

Hauptseminar
Seminar Modellierung und Verifikation
kryptografischer Protokolle

Betreuer:
Frank Kargl und Stefan Schlott

Wintersemester 2002/2003
Abteilung Medieninformatik
Universität Ulm

Inhaltsverzeichnis

1	Verifikation mit Zustandsautomaten	2
1.1	Einleitung	2
1.2	Analyseverfahren für kryptographische Protokolle	3
1.3	Verifikation mit Zustandsautomaten	3
1.4	Modellierung eines Protokolls mit Zustandsautomaten	5
1.5	Die Methodik	7
1.6	Verifikation mit Mur φ	8
1.7	Das Needham-Schröder Protokoll	9
1.8	Modellierung von Needham-Schröder mit Mur φ	11
1.9	Schlussbemerkung	14

Kapitel 1

Verifikation mit Zustandsautomaten

Stefan Schreinert
stefan@schreinert.com

Abstract: Kryptographische Protokolle spielen im täglichen Leben eine immer wichtigere Rolle. Doch für einen sicheren Einsatz der Protokolle ist es von entscheidender Bedeutung, dass die kryptographischen Verfahren korrekt sind. Da die menschliche Kontrolle nicht ausreichend ist, bedarf es einer automatisierten Überprüfung durch einen Computer. Diese Arbeit zeigt, wie kryptographische Protokolle weitestgehend automatisch verifiziert werden können, indem die Protokolle auf endliche Zustandsautomaten übertragen werden. Nach den theoretischen Grundlagen wird abschließend gezeigt, wie das Needham-Schröder Protokoll mithilfe des Werkzeugs Mur φ modelliert und verifiziert werden kann.

1.1 Einleitung

Kryptographische Methoden und Protokolle spielen eine wichtige Rolle im täglichen Leben. Sei es, um eine eMail zu verschlüsseln, eine Warenbestellung elektronisch zu signieren oder aber sich gegenüber einem anderen zu authentisieren. Für diese Einsatzgebiete kommen verschiedene kryptographische Protokolle zum Einsatz, deren Korrektheit entscheidend für ihre Einsatzfähigkeit ist. Aus diesem Grund muss ein großes Augenmerk darauf gelegt werden, ein neues Protokoll zu verifizieren.

Die Verifikation der Protokolle durch Menschen ist jedoch sehr aufwendig und liefert dennoch nicht die nötige Korrektheit. So wurden über längere Zeit Protokolle eingesetzt, die von Experten als korrekt und damit sicher eingestuft wurden, es jedoch nicht waren. Beispielsweise enthielt das Needham-Schröder-Lowe (NSL) Protokoll einen Fehler, der einem Angreifer ermöglichte, einen alten, abgefangenen

Sitzungsschlüssel erneut an den Kommunikationsteilnehmer zu schicken und sich somit Zutritt zu verschaffen ([den81]).

Das NSL-Beispiel ist nur eines von vielen, welches zeigt, dass kryptographische Protokolle nicht ausschließlich durch menschliche Experten verifiziert werden sollten. Für eine automatisierte Verifikation durch einen Computer muss aber das Protokoll zum einen in einer geeigneten Form im Computer modelliert werden und zum anderen bedarf es Algorithmen, die mit vertretbarem Ressourcenaufwand Protokollfehler entdecken können.

1.2 Analyseverfahren für kryptographische Protokolle

Es gibt zahlreiche Verfahren, um kryptographische Protokolle zu analysieren und damit Fehler zu finden. Eine davon ist die BAN-Logik, welche die Überzeugungen und das Wissen der Protokollteilnehmer modelliert. Geprüft wird, ob der unberechtigte „Teilnehmer“ (der Angreifer) an Wissen gelangt, welches er niemals bekommen darf. Beispielsweise muss es unmöglich sein, dass der Angreifer an einen gültigen Sitzungsschlüssel kommt. Die BAN-Logik wird zumeist für Tests eingesetzt, die ein menschlicher Experte durchführt.

Des Weiteren gibt es die Klasse der Theorembeweiser und Modell-Prüfer, zu denen etwa auch das NRL-Protokoll gehört. Dieses Tool modelliert das zu prüfende Protokoll in der umgekehrten Reihenfolge, so dass der fehlerhafte Zustand, welcher nicht erreicht werden darf, zum Startzustand wird. Geprüft wird, ob es einen Weg zum ursprünglichen Startzustand des Protokolls gibt. Ist ein Weg gefunden, so existiert folglich im zu prüfenden Protokoll ein Weg vom (ursprünglichen) Startzustand zu dem fehlerhaften Zustand, bei dem etwa ein gültiger Sitzungsschlüssel dem Angreifer bekannt ist. Damit wäre das kryptographische Protokoll fehlerhaft.

Diese Arbeit beschäftigt sich mit einer weiteren Klasse von Analyseverfahren: der Verifikation mit Zustandsautomaten. Auch hier wird ein Weg gesucht, der vom Startzustand in einen ungültigen Zustand endet.

1.3 Verifikation mit Zustandsautomaten

Die Verifikation kryptographischer Protokolle mit endlichen Zustandsautomaten (Finite-State Machines) basiert darauf, jedes mögliche Verhalten des Protokolls erschöpfend zu simulieren. Dies bedeutet, dass beginnend vom Startzustand s_0 jeder mögliche Zustandsübergang durchprobiert wird, um so zu einem Zustand zu finden, der nicht erreicht werden dürfte. Der Algorithmus 1 zeigt den grundlegenden Analyse-Algorithmus.

Der Algorithmus arbeitet auf Basis der Mengen Q , $Unready$ und Δ (Zeile 2), wobei Q die Menge aller gefundenen Zustände ist (beginnend mit den Startzuständen (hier q_0)) und $Unready$ die Menge aller noch nicht bearbeiteten Zustände ist. Die Liste aller Übergänge wird in Δ gespeichert (Zeile 16). Der Algorithmus iteriert

Algorithmus 1 Erreichbarkeits-Analyse Algorithmus nach [hel98]

```

1: proc create_rg  $\equiv$ 
2:  $\Delta := \emptyset; Q := \{q_0\}; Unready := Q;$ 
3: while  $Unready \neq \emptyset$  do
4:   choose any  $q \in Unready;$ 
5:    $Unready := Unready \setminus \{q\};$ 
6:   foreach  $t \in T$  such that  $t$  is enabled at  $q$  do
7:     foreach  $q'$  such that  $(q, t, q')$  is a transition do
8:       if  $\neg \text{assert\_ok}(q')$  then
9:          $\text{report\_error}(q');$ 
10:         $\text{exit}();$ 
11:      fi
12:      if  $q' \notin Q$  then
13:         $Q := Q \cup \{q'\};$ 
14:         $Unready := Unready \cup \{q'\};$ 
15:      fi
16:       $\Delta := \Delta \cup \{(q, t, q')\};$ 
17:    od
18:  od
19: od

```

solange, bis keine unbearbeiteten Zustände in der Menge $Unready$ vorhanden sind (Zeile 3).

Dazu nimmt der Algorithmus einen beliebigen, noch nicht bearbeiteten Zustand aus $Unready$ heraus (Zeile 4), entfernt diesen aus $Unready$ (Zeile 5) und bearbeitet *alle* Zustandsübergänge t mit den entsprechenden Zielzuständen q' (Zeilen 6 und 7). Damit werden alle Zustände gefunden, die von q aus *unmittelbar* erreichbar sind. Wird nun ein Zustand q' erreicht, der zuvor markiert wurde als „darf nicht erreicht werden“ (Zeile 8), so ist ein Protokollfehler gefunden und der Algorithmus kann abbrechen (Zeile 9 und 10). Im anderen Fall wird der neu gefundene Zustand q' sowohl in die Menge der noch zu prüfenden Zustände $Unready$ aufgenommen (Zeile 14) als auch in die Menge gefundener Zustände (Zeile 13). In der Zeile 16 werden die gefundenen Übergänge als Tupel (Ausgangszustand, Zustandsübergang, Zielzustand) gespeichert.

Finite-State Verfahren unterliegen dabei Einschränkungen beziehungsweise Annahmen, welche zwar die Verarbeitungsgeschwindigkeit erhöhen, dafür aber auch die Anzahl der auffindbaren Angriffsarten begrenzen. Einige der Annahmen sind ([hel98]):

- **Keine Kryptoanalyse.** Es wird davon ausgegangen, dass etwa die verwendete Nachrichtenverschlüsselung perfekt sei. Damit ist es nur dann möglich, die Nachricht zu lesen, wenn ein gültiger, privater Schlüssel vorliegt. Ein An-

greifer ist typischerweise nicht im Besitz dieses Schlüssels.

- **Feste Anzahl von Protokollteilnehmern.** Normalerweise ist es gestattet, dass mehrere Teilnehmer gleichzeitig das zu untersuchende Protokoll verwenden, etwa wenn sich zwei Teilnehmer gleichzeitig gegenüber einem Dritten authentifizieren möchten. Lowe [lowe96] hat gezeigt, dass ein Protokoll auch für eine beliebige Anzahl von Teilnehmern korrekt ist, wenn es für eine fixe Anzahl von Teilnehmern korrekt ist.
- **State explosion problem.** Dies ist ein großes Problem der Analyseverfahren mit Zustandsautomaten. Mit jedem neuen Zustand nimmt die Komplexität des Systems exponentiell zu. Dies macht es nötig, die Anzahl der modellierten Zustände zu reduzieren beziehungsweise klein zu halten.
- **Begrenzte Anzahl Protokollläufe.** Mit jedem Protokolllauf vergrößert sich das Wissen des Angreifers. Durch die Begrenzung der Protokollläufe wird dem simulierten Angreifer während der Analyse verwehrt, potentiell benötigte Informationen über das Protokoll zu sammeln.
- **Begrenzter Speicher des Angreifers.** Zudem muss der Speicher für die Wissensbasis des Angreifers begrenzt werden. Damit verliert der Angreifer potentiell wertvolle Informationen.

Aus diesen Gründen lässt sich mit Zustandsautomaten nur prüfen, ob ein kryptographisches Protokoll unsicher ist, nicht aber, ob das Protokoll korrekt ist: Meldet der Verifikationsprozess einen Fehler, so ist das untersuchte Protokoll unsicher. Findet der Prozess keinen Fehler, so kann nichts über die Sicherheit des Protokolls gesagt werden, wenn etwa mehrere gleichzeitige Protokollläufe vorkommen, als bei der Verifikation angenommen.

Der Vorteil solcher Annahmen liegt darin, den zu simulierenden Suchraum einzugrenzen. Im anderen Fall müsste der Verifikationsprozess über einen sehr großen bis unendlichen Raum iterieren, wodurch die Verifikation nicht innerhalb akzeptabler Zeit ablaufen würde.

1.4 Modellierung eines Protokolls mit Zustandsautomaten

Wie bereits beschrieben, werden die zu untersuchenden kryptographischen Protokolle dahingehend begrenzt, dass die Anzahl der Teilnehmer und Protokollläufe fixiert werden. Im nachfolgenden existieren nur die minimal notwendigen Teilnehmer, Alice und Bob. Diese tauschen über ein Kommunikationsnetzwerk Nachrichten miteinander aus, um so etwa die Authentizität ihres Gegenüber festzustellen (vgl. Abbildung 1.1).

Bei der Modellierung des Protokolls wird dabei das Kommunikationsnetzwerk durch einen weiteren Teilnehmer ersetzt, dem Angreifer.

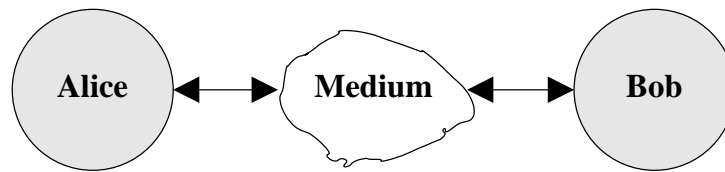


Abbildung 1.1: Das Kommunikations-Modell

Es wird unterstellt, dass der Angreifer alle zwischen Alice und Bob gesendeten Nachrichten abfangen kann, wodurch Alice und Bob nicht mehr direkt miteinander kommunizieren, sondern nur noch über den Angreifer, wie in Abbildung 1.2 dargestellt.

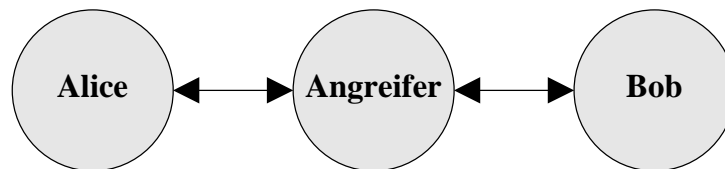


Abbildung 1.2: Das Protokoll-Modell

Damit erhält der Angreifer die totale Kontrolle über das Kommunikationsmedium. Er kann somit jede Nachricht verändern, zurückhalten oder ganz löschen. Auch kann er eigene Nachrichten verschicken und natürlich Informationen aus den abgefangenen Nachrichten sammeln. Dazu speichert der Angreifer alle Informationen einer Nachricht in seiner Wissensbasis, wodurch er zu einem späteren Zeitpunkt etwa eine neue Nachricht generieren kann oder aber eine abgefangene Nachricht unverändert verschicken kann. Die Empfänger einer solchen unautorisierten Nachricht können dabei jedoch nicht unterscheiden, ob eine Nachricht vom Angreifer kommt oder vom legitimen Kommunikationspartner.

Die Eigenschaft, dass ein Angreifer das Kommunikationsnetzwerk derart beherrscht ist sicher nicht immer gegeben. Vielmehr dürfte ein Angreifer in der Wirklichkeit nur einige der beschriebenen Methoden anwenden können oder aber nicht alle Nachrichten gehen über seinen „Schreibtisch“. Anstatt aber diese Möglichkeiten als optional zu kennzeichnen, wird dennoch unterstellt, dass ein Angreifer jede Nachricht erhält und mit dieser alles beschriebene anstellen kann. Dies reduziert die Anzahl möglicher Zustände beziehungsweise Zustandsübergänge, wodurch der Suchraum reduziert wird. Dennoch findet der Analyse-Algorithmus alle Fehler, die er auch ohne die Reduzierung gefunden hätte.

Zusätzlich zur Kommunikations-Infrastruktur müssen die Nachrichten modelliert werden. Ein Thema hierbei ist, welche Informationen einer Nachricht gelesen

werden können. Es wird davon ausgegangen, dass die Algorithmen, welche den Inhalt einer Nachricht verschlüsseln, perfekt sind und somit der Nachrichteninhalt nur mit dem gültigen Schlüssel dechiffriert werden kann. Ein Angreifer kann folglich die verschlüsselten Felder einer Nachricht nicht lesen. Das Modell einer Nachricht muss deshalb genau spezifizieren, welche Felder einer Nachricht gelöscht oder verändert werden können und welche Felder verschlüsselt sind und folglich weder genutzt noch geändert werden können.

Bei der Modellierung eines Zustandes müssen die möglichen Zustandsübergänge und deren Bedingungen beachtet werden. Basiert der Analyse-Algorithmus auf der Erreichbarkeits-Analyse, so muss zu einem Zustand, der nicht erreicht werden darf, diese Information gekennzeichnet werden. Gelangt der Algorithmus dennoch in diesen ungültigen Zustand, so bricht die Analyse ab und meldet einen Fehler im untersuchten kryptographischen Protokoll.

Anders bei Analyse-Algorithmen, die auf „safety properties“ beruhen ([hel98]). Hier sammeln die Protokollteilnehmer — einschließlich dem Angreifer — über die Zeit Informationen. Ein zusätzlicher Observer-Prozess prüft dann permanent, ob die Wissensbasis eines Protokollteilnehmers auch wirklich nur die Informationen enthält, die vom Protokoll vorgesehen sind. Befindet sich beispielsweise ein gültiger Sitzungsschlüssel in der Wissensbasis des Angreifers, so ist ein Protokollfehler gefunden und der Analyse-Algorithmus bricht ab.

1.5 Die Methodik

Nach [mitc97] folgen die Analyseverfahren den dargestellten Schritten:

1. *Das Protokoll formulieren.* Dazu muss das zu untersuchende Protokoll vereinfacht werden. Dabei müssen die wichtigsten Protokollvorgänge und -primitiven herausgefunden werden, um die Charakteristika des kryptographischen Protokolls zu bewahren. Das wichtigste dabei ist, genau zu entscheiden, welche Nachricht durch einen Kommunikationspartner akzeptiert wird.
2. *Einen Angreifer hinzufügen.* Es wird allgemein angenommen, dass der Angreifer ein „normaler“ Teilnehmer des Systems ist, welcher auch in der Lage ist, eine Kommunikation zu initiieren. Der Angreifer kontrolliert dabei das gesamte Netzwerk.
Obwohl es am einfachsten ist, den Angreifer so zu modellieren, dass er nicht-deterministisch zwischen *allen* möglichen Aktionen in jedem Protokollschritt wählt, ist es effizienter die Auswahl auf die Aktionen zu begrenzen, die potentiell die anderen Kommunikationsteilnehmer beeinflussen.
3. *Die gewünschten Korrektheitsbedingungen angeben.* Es müssen alle Bedingungen formuliert werden, die in einem korrekten Protokoll erfüllt sein müssen.
4. *Protokollläufe starten.* Jetzt kann das modellierte Protokoll dem ersten Test unterzogen werden. Dazu wird das Analyseverfahren mit einigen Parametern

gestartet, etwa vier bis fünf Teilnehmern einschließlich dem Angreifer. Auf modernen Computern sollte dieser Protokolllauf nach wenigen Sekunden bis Minuten beendet sein. Werden die Parameterwerte etwa verdoppelt, so kann ein Lauf schon mehrere Stunden benötigen oder er terminiert aufgrund zu extensiven Speicherverbrauchs.

5. *Alternative Formulierungen verwenden.* Haben die Schritte 1–4 keinen Protokollfehler entdeckt, so wird die beschriebene Methodik wiederholt. Dabei erhält beispielsweise der Angreifer mehr Speicher für seine Wissensbasis oder aber der Angreifer konzentriert sich auf eine andere Angriffsvariante. Ebenso kann das Protokoll neu modelliert werden, etwa indem Annahmen entfernt oder begrenzt werden. Wenn auch die veränderten Modelle keinen Protokollfehler finden, so kann nun noch versucht werden, dem Angreifer einige Teile der geheimen Information zu zuspieren.

Wie vor allem am letzten Schritt zu sehen ist, kann die Finite-State Analyse nicht als automatisiertes Analyseverfahren gesehen werden. Vielmehr bedarf es der Kreativität und der Erfahrung der menschlichen Prüfer.

1.6 Verifikation mit Mur φ

Mur φ ist ein Werkzeug zur Verifikation mit Zustandsautomaten und wurde bereits mehrfach erfolgreich eingesetzt, um industrielle Protokolle zu überprüfen, besonders im Bereich der Cache-Kohärenz in Multiprozessoren, und auch um Sicherheitsprotokolle zu verifizieren.

Um ein kryptographisches Protokoll mit Mur φ zu verifizieren, müssen das Protokoll und der Angreifer modelliert werden ([mitc97], [lowe96ns]). Zudem müssen die Eigenschaften des Systems spezifiziert werden. Daraufhin überprüft Mur φ selbständig und automatisch, ob alle erreichbaren Zustände die angegebenen Eigenschaften erfüllen.

Mur φ verwendet dazu die bereits beschriebene Technik der Erreichbarkeits-Analyse, indem alle möglichen Zustände durchlaufen werden. Der Anwender kann dabei spezifizieren, ob er eine breadth-first oder depth-first Suche bevorzugt. Bereits erreichte Zustände speichert das Werkzeug in einer Hashtabelle, um redundante Überprüfungen auszuschließen, was letztendlich der Performance zugute kommt. Die Größe der Hashtabelle entspricht typischerweise dem größten auffindbaren Problem.

Die benötigten Angaben über das Protokoll und den Angreifer werden über eine Mur φ -eigene, einfache high-level Sprache beschrieben. Mit dieser Sprache ist es möglich, nicht-deterministische endliche Zustandsautomaten zu definieren. Die meisten Befehlskonstrukte ähneln denen von herkömmlichen high-level Sprachen wie in C oder Java. Dazu kommen noch spezielle Erweiterungen, die nachfolgend beschrieben werden sollen.

Der Zustand des Modells besteht aus den Werten aller globalen Variablen. Im Startzustand werden diese globalen Variablen mit initialen Werten belegt. Der Übergang von einem Zustand in einen anderen wird durch Regeln beschrieben, wobei jede Regel eine boolesche Bedingung und eine Aktion enthält. Die Aktion ist dabei ein Programmcode, der ausgeführt wird, wenn die Bedingung wahr ist. Typischerweise verändert eine Aktion die globalen Variablen des Systems, wodurch ein neuer Systemzustand entsteht. Die meisten $\text{Mur}\varphi$ Modelle sind dabei nicht-deterministisch, so dass mehr als eine Regel wahr sein kann und folglich mehr als eine Aktion ausgeführt wird. Damit ist es beispielsweise möglich, einen Angreifer zu modellieren, der normalerweise nicht-deterministisch zwischen verschiedenen, neu zu sendenden Nachrichten wählen kann.

$\text{Mur}\varphi$ ermöglicht die parallele Ausführung mehrerer Prozesse, wobei mehrere Prozesse durch einen Satz von Regeln definiert werden. Die Prozesse können dabei nur über gemeinsame Variablen kommunizieren, eine andere Kommunikationsmöglichkeit besteht nicht. Ein Prozess kann eine beliebige Anzahl von Aktionen ausführen. Dies bedeutet, dass das Rechenmodell *asynchronous, interleaving concurrency* ist.

Die gewünschten Eigenschaften werden in $\text{Mur}\varphi$ über Konstanten spezifiziert. Diese Konstanten sind boolesche Bedingungen, welche in jedem Zustand gültig (wahr) sein müssen. Sobald ein Zustand eine solche Bedingung verletzt, gibt $\text{Mur}\varphi$ eine Fehlerbeschreibung aus — einschließlich des Weges vom Startzustand zum Fehlerzustand. Damit wäre ein Protokollfehler gefunden.

Die Konstanten, welche die gewünschten Eigenschaften des Systems beschreiben, sind typischerweise der Form „Ein Angreifer kennt das Geheimnis X nicht“ oder „Wenn der Teilnehmer Alice im Zustand s_1 ist, dann muss der Teilnehmer Bob im Zustand s_2 sein“.

1.7 Das Needham-Schröder Protokoll

Als Beispiel-Protokoll für die Funktionsweise von $\text{Mur}\varphi$ soll das Needham-Schröder Protokoll gezeigt werden ([mitc97]). Das Needham-Schröder Protokoll sorgt dafür, dass sich zwei Teilnehmer einander authentifizieren können. Beide Teilnehmer, der Initiator A und der Responder B , sollen anschließend sicher sein können, auch wirklich mit dem Partner zu kommunizieren, für den er sich ausgibt. Die vereinfachte Darstellung des Needham-Schröder Protokolls besteht aus drei Schritten, wie der Algorithmus 2 zeigt.

Algorithmus 2 Needham-Schröder Protokoll (fehlerhaft) nach [mitc97]

$$A \rightarrow B : \{N_a, A\}_{K_b}$$

$$B \rightarrow A : \{N_a, N_b\}_{K_a}$$

$$A \rightarrow B : \{N_b\}_{K_b}$$

Im ersten Schritt sendet der Initiator A ein neuen, zufälligen Authentifikationsschlüssel (auch als Nonce bezeichnet) N_a zusammen mit seinem Namen an den Responder B . Beide Angaben sind verschlüsselt mit dem öffentlichen Schlüssel K_b von B . Der Responder B hat als einziger den privaten Schlüssel und kann somit die Nachricht von A entschlüsseln. Im zweiten Schritt generiert daraufhin B ebenfalls einen neuen, zufälligen Authentifikationsschlüssel N_b und schickt sowohl N_a als auch N_b an den Initiator A zurück — beides verschlüsselt mit dem öffentlichen Schlüssel K_a von A . Der Initiator A entschlüsselt die Nachricht und prüft, ob sein mitgeschickter Authentifikationsschlüssel N_a auch derjenige ist, den er an B geschickt hat. Denn nur B ist in der Lage, das verschlüsselte N_a zu entschlüsseln. Ist der Authentifikationsschlüssel N_a korrekt, so kann A sicher sein, dass der Authentifikationsschlüssel N_b vom korrekten Responder B stammt. Im dritten Schritt bestätigt der Initiator A auf ähnliche Weise seine Identität, indem er den Authentifikationsschlüssel N_b an B verschlüsselt zurückschickt. Da N_b von B mit dem öffentlichen Schlüssel von A verschlüsselt wurde, kann nur A den Authentifikationsschlüssel N_b kennen, womit auch A sich gegenüber B authentifiziert hat.

Dieser hier dargestellte Needham-Schröder-Algorithmus ist jedoch fehlerhaft, wie nachfolgend gezeigt wird. Ein Angreifer I möchte sich gegenüber dem Responder B als Initiator A ausgeben. Sei der Angreifer I ein *autorisierter* Teilnehmer des Protokolls. So muss I darauf warten, bis A eine Verbindung mit *ihm* aufbaut ([lowe96ns]):

$$A \rightarrow I : \{N_a, A\}_{K_i}$$

Der Angreifer I baut sodann eine Verbindung mit B auf, gibt sich diesem gegenüber als A aus und sendet ihm den Authentifikationsschlüssel N_a , den er gerade vom wahren Teilnehmer A erhalten hat (I kann N_a lesen, da N_a mit dem öffentlichen Schlüssel K_i von I verschlüsselt ist):

$$I(A) \rightarrow B : \{N_a, A\}_{K_b}$$

Der Responder B antwortet an I mit seinem neu erzeugten Authentifikationsschlüssel N_b und verschlüsselt diesen mit dem öffentlichen Schlüssel K_a von A :

$$B \rightarrow I(A) : \{N_a, N_b\}_{K_a}$$

Da der Angreifer I die Nachricht von B nicht lesen kann (diese ist ja nur von A entschlüsselbar), vermag er auch nicht den Authentifikationsschlüssel N_b an B zurückzuschicken. Stattdessen schickt I die soeben erhaltene, für A verschlüsselte Nachricht an A :

$$I \rightarrow A : \{N_a, N_b\}_{K_a}$$

A entschlüsselt N_a und N_b und prüft, ob N_a auch der Authentifikationsschlüssel ist, den er an I geschickt hat. Damit ist A davon überzeugt, dass I auch wirklich I ist (I hat den Authentifikationsschlüssel N_a entschlüsseln können, was nur

mit dem privaten Schlüssel von I möglich ist). A bestätigt daraufhin seine Identität, indem er I den erhaltenen Authentifikationsschlüssel N_b zurückschickt — im Glauben, der Authentifikationsschlüssel würde tatsächlich von I kommen. Damit aber macht A den von B erzeugten Authentifikationsschlüssel für den Angreifer I lesbar:

$$A \rightarrow I : \{N_b\}_{K_i}$$

Jetzt ist der Angreifer I im Besitz des Authentifikationsschlüssels N_b von B , den B eigentlich an A geschickt hatte und von I abgefangen wurde (I kontrolliert ja das gesamte Netzwerk und kann somit Nachrichten abfangen). I ist damit in der Lage, sich gegenüber B als A zu authentifizieren:

$$I(A) \rightarrow B : \{N_b\}_{K_b}$$

Nun prüft B , ob der übermittelte Authentifikationsschlüssel N_b dem entspricht, den er zuvor an A geschickt hatte und der von I abgefangen wurde. Da der Authentifikationsschlüssel N_b korrekt ist, geht B davon aus, wirklich mit A zu sprechen, auch wenn er in Wirklichkeit mit I kommuniziert.

Über 17 Jahre blieb dieser Fehler unentdeckt, mit Mur ϕ wurde der Fehler nach 1,7 Sekunden gefunden. Als Lösung dieses Fehlers muss der Responder B zusätzlich zu den beiden Authentifikationsschlüsseln N_a und N_b seine Identität mit-schicken, wie nachfolgend dargestellt (Algorithmus 3):

Algorithmus 3 Needham-Schröder Protokoll (korrigiert)

$$\begin{aligned} A \rightarrow B & : \{N_a, A\}_{K_b} \\ B \rightarrow A & : \{N_a, N_b, B\}_{K_a} \\ A \rightarrow B & : \{N_b\}_{K_b} \end{aligned}$$

Es sei noch erwähnt, dass auch der Algorithmus 3 vereinfacht dargestellt ist. Das vollständige Needham-Schröder Protokoll umfasst noch einen weiteren Teilnehmer, nämlich einem Schlüssel-Server S , dem sowohl A als auch B vollständig vertrauen ([ma00fo]).

1.8 Modellierung von Needham-Schröder mit Mur ϕ

Dieser Abschnitt soll einen kleinen Einblick in die Programmierstrukturen von Mur ϕ geben ([mitc97]). Abgebildet ist nur initiale Teil, da dieser die Möglichkeiten und die Bedienung von Mur ϕ ausreichend erklärt. Nachfolgend sind die initialen Datenstrukturen angegeben:

```
const
```

```

    NumInitiators:1;
type
  InitiatorId: scalarset(NumInitiators)
  InitiatorStates: enum{I_SLEEP,I_WAIT,I_COMMIT};
  Initiator:record
    state:initiatorStates;
    responder:AgentId;
  end;
var
  ini:array[InitiatorId] of Initiator;

```

Die Anzahl der Initiatoren ist skalierbar und durch die Konstante `NumInitiators` definiert. Der Typ `InitiatorId` ist eine Untermenge von $1 \dots \text{NumInitiators}$ und sorgt dafür, dass das Programm automatisch versucht, die Anzahl der Zustände zu reduzieren. Der Zustand *jedes* Initiators ist im Array `ini` gespeichert. Der lokale Zustand eines Initiators ist in `state` gespeichert und zu Beginn auf `I_SLEEP` gesetzt, was bedeutet, dass noch kein Initiator das Protokoll gestartet hat.

Das Verhalten eines Initiators ist durch zwei $\text{Mur}\varphi$ -Regeln festgelegt. In der ersten Regel startet ein Initiator das Protokoll, indem er eine initiale Nachricht an einen anderen Agenten schickt und damit seinen Zustand von `I_SLEEP` auf `I_WAIT` setzt. Die zweite Regel beschreibt den Empfang und die Prüfung einer empfangenen Antwort, die Bestätigung und die abschließende Nachricht.

Die erste Regel sieht in $\text{Mur}\varphi$ wie folgt aus:

```

ruleset i: InitiatorId do
  ruleset j: AgentId do
    rule "initiator starts protocol"
      ini[i].state = I_SLEEP & -- condition
      !ismember(j,InitiatorId) &
      multisetcount (l:net, true) ! NetworkSize
    ==>
    var
      outM: Message; -- outgoing message
    begin -- action
      undefine outM;
      outM.source := i;
      outM.dest := j;
      ... set remaining fields of outM
      multisetadd (outM,net);
      ini[i].state := I_WAIT;
      ini[i].responder := j;
    end;
  end;
end;

```

```

    end;
end;

```

Die Bedingung in dieser Regel ist, dass ein Initiator i im Zustand I_SLEEP ist, der Teilnehmer j kein Initiator ist (und damit entweder ein Responder oder Angreifer) und dass es noch Platz im Netzwerk für eine weitere Nachricht gibt. Das Netzwerk ist dazu in der Variable `net` modelliert. Jedes Netzwerk kann höchstens eine Nachricht enthalten.

Die Aktion der Regel ist, dass die ausgehende Nachricht erzeugt und in das Netzwerk gestellt wird. Zudem wird der lokale Zustand verändert und die Identität des Teilnehmers, mit dem eine Kommunikation aufgebaut werden soll, in `responder` gespeichert.

Die zweite der initialen Regeln sieht so aus:

```

ruleset i: InitiatorId do
  choose j: net do
    rule "initiator reacts to nonce received"
      ini[i].state = I.WAIT & -- condition
      net[j].dest = i
    ==>
    var
      inM, outM: Message;
    begin -- action
      inM := net[j];
      multisetremove (j,net);
      if inM.key=i &
        inM.mType=M.NonceNonce &
        inM.noncel=i then
        ... set fields of outgoing message out
        multisetadd (outM,net);
        ini[i].state := I.COMMIT;
      end;
    end;
  end;
end;

```

Die Bedingung dieser Regel ist, dass der Initiator i im lokalen Zustand I_WAIT ist und dass das Ziel der aktuellen Nachricht im Netzwerk `net[j]` auch wirklich der Initiator ist. Die Aktion der Regel entfernt zuerst die Nachricht aus dem Netzwerk. Dann prüft der Initiator,

- ob er die Nachricht entschlüsseln kann,
- ob die Nachricht korrekt ist und

- ob der korrekte Authentifikationsschlüssel enthalten ist.

Sind alle Bedingungen erfüllt, so wird eine ausgehende Nachricht erzeugt und ins Netzwerk gestellt. Der Initiator bestätigt die Sitzung, indem er seinen lokalen Zustand auf `I_COMMIT` setzt.

Nun soll noch gezeigt werden, wie Konstanten angegeben werden, die ein korrektes System spezifizieren. Die dargestellte Konstante gibt an, dass der Initiator und der Responder korrekt authentifiziert sein müssen.

```
invariant "responder correctly authenticated"
forall i: InitiatorId do
  ini[i].state = I.COMMIT &
  ismember(ini[i].responder, ResponderId)
->
  res[ini[i].responder].initiator = i &
  ( res[ini[i].responder].state = R.WAIT |
  res[ini[i].responder].state = R.COMMIT )
end;
```

Diese Konstante überwacht das Verhalten des Initiators `i` und wird dann aktiv, wenn der Initiator eine Sitzung *bestätigt*. Die Konstante kontrolliert dann, ob der Responder (gespeichert in `ini[i].responder`) auch wirklich das Protokoll mit `i` gestartet hat. Dazu muss der Responder sowohl den Initiator `i` in seinem Feld `initiator` stehen haben, als auch im Zustand `R_WAIT` oder `R_COMMIT` sein. Ist diese Bedingung nicht mehr erfüllt, so hat der Initiator eine Sitzung zu einem Teilnehmer *bestätigt*, der nicht der ist, den er vorgibt zu sein. Damit ist ein Fehler im Protokoll gefunden.

1.9 Schlussbemerkung

In der Vergangenheit wurden bereits zahlreiche kryptographische Protokolle mithilfe der endlichen Zustandsautomaten verifiziert, wodurch zahlreiche Protokollfehler zum Vorschein kamen — teilweise sogar in 17 Jahre alten Protokollen. Dies zeigt die hohe Güte dieses Verfahrens, das sogar weitestgehend automatisch verifiziert. Für die Modellierung der Protokolle gibt es bereits Werkzeuge, wie `Murφ`, die speziell für nicht-deterministische Automaten konzipiert sind.

Allerdings leidet dieses Analyseverfahren an der Zustandsexplosion, d.h., dass mit zunehmender Anzahl von Zuständen die Komplexität exponentiell ansteigt. Hierfür bedarf es Verfahren, um die Anzahl der Zustände zu reduzieren, ohne dabei Funktionalität zu verlieren ([shm98]): Es sollen nach der Reduzierung alle Fehler gefunden werden, die auch zuvor gefunden worden wären.

Auch besteht das Problem, wie der Angreifer zu modellieren ist. Wie etwa muss die Wissensbasis aufgebaut sein, kann der Angreifer beispielsweise verschlüsselte Nachrichten einfach ungelesen für einen Angriff nutzen?

Hauptproblem bei der Verifizierung kryptographischer Protokolle ist jedoch der Faktor Mensch. So können trotz Analyseverfahren und Analysewerkzeugen Fehler verborgen bleiben, wenn der Operator zu wenig Wissen über kryptographische Verfahren besitzt.

Literaturverzeichnis

- [hel98] Heljanko, H.: Can Finite-State System Verification Methods Help Cryptographic Protocol Analysis?, Helsinki University of Technology, Helsinki, 1998.
- [lowe96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Margaria and Steffen, editors, Tools and Algorithms for the Construction and Analysis of Systems, number 1055 in Lecture Notes in Computer Science, pages 147-166. Springer-Verlag, 1996.
- [den81] Dennings, D. and Sacco, G. Timestamps in Key Distribution Protocols. Communications of the ACM, 24(8):198-208, 1981
- [shm98] Shmatikov, V. and Stern, U. Efficient Finite-State Analysis for Large Security Protocols, Stanford University, IEEE Computer Society Press, 1998
- [mitc97] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using MurOE. In Proceedings of the 1997 IEEE Symposium on Security and Privacy, pages 141-151. IEEE Computer Society Press, May 1997.
- [lowe96ns] G. Lowe, Breaking and fixing the Needham-Schroeder public-key protocol using FDR, In Tools and Algorithms for the Construction and Analysis of Systems, volume 1055 of Lecture Notes in Computer Science, pages 147-166, Springer-Verlag, 1996.
- [ma00fo] "Lu Ma and Jeffrey J. P. Tsai, Verification Techniques For Computer Communication Security Protocols, University of Illinois at Chicago, 2000.