministisch und ohne Zeitverzögerung stattfinden. Die spontanen, nicht-deterministischen Transitionen sind vor allem für "over-the-air" Simulationen gedacht und wurden ausgeklammert.

- Es werden nur Dolev-Yao [4] Kanäle akzeptiert, da Kanäle hauptsächlich die Art und Weise beschreiben, wie eine Nachricht übermittelt wird und deshalb für unsere Zwecke von geringerer Bedeutung sind.

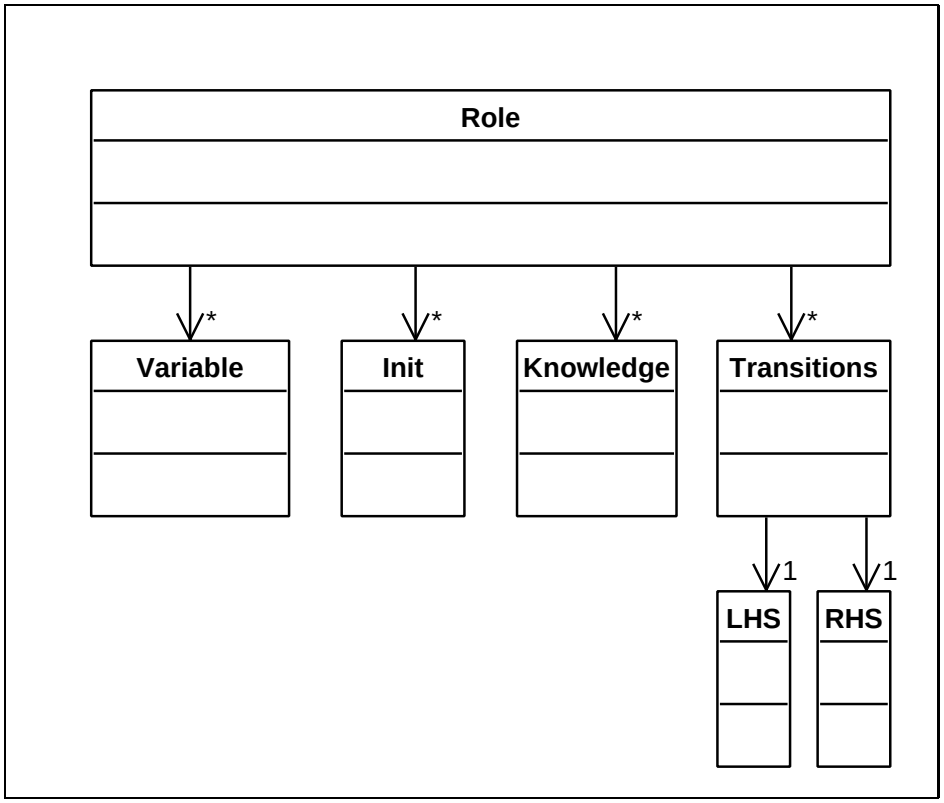

Abbildung 4: Die Simulator-interne Darstellung einer Rolle

- Die in HLPSL erlaubten Composed Roles werden vom Simulator ignoriert, weil diese vor allem dazu dienen, mehrere Protokollinstanzen zu erzeugen. Eine ausgezeichnete Composed Role jedoch, die speziell für den Simulator zu spezifizieren ist und den Identifier "Simulaton_Environment" trägt, wird interpretiert. Diese zusammengesetzte Rolle beschreibt die Instanzierung der Basic Roles nach vorgegebenem Muster (siehe Abschnitt 6).

- Da HLPSL keine Nachrichtenadressierung kennt, ist es nicht möglich, mehr als zwei verschiedene Rollen zuzulassen. Es wäre nicht klar, an welche Rolle eine Nachricht geschickt werden soll.

- Generelle Unifikation wird nicht unterstützt: Eine Transition (LHS) der Form `in(A', List) /\ A'=1` gibt `false` zurück, wenn die Liste `List` aus `{0,1}` besteht, da die Variable `A` im `in`-Statement auf den erstbesten Wert gesetzt wird.

- Das Schlüsselwort `accept` wird ignoriert. Ob die Rolle in einen akteptierenden Zustand gelangt, muss an Hand der Zustandsvariable abgelesen werden. Das Konzept des akzeptierenden Zustands wird normalerweise von zusammengesetzten Rollen verwendet, um automatisiert feststellen zu können, ob eine (Sub-) Rolle ihr Programm beendet hat.

- Das Schlüsselwort `own` wird ignoriert, da es für unsere Zwecke selten einen praktischen Sinn machen würde.

4.2 Evaluator & Expressions

Die in der Rolle enthaltenen Transitionen müssen zur gegebenen Zeit ausgewertet werden. Diese Aufgabe übernimmt die Evaluationsmethode, die im Rollenobjekt integriert ist. Evaluiert wird der vom Parser (siehe Abschnitt 4.1.1) erzeugte Transitionsbaum. Zum Beispiel hat dieser Baum, für den Ausdruck `State=1 /\ RCV({Na'.A}Ka)` als Wurzel einen Und-Knoten, als linkes Kind eine Gleichheits-Knoten und als rechtes Kind einen Receive-Knoten.

Die Evaluationsmethode arbeitet rekursiv über diesem Baum, d.h. sie verknüpft mit einem logischen "Und" – um an obiges Beispiel anzuknüpfen – das Ergebnis des linken mit demjenigen des rechten Teilbaumes und gibt diesen (boolschen) Wert zurück.

Die Auswertung wird etwas schwieriger, wenn es sich um einen Receive-Knoten (RCV) handelt. Die im Receivebuffer anliegende Nachricht muss hier mit der erwarteten Vorgabe (im obigen Beispiel `{Na'.A}Ka`) unifiziert werden.

Nachrichten wie eben `{Na'.A}Ka` werden vom Simulator nicht als Knoten des Transitionsbaums, sondern als Expression repräsentiert. Jeder Receive- und Send-Knoten enthält demnach eine Expression (siehe Abb. 5).

Die Unifikation erfolgt über die Methode `canAdaptTo(Expression e)`, die von jedem Expression-Objekt implementiert wird. `{Na'.A}Ka` ist im obigen Beispiel eine verschlüsselte Expression, ein sogenanntes "Encrypted". Ruft man bei diesem Objekt die Methode `canAdaptTo(message)` auf, prüft es, ob "message" ebenfalls ein "Encrypted" ist, und falls ja, ob sowohl die "canAdaptTo" Methode des Schlüssels wie auch jene des Inhalts `wahr` zurückgibt. Auf diese Weise wird die Unifikation an die entsprechenden Subobjekte delegiert.

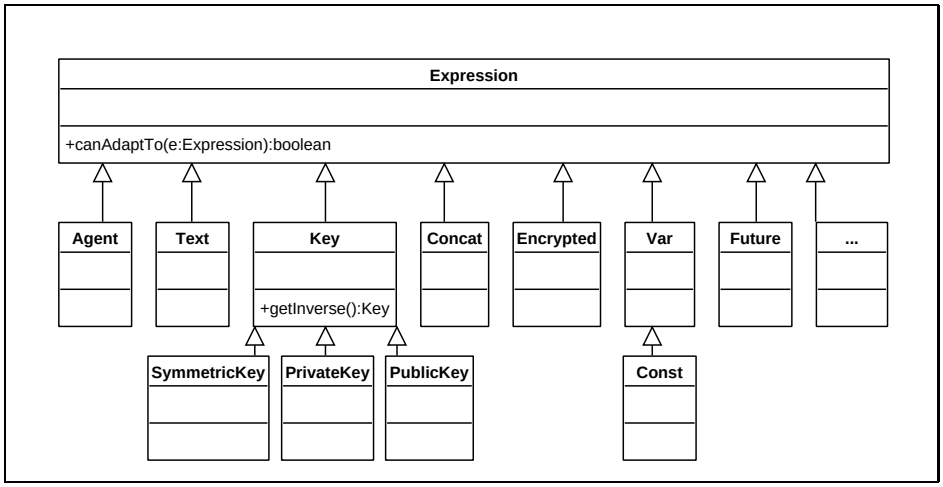

Abbildung 5: Die interne Repräsentation von Nachrichten

4.3 Verteiltheit

Der Simulator funktioniert lokal (auf einer Maschine) wie auch verteilt (auf mehreren Maschinen). Die von uns verwendete, sehr geeignete Lösung heisst JavaGroups [7]. Dieses Package abstrahiert von der effektiven Netzumgebung und erlaubt es, sich an Hand eines Identifiers zu einer Gruppe zusammenzuschliessen, vorausgesetzt, die teilnehmenden Kommunikationspartner befinden sich im gleichen lokalen Netzwerk und dieses wiederum unterstützt Multicast.

5 API

Das API für Entwickler befindet sich in Form einer Javadokumentation im Verzeichnis Javadoc.

6 Benutzeranleitung

In diesem Abschnitt beschreiben wir die Benutzung des Simulators.

6.1 Das Inputfile

Das Sicherheitsprotokoll, das simuliert werden soll, muss folgende Eigenschaften erfüllen (siehe Anhang für ein Beispielprotokoll):

- Es darf nur aus zwei Basisrollen und einer speziellen zusammengesetzten Rolle bestehen, die den Namen "Simulation_Environment" trägt. In dieser müssen die Übergabeparameter der beiden Basisrollen spezifiziert werden.
- Die Basisrollen müssen mit den Einschränkungen gegenüber HLPSL (Abschnitt 4.1.2) konform sein.

- Sie dürfen nicht von zusammengesetzten Rollen abhängig sein, da diese ignoriert werden (Ausnahme "Simulation_Environment").

6.2 Die Konsole

6.2.1 Allgemeine Synopsis

Folgendermassen muss das Jar-File gestartet werden:

```
java -jar SPSimulator.jar -file=<path to input file>
     -name=<name of the agent>
     -group=<a group specifier> -[m|s]
```

Man beachte, dass vor und nach den jeweiligen Gleichheitszeichen keine Abstände enthalten sein dürfen.

Bedeutung:

- -m: Der Simulator wird im Mastermodus gestartet, d.h. er ist ein Master-Simulator.
- -s: Der Simulator wird im Slavemodus gestartet, d.h. er ist ein Slave-Simulator.
- -file: Das zu simulierende Protokoll (HLPSL-File). Muss nur angegeben werden, wenn der Simulator im Mastermodus gestartet wird.
- -name: Der Name des Agenten, der angibt, *welche* Rolle der Simulator zu simulieren gedenkt. Der Name muss exakt derselbe sein wie im HLPSL-File, da hierüber die Rollenverteilung gemacht wird. Wird kein Name spezifiziert, versucht der Simulator defaultmässig die Rollennamen `alice` und `bob` zu vergeben. Finden sich keine solchen im Protokoll, wird die Simulation beendet.
- -group: Ein Identifier für die (JavaGroup-) Gruppe (siehe Abschnitt 4.3). Damit sich zwei Simulatoren finden, muss hier unbedingt derselbe Name angegeben werden.

6.2.2 Beispiel

```
java -jar SPSimulator.jar -file=NSPK -name=alice -m -group=sps0
java -jar SPSimulator.jar -name=bob -s -group=sps0
```

In der ersten Zeile wird der Master-Simulator gestartet, dem auch das Protokoll mitgegeben wird. Er spielt "alice". In der zweiten Zeile wird der Slave-Simulator aufgestartet, der "bob" spielen wird. Hier ist wichtig, dass derselbe Name für die Gruppe gewählt wurde (`sps0`).

6.3 Gezielte Kontrolle

Um vermehrte Kontrolle über den Simulator zu erhalten, muss das Wrapper-File "RunSimulator" im Source-Code verändert werden.

7 Schlussfolgerungen

Im Hinblick auf künftige Arbeiten investierten wir viel Zeit und Aufwand in eine sorgfältige und erweiterbare Architektur. Das vorliegende Resultat erfüllt alle Grundfunktionalitäten, die zur Simulation der Protokolle benötigt werden, wie z.B. das Internalisieren und Interpretieren der Protokolle, deren Umsetzung in eine Simulationsumgebung, sowie die eigentliche Simulation. In Kombination mit einem ansprechenden, noch zu implementierenden GUI (siehe Abschnitt 8.1) eignet sich der Simulator für den Unterricht oder zu demonstrationszwecken. Bereits jetzt hilfreich für das Studium eines Protokolls ist die Ausgabe des Wissensstandes jedes teilnehmenden Agenten zu jedem Zeitpunkt.

8 Ausblick

8.1 GUI und Interaktivität

Um ein GUI aufzusetzen und die damit einhergehende Interaktivität zu erhalten, muss wie folgt vorgegangen werden. Die GUI-Klasse muss einerseits das PlayerListener-Inteface implementieren und zweitens beim Agenten als PlayerListener registriert werden, damit dieser die Schnittstelle beliefern kann.

Die wichtigsten Methoden des PlayerListener-Interface:

```
public void composeMessage(Expression template);
public void determineParticipants(AgentMessage template);
```

Wichtig ist vor allem die Methode `composeMessage(Expression template)`, die die tatsächlich von der Rolle verschickte Nachricht (`template`) als Input nimmt, wonach diese beliebig modifiziert werden kann. Dies geschieht über ein geeignetes GUI.

Über die zweite Methode, `determineParticipants(...)`, können Absender und Empfänger der Nachricht verändert werden. Dies erlaubt sowohl Address-Spoofing wie auch ein Umlenken der Nachricht.

In der aktuellen Implementation erfolgt jeweils eine reine Ausgabe der Nachricht; auch eine Null-Aktion wäre erdenklich, z.B. für einen schlafenden Intruder oder einen "stummen" Bob.

Des weiteren wäre ein Message Sequence Chart (MSC, siehe Abb. 6) denkbar, der alle verschickten Nachrichten aufzeichnet und chronologisch darstellt. Auch die Visualisierung der endlichen Automaten (die Rollen!) könnte dem User einen Nutzen bereiten.

Im Anhang finden sich unsere Überlegungen, wie ein interaktives GUI funktionieren könnte.

8.2 Angriffe

Die Simulation von Angriffen ist soweit schon integriert, als – wenn das GUI und die Interaktivität bereit stehen – sich lediglich ein dritter User (Intruder) zwischen die Agenten schalten muss. Vom reinen Mitlesen zur Modifikation oder gar dem Ersetzen einer Nachrichten ist alles vorbereitet.

Mehr Aufwand – intellektuell wie auch manuell – dürfte die allgemeine Attackensimulation bereiten: Parallele Sessionen sei hier das Stichwort.

8.3 Erweiterung auf > 2 Rollen

Hierfür muss ev. HLPSL erweitert oder uminterpretiert werden.

8.4 Einbinden der AVISPA-Tools

Dies erfordert konzeptionell wie auch im Arbeitsaufwand einige Leistung. Speziell angesprochen sei die sinnvolle Darstellung einer Attacke im Simulator, sowie rein technisch die Interpretation des Ausgabeformats dieser Tools.

9 Danksagung

Unser Dank geht in erster Linie an die drei Assistenten Michael Näf, Paul Hankes Drielsma und Paul E. Sevinç für deren unermüdlichen Einsatz mit Rat und Tat bei zahlreichen und zumeist langen Sitzungen!

Des weiteren freute uns sehr, bei Prof. David Basin eine Semesterarbeit durchführen zu können, wofür wir uns an dieser Stelle bedanken möchten.

A Quellenangaben

Literatur

[1] AVISPA – Automated Validation of Internet Security Protocols and Applications, `http://www.avispa-project.org`

[2] HLPSL – High Level Protocol Specification Language, wird demnächst unter `http://www.avispa-project.org` publiziert.

[3] Needham-Schröder Public Key Protocol, `http://dimacs.rutgers.edu/Workshops/Security/program2/boyd/node14.html`

[4] Dolev-Yao Kanäle, `http://www.dychannels.org`

[5] Javacup, der Parsergenerator, `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[6] JFlex, der schnelle Scanner, `http://www.jflex.de/`

[7] JavaGroups, die Bibliothek für einfache Gruppenbildung, `http://www.jgroups.org`

[8] Eclipse, die lizenzfreie Entwicklungsumgebung, `http://www.eclipse.org`

[9] Concurrent Versioning System, `http://www.cvs-home.com`

B GUI-Studien

Im Folgenden finden sich die Gedanken, die wir bezüglich GUI bereits angestellt, aus zeitlichen Gründen jedoch nicht umgesetzt hatten.

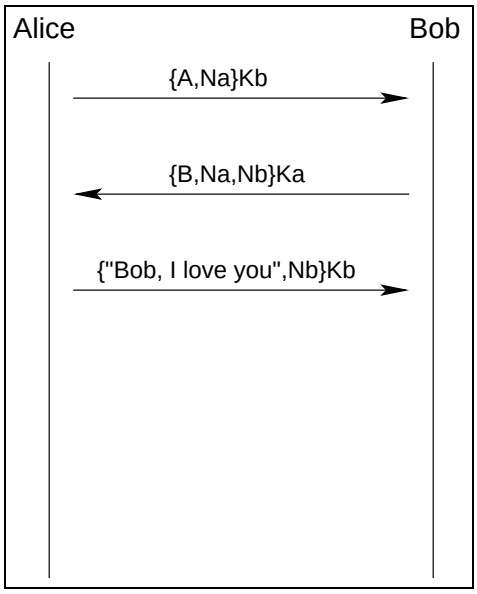

Abbildung 6: Ein Message Sequence Chart

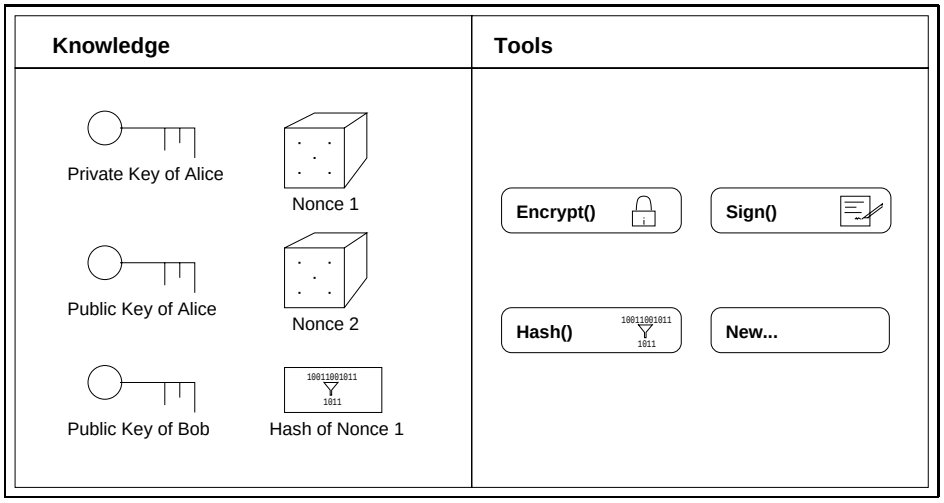

Abbildung 7: Hauptfenster eines Users

Dem Simulations-GUI kommt besondere Bedeutung zu, macht es doch praktisch alleine die didaktisch geforderte Verständlichkeit aus. Die Schnittstelle mit den Usern soll deshalb möglichst intuitiv und leicht bedienbar sein.

Um ein Sicherheitsprotokoll zu verstehen, ist das aktuelle Wissen eines jeden Agenten von zentraler Bedeutung. Deshalb wird das GUI für jeden Agenten ein (Simulations-)Fenster öffnen, in dem links das Gesamtwissen grafisch und beschreibend dargestellt wird (siehe Abbildung 7). Kommen Nachrichten von anderen Agenten an, wird das Gesamtwissen aktualisiert. Rechts sind Funktion zur Manipulation von Nachrichten untergebracht.

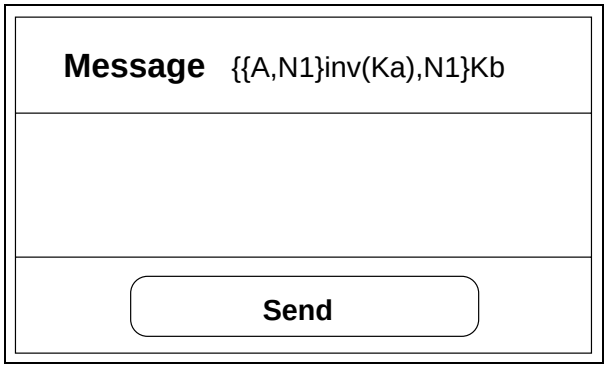

Abbildung 8: Messagefenster

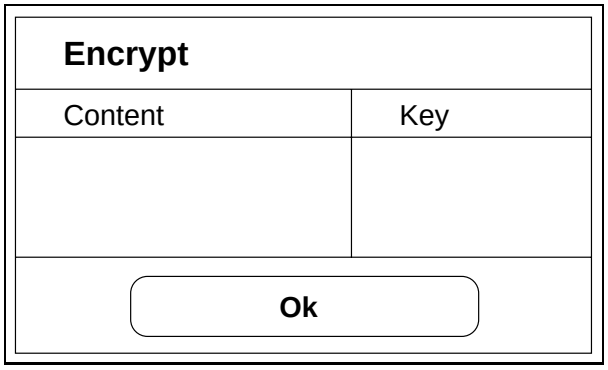

Abbildung 9: Verschlüsselungsfenster

Nachrichten werden verfasst, indem die zu verschickenden Elemente (Schlüssel, Nonces, Hashes, etc...) in ein Nachrichtenfenster gezogen werden. Das Nachrichtenfenster enthält einen Hilfstext, der das Soll-Format der Nachricht anzeigt (siehe Abb. 8).

Bei Bedarf werden Tool-Funktionen (siehe Abb. 7, rechts) mit der Maus in das Nachrichtenfenster gezogen, wonach sich sofort ein neues Fenster öffnet, in welches wiederum beliebige Icons gezogen werden können. Beispiel: Alice möchte `{A}Kb` an Bob schicken. Sie zieht dazu das Encrypt()-Icon in die Nachricht, und unmittelbar nach dem Loslassen öffnet sich ein Encrypt-Fenster (siehe Abb. 9), in welchem sie links `A` (Inhalt) und rechts `Kb` (Schlüssel) reinziehen kann. Nach

einem Ok-Klick geht das Encrypt-Fenster zu und das ursprüngliche Encrypt-Icon verwandelt sich optisch (z.B. wird es dunkler), so dass Alice erkennen kann, dass dort etwas drinsteht. Mit einem Klick auf "Send" verschwindet das Nachrichtenfenster und die Botschaft wird abgeschickt.

C NSPK-Protokoll

```
role Alice ( A,B : agent,
             Ka,Kb : public_key,
             SND,RCV : channel(dy)) played_by A def=
exists State : nat, Na : text(fresh), Nb : text
init State=0
knowledge(A) = { inv(Ka) }
transition
      step1. State=0 /\ RCV(start) =|>
             State'=1 /\ SND({Na'.A}Kb)
      step2. State=1 /\ RCV({Na.Nb'}Ka) =|>
             State'=2 /\ SND({Nb'}Kb)
end role

role Bob   ( A,B : agent,
             Ka,Kb : public_key,
             SND,RCV : channel(dy)) played_by B def=
exists State : nat, Nb : text(fresh), Na : text
init State=0
knowledge(B) = { inv(Kb) }
transition
      step1. State=0 /\ RCV({Na'.A}Kb) =|>
             State'=1 /\ SND({Na'.Nb'}Ka)
      step2. State=1 /\ RCV({Nb}Kb) =|>
             State'=2
end role

role Simulation_Environment() def=
composition
      Alice(alice,bob,ka,kb,sa,ra) /\ Bob(alice,bob,ka,kb,sa,ra)
end role

Simulation_Environment()
```