Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Verificação Formal de Protocolos Criptográficos – O Caso dos Protocolos em Cascata

Rodrigo Borges Nogueira

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre no Curso de Pós-graduação em Informática

Orientador
Maurício Ayala Rincón

Brasília
2008

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Li Weigang

Banca examinadora composta por:

Prof. Dr. Maurício Ayala Rincón (Orientador) – MAT/UnB

Prof. Dr. Anderson Clayton Alves Nascimento – ENE/UnB

Prof. Dr. Flávio Leonardo Cavalcanti de Moura – CIC/UnB

CIP — Catalogação Internacional na Publicação

Nogueira, Rodrigo Borges.

Verificação Formal de Protocolos Criptográficos – O Caso dos Protocolos em Cascata / Rodrigo Borges Nogueira. Brasília: UnB, 2008. 90 p. : il. ; 29,5 cm.

Dissertação de Mestrado – Universidade de Brasília, Brasília, 2008.

1. Verificação Formal, 2. Protocolos Criptográficos,
3. Sistemas de provas automáticas, 4. PVS

CDU 004.4

Endereço: Universidade de Brasília

Campus Universitário Darcy Ribeiro – Asa Norte

CEP 70910-900

Brasília–DF – Brasil

Resumo

Garantir a segurança de protocolos criptográficos não é uma tarefa simples. O *modelo Dolev-Yao*, proposto no início da década de 80, constitui uma importante metodologia matemática de modelagem de protocolos criptográficos, possibilitando a análise e verificação da segurança destes protocolos. Porém, a verificação analítica não garante isenção de erros nas implementações. Com efeito, existem vários exemplos de protocolos que foram considerados matematicamente seguros, mas com falhas descobertas até mesmo após mais de uma década de quando propostos. Dessa forma, as abordagens baseadas em métodos formais são de grande utilidade para garantir, efetivamente, a segurança de implementações destes protocolos. Neste trabalho, utilizamos um sistema de especificação e prova, o *PVS* (*Prototype Verification System*), para especificar e verificar mecanicamente a segurança de uma classe de protocolos no modelo Dolev-Yao: os *Protocolos em Cascata* para dois usuários. Fazendo isto, detectaram-se falhas definicionais na especificação matemática desta classe de protocolos e formularam-se noções e lemas mais gerais que os inicialmente propostos.

Palavras-chave: Verificação formal, protocolos criptográficos, PVS

Abstract

Ensuring the security of cryptographic protocols is not a simple task. The *Dolev-Yao model*, proposed in the early 80s, is an important mathematical modeling method of cryptographic protocols, enabling the analysis and verification of the safety of these protocols. But the analytical verification does not guarantee that implementations are not error-prone. Indeed, we have, in the history of protocols, a lot of examples with flaws detected even after a decade of its introduction. Thus, formal method approaches are very useful to ensure the security of protocol implementations. In this work, we use the *PVS* (*Prototype Verification System*) to specify and mechanically verify the safety of a protocol class in the Dolev-Yao model: the two-party cascade protocols. By verifying our specification, we have detected a few flaws in the original definitions and we have proposed more general concepts and lemmas.

Keywords: Formal verification, cryptographic protocols, PVS

SUMÁRIO

Lista de Figuras

Lista de Anexos

Capítulo 1.

Introdução

Protocolos criptográficos ou protocolos de segurança são pequenas aplicações distribuídas que garantem propriedades de segurança em um ambiente hostil [61]. Autenticidade e confidencialidade das mensagens trocadas entre os participantes de um protocolo são exemplos de requisitos esperados em uma comunicação. Vários serviços e tecnologias são viabilizados pela utilização de protocolos criptográficos como, por exemplo, transações bancárias on-line, comércio eletrônico, sistemas eleitorais eletrônicos e comunicações militares.

O principal problema do projeto de um protocolo criptográfico está na forma como o conceito de *segurança* é definido e atribuído a este protocolo, principalmente porque um protocolo criptográfico, geralmente, agrega diversas outras ferramentas da criptografia, como as funções de hash, algoritmos de encriptação, etc, e cada uma dessas ferramentas pode definir a segurança de um modo específico. Formalmente, a segurança de protocolos criptográficos possui diferentes níveis de definições, que variam de acordo com o grau de abstração de determinadas diretivas de segurança. Assim, para se dizer que um protocolo criptográfico é seguro, é necessário dizer também sob que hipóteses a segurança é definida. Em protocolos criptográficos, a segurança, basicamente, diz respeito ao provimento de garantias [9], como:

- *Autenticação*: garante que um participante A esteja se comunicando com B, quando A acredita estar se comunicando com B.
- *Sigilo*: garante que mensagens enviadas sejam inteligíveis somente aos receptores para quem as mensagens são destinadas.
- *Não-repúdio*: Nenhum dos participantes de um protocolo pode negar a participação no protocolo após o fim da sessão.

- *Não-interferência*: informações secretas não devem vazar para intrusos durante a transmissão. Para prover isso, o sistema deve considerar todo nível de interação com agentes quaisquer.
- *Atomicidade*: todos os passos de uma transação devem ser concluídos completamente, ou nenhum passo é concluído.
- *Imparcialidade/Justiça (Fairness)*: não é admitido a um participante obter qualquer tipo de vantagem sobre outro(s) participante(s) durante a execução do protocolo.

O desenvolvimento de protocolos criptográficos, no início das pesquisas em segurança de dados, realizou-se de modo simples, iterativamente: proposta do protocolo, descoberta de algum tipo de ataque, reestruturação do protocolo até que nenhum outro ataque conhecido fosse descoberto [1]. Seguindo essa metodologia, o projeto de um protocolo criptográfico fica suscetível a graves erros. Um exemplo clássico disso é o protocolo de chaves públicas de Needham-Schroeder [4], proposto em 1978. Durante dezessete anos considerou-se este protocolo seguro, até que Gavin Lowe descobriu uma falha e apresentou uma versão corrigida do protocolo, além de provar que o novo protocolo é correto [5]. Outro exemplo é a descoberta de falhas em um protocolo do projeto do padrão CCITT X.509 [13]. E ainda, recentemente, descobriu-se uma falha no *SET* (*Secure Electronic Transaction*) [19], que é uma suíte de protocolos proposta por um consórcio de companhias de cartões de crédito e corporações de *software* para comércio eletrônico. Descobriu-se ser impossível provar que o SET satisfaz a seguinte garantia esperada: *o proprietário de um cartão de crédito envia os detalhes do seu cartão somente ao gateway de pagamento, mantendo esses detalhes desconhecidos ao vendedor*.

Durante aproximadamente 20 anos, as comunidades de métodos formais e criptografia trabalharam em sentidos opostos [7]. A primeira tinha como principal objetivo expressar a segurança focando em metodologias de provas automáticas eficientes para expressar objetivos de segurança. Na segunda comunidade, a maior parte dos trabalhos se concentrou em definir a segurança de algoritmos e protocolos criptográficos contra os mais expressivos tipos de ataques, restritos a condições de tempo polinomial, provando que a segurança se obtinha baseada, tipicamente, em hipóteses da teoria dos números.

Com o desenvolvimento das técnicas automáticas de verificação formal e a demanda cada vez maior por protocolos criptográficos confiáveis, a conexão entre métodos formais e criptografia tem recebido grande interesse atualmente, a fim de se modelar matematicamente requisitos de segurança de protocolos criptográficos e provar que estes requisitos são atendidos nos modelos. Mais que isso, especificar protocolos criptográficos baseados em operações abstratas, de modo que se possam provar propriedades de segurança que sejam válidas para implementações reais.

O uso de métodos formais na verificação de protocolos criptográficos teve início, principalmente, com o trabalho de Dolev e Yao em [6], que propuseram, basicamente, um modelo que consiste na definição de um adversário com total controle sobre a rede. Este controle é inferido da modelagem, em *álgebra abstrata*, das regras de um protocolo e interação deste com o adversário. É utilizada a *criptografia perfeita*, ou seja, é impossível desfazer, sem a chave privada, uma encriptação ou aprender qualquer informação relativa às mensagens dos usuários honestos do protocolo.

Neste trabalho, mostramos como técnicas de provas disponíveis no assistente de provas *PVS* (*Prototype Verification System*) [8] [63] podem ser aplicadas na análise de segurança de protocolos criptográficos. Utilizando a linguagem e a lógica do PVS, provamos a segurança do *modelo Dolev-Yao* para protocolos em cascata [6]. Para a prova deste modelo, especificamos algumas *teorias* gerais sobre propriedades básicas de protocolos criptográficos e, a partir destas teorias, desenvolvemos nossa modelagem, mostrando, assim, como se podem construir modelos de protocolos criptográficos utilizando sistemas como o PVS. Além disso, identificamos falhas e omissões no formalismo do modelo original proposto por Dolev e Yao e pudemos expressar de modo mais genérico alguns lemas e definições.

Dentre as motivações para uso do PVS em nossa modelagem, estão a robustez do sistema e sua utilização em resultados importantes da literatura e aplicações industriais. Além disso, o PVS possui um sistema de tipos adequado à modelagem de protocolos criptográficos, especialmente por permitir a definição de tipos não interpretados e tipos/subtipos predicados, amplamente utilizados em nossas especificações. Isso possibilita especificar elementos da criptografia de forma flexível e reutilizável. O PVS utiliza o *cálculo de seqüentes* como lógica de demonstração de teoremas e implementa um conjunto

bastante diversificado de estratégias de provas, o que permite o desenvolvimento de provas com níveis de abstração adequados. A seguir, algumas aplicações importantes da utilização do PVS.

- Provas relacionadas a algoritmos de diagnóstico para arquiteturas de tolerância a falhas bizantinas [3].
- Verificação de micro-código para um conjunto de instruções do microprocessador *pipelined* Collins AAMP5 avionics, que contém 500.000 transistores. Especificaram-se formalmente, em PVS, a micro arquitetura do microprocessador e 108 das 209 instruções, sendo que 11 instruções importantes foram formalmente verificadas com a descoberta de erros [3].
- Verificação formal de protocolos criptográficos, onde podemos citar, por exemplo, os trabalhos em [54] e [55], em que se verificam protocolos de autenticação recursiva; e os trabalhos de Backes em [1], [65] e [66], onde uma biblioteca criptográfica foi criada para utilização em provas automáticas; ou ainda [56], onde se realizou a verificação do protocolo *Enclaves*.

A fim de ilustrar a importância do estudo e verificação formal da segurança de protocolos criptográficos, apresentamos no **Anexo 2** o protocolo Needham-Schroeder de autenticação [4], e como este protocolo pode ser subvertido.

Este documento está organizado da seguinte forma: No capítulo **Capítulo** 2, apresentamos um *survey* mostrando diferentes metodologias aplicadas à análise de protocolos criptográficos. O capítulo **Capítulo** 3 mostra os fundamentos do assistente de provas PVS e a lógica implementada por este sistema. O capítulo **Capítulo** 4 descreve, em detalhes, nossa modelagem analítica para o modelo Dolev-Yao de protocolos em cascata. No capítulo **Capítulo** 5, mostraremos definições e provas realizadas em PVS, correspondentes à modelagem analítica do capítulo **Capítulo** 4. E, no capítulo **Capítulo** 6, concluímos e apresentamos alguns trabalhos futuros.

Capítulo 2.

Métodos Formais e Criptografia

Em [9], [13] e [21] encontramos quatro abordagens para análise formal de protocolos criptográficos.

1. Modelagem e verificação de protocolos criptográficos utilizando métodos e ferramentas não especificamente desenvolvidos para segurança.
2. Desenvolvimento de sistemas especialistas para avaliação de possibilidades em diferentes cenários.
3. Modelagem e verificação de protocolos criptográficos utilizando lógicas baseadas em conhecimento e confiança.
4. Desenvolvimento de modelos formais como propriedades da álgebra de termos.

A seguir, apresentamos a idéia geral de cada um dos paradigmas acima, mostrando casos em que são aplicados.

2.1 Verificação por ferramentas de propósito geral

Diferentes ferramentas já se utilizaram para verificação formal de protocolos criptográficos como, por exemplo: Redes de Petri [24], CSP (*Communicating Sequential Processes*) [22] [23], FDR (*Failures-Divergence Refinement*) [5], máquinas de estados finitos [25] [26] e análise via complexidade computacional [29].

Em [22], utilizou-se CSP para analisar o protocolo de Woo-Lam, proposto em 1992 [27]. CSP é uma linguagem em que um sistema é modelado em termos de processos e eventos, onde os processos podem mudar de estado na ocorrência de eventos, que são atômicos.

Usando FDR em [5], Lowe, após a descoberta do ataque *man-in-the-middle* ao protocolo de autenticação Needham-Schroeder, propôs uma correção e provou a segurança deste protocolo modificado (com nome *Needham-Schroeder-Lowe*) no *modelo Dolev-Yao*.

O FDR é uma ferramenta de verificação de modelos para máquinas de estados, com fundamentos na teoria da concorrência baseada em CSP.

Nas duas seções a seguir, apresentamos exemplos mais detalhados de utilização de ferramentas de propósito geral na verificação de protocolos criptográficos.

2.1.1 Verificação de protocolos por máquinas de estados finitos

A **Figura 1** a seguir, extraída de [9], modela dois participantes *A* e *B* durante a execução de um protocolo. Os arcos U^{-n} e U^{+n}, respectivamente, significam que um participante *U* envia e recebe uma mensagem *n* e, para cada mensagem enviada ou recebida, há uma mudança de estado. A máquina de estados finitos *M*, na **Figura 1**, é construída pelo produto cartesiano das máquinas individuais M_A e M_B de dois participantes *A* e *B*, respectivamente. Durante a execução do protocolo, um estado S_{ij} de *M* indica que *A* está em um estado S_i de M_A e *B* em um estado S_j de M_B. Se o número de estados em M_A e M_B é *m*, então o produto cartesiano destas duas máquinas resulta na máquina *M* com m^2+1 estados, incluindo o estado final.

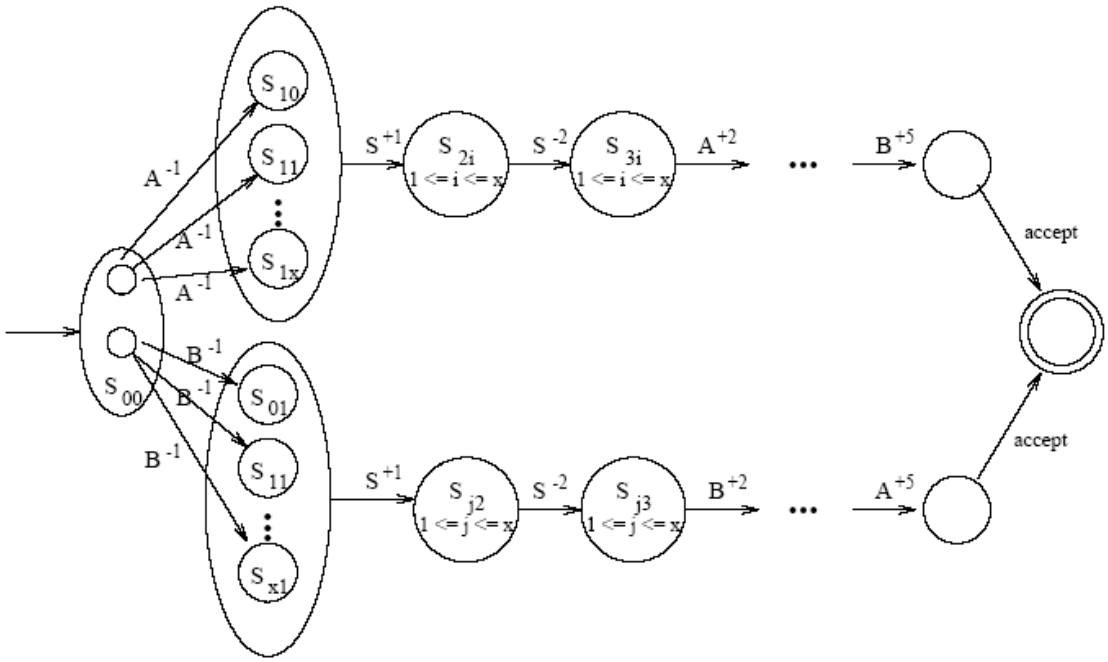

Figura 1: Máquina de estados finitos não-determinística modelando dois participantes *A* e *B* durante a execução de um protocolo.

Para estudo do protocolo modelado por *M*, utiliza-se a técnica da análise de *alcançabilidade* [28], onde, para cada transição, o estado global do sistema em *M* é analisado considerando-se os participantes do protocolo e os canais de comunicação. Por exemplo, se, em um estado global de *M*, um participante *U* não pode receber uma mensagem que era esperada, então há um problema no protocolo.

Com esta técnica podem-se identificar problemas relativos à correção do protocolo quanto a sua especificação, mas não se garante a segurança contra um adversário ativo.

2.1.2 Protocolos criptográficos e complexidade computacional

A teoria da *NP-completude* é parte de uma teoria abrangente chamada *complexidade computacional* [31]. A idéia básica dessa teoria é caracterizar uma classe de problemas para os quais existe um algoritmo *não-determinístico*. Em um problema *não-determinístico* as soluções possíveis estão localizadas em um espaço de busca exponencial, mas podem ser verificadas em tempo polinomial. Todo problema *NP-completo* pode ser reduzido, polinomialmente, a outro problema NP-completo. Esta é a base das provas de *NP-completude*, ou seja, encontrar um problema NP-completo ou *NP-difícil* conhecido que pode ser reduzido ao problema avaliado, além de se provar que este problema pertence à *classe de problemas NP*.

Em [29], mostra-se que a complexidade do *problema da insegurança de um protocolo* para um número finito de sessões é NP-completo, considerando-se o modelo Dolev-Yao de adversários. Na prova, o problema *3-SAT* foi reduzido ao problema em questão, e a idéia da redução é deixar o adversário gerar a primeira mensagem do protocolo o que é representado por uma solução para o problema 3-SAT. O teorema principal provado em [29] diz que "*Descobrir um ataque a um protocolo com um número fixo de sessões é um problema NP-completo*". O problema da insegurança de um protocolo consiste em determinar se existe um cenário de processos em execução, no qual há um adversário tentando definir um ataque potencial.

Trabalhos recentes e importantes de verificação de propriedades de protocolos criptográficos, via complexidade computacional, são [32], [33] e [34].

2.2 Verificação por sistemas especialistas

Com sistemas especialistas, a verificação de protocolos criptográficos se beneficia da geração arbitrária de várias ações possíveis e condução de ataques, e o sistema verifica a integridade operacional do protocolo para todo estado de ataque. Diferentemente das máquinas de estados finitos mostradas na seção **2.1.1**, os sistemas especialistas iniciam em um estado indesejável e tentam descobrir se este estado é alcançável a partir do estado inicial ou vice-versa.

Este tipo de sistema é desenvolvido especificamente para a análise de protocolos criptográficos e alguns apresentam a limitação de que a construção de um ataque vincula-se à limitação humana de conhecimento dos ataques. Geralmente, existe um mecanismo de busca exaustiva com necessidade de interferência do usuário, a fim de se enriquecer a quantidade de estados inseguros e caminhos que levam a uma falha.

2.2.1 Interrogator [20]

Sistema especialista baseado em máquinas de estados finitos, útil para análise de protocolos sob ataques conhecidos, em que um adversário pode destruir, interceptar e modificar todas as mensagens. O protocolo é visto como uma coleção de processos que se comunicam, sendo um processo para cada participante. Um processo possui um conjunto de estados e a transmissão de uma mensagem pode provocar uma transição de um estado a outro.

A entrada para o sistema é uma especificação de protocolo e um dado alvo que pode ser uma mensagem, por exemplo. A saída é um histórico de mensagens mostrando como o adversário pôde interagir com o dado alvo.

No Interrogator, um grande número de caminhos possíveis para execução dos processos é gerado. Se um caminho, a partir de um estado inicial, termina em um estado inseguro, então se pode dizer que há uma falha no protocolo.

2.2.2 CryptoVerif

Blanchet, em [12], afirma que existem dois principais *frameworks* para o estudo de protocolos criptográficos: O *modelo computacional*, onde mensagens são *bitstrings* e o adversário é uma máquina de Turing probabilística de tempo polinomial. Neste modelo, que mais se aproxima do modelo real de execução dos protocolos, as provas são geralmente realizadas de forma manual e informal. O *modelo Dolev-Yao*, onde as primitivas criptográficas são definidas como caixas pretas perfeitas através de abstrações da álgebra de termos. Neste modelo, pode-se mais facilmente construir provas automáticas, mas as provas de segurança geralmente não são robustas como no modelo computacional.

Blanchet desenvolveu um *process calculus* inspirado, principalmente, pelo *pi-calculus*, e sobre este fundamento construiu um provador automático de protocolos no modelo computacional. As provas seguem uma metodologia iterativa de *games* [35], onde o game inicial representa o protocolo a ser provado. O objetivo é mostrar que a probabilidade de se quebrar um requisito de segurança é insignificante em um game. A partir de um game, outro subseqüente é obtido através de transformações tais que a diferença de probabilidade entre games consecutivos é desprezível. No game final a probabilidade de se quebrar o requisito de segurança é, pela forma do game, claramente desprezível.

O *process calculus* criado possui uma semântica probabilística e todos os processos rodam em tempo polinomial. A principal ferramenta para especificar propriedades de segurança é a equivalência observacional: um processo Q é observacionalmente equivalente a Q', $Q \approx Q'$, quando um adversário tem probabilidade desprezível de distinguir Q de Q'.

Para se conseguir transformar o protocolo inicial em um game cuja propriedade de segurança desejada é óbvia, o mais importante tipo de transformação vem da definição de segurança das primitivas criptográficas. Esta transformação pode ser especificada de um modo genérico: a definição de segurança de uma primitiva criptográfica é realizada pela equivalência observacional $L \approx R$, onde L e R são processos que codificam funções, ou seja, são processos que recebem argumentos de funções e devolvem seu resultado. Então, o provador pode, automaticamente, transformar um processo Q, que chama as funções de L, em um processo Q', que chama as funções de R. Esta técnica tem sido usada para especificar encriptações de chaves públicas e/ou compartilhadas, assinaturas, *MAC*'s (*Message Authentication Codes*) e funções de hash, passando para o provador as

equivalências $L \approx R$ apropriadas. Outras transformações de games são sintáticas, utilizadas para permitir a aplicação de definições das primitivas criptográficas, ou para simplificar o game obtido após se aplicar estas definições.

Além dos dois sistemas especialistas citados, outros foram propostos, como o *NRL Protocol Analyzer* por Meadows [36] e o *Longley and Rigby* [37].

2.3 Verificação por lógicas de análise de conhecimento e confiança

Nesta abordagem da verificação formal de protocolos criptográficos estão as lógicas modais, que consistem de uma linguagem que descreve várias declarações sobre confiança, conhecimento de mensagens em um sistema distribuído e algumas regras de inferência usadas para derivação de declarações a partir de outras declarações. O objetivo é derivar declarações que representem condições corretas de um protocolo.

2.3.1 Lógica BAN

A lógica BAN é a lógica mais amplamente utilizada na análise de protocolos de autenticação. Foram publicados diversos artigos mostrando a utilização da lógica BAN para provas de segurança de protocolos. Exemplos importantes são [39] e [40].

Alguns termos básicos da lógica BAN:

- $P \lhd \{X\}_K$: Conhecendo-se a chave K, indica que "*P* reconhece *X*", ou seja, *P* recebe uma mensagem *X* da qual *P* consegue obter significado.
- $P \ni K^{-1}$: Indica possessão de uma chave privada K^{-1} por *P*.
- $\mathrm{sec}ret(X,P,Q)$: Significa que existe um segredo *X* (chave ou *nonce*) compartilhado entre os participantes *P* e *Q*.
- $P \mid\equiv \$$: Indica que *P* é levado a acreditar que $ é válido.
- $P \mid\sim \$$: Descreve que *P*, alguma vez, declarou $, ou agora ou no passado.
- $fresh(X)$: É um predicado que afirma ser *X* gerado no presente.
- $P \mid -\$$: Significa que *P* proferiu $ no passo corrente do protocolo.

Exemplos de regras de inferência BAN [11]:

$$(Dec)\ \frac{P \lhd \{X\}_K,\ P \ni K^{-1}}{P \lhd X} \qquad (SS)\ \frac{P \lhd X,Y,\ P \mid\equiv \sec ret(X,P,Q)}{P \mid\equiv Q \mid\sim X,Y}$$

$$(Say)\ \frac{P \mid\equiv Q \mid\sim X,\ P \mid\equiv fresh(X)}{P \mid\equiv Q \mid -X}$$

- *Regra de decriptação (Dec)*: Se P possui a chave K^{-1} e recebeu uma mensagem X encriptada por uma chave K, então P recebeu X.
- *Regra de segredo compartilhado* (*SS*): Se P reconhece a mensagem composta por X e Y, e P acredita que X é um segredo compartilhado entre P e Q, então P acredita que Q enviou, ou agora ou no passado, a mensagem X,Y.
- *Regra de proclamação* (*Say*) Se P acredita que Q, alguma vez, declarou X e P acredita que X é uma mensagem gerada no presente, então P acredita que Q proferiu X no passo corrente do protocolo.

Como se pode perceber, com a lógica BAN é possível lidar com protocolos somente em um nível abstrato.

Segundo os criadores da lógica BAN [38], para análise de protocolos devem-se seguir os seguintes passos: 1) O protocolo idealizado é derivado do protocolo original; 2) Especificam-se as suposições sobre o estado inicial; 3) Às declarações do protocolo, anexam-se fórmulas lógicas como assertivas sobre o estado do sistema após cada declaração; 4) Aplicam-se postulados lógicos às suposições e assertivas para verificar se os critérios de confiança pelos participantes do protocolo são válidos.

A lógica BAN foi projetada de modo a permitir que fosse estendida, ou seja, novas construções e postulados podem ser adicionados, adequando-se a uma aplicação particular. Algumas extensões foram criadas, como por exemplo, a lógica GNY em [41].

Diversas outras lógicas foram criadas para expressar propriedades de protocolos criptográficos. Sucintamente, apresentaremos dois exemplos:

- Moser, em [42], propôs uma *lógica não-monotônica da confiança*, onde ele afirma que, no mundo real, a confiança no sigilo de uma chave de sessão pode mudar caso esta chave seja comprometida, por exemplo. Nesta lógica, conhecimento e confiança podem ser não-monotônicas.

- Mais recentemente, em [43], Vigano e equipe desenvolveram uma lógica para formalização de ataques, com o desenvolvimento de novas semânticas que combinam características de lógicas já existentes e métodos indutivos. O método contempla o problema da modelagem de protocolos com execuções em intervalos disjuntos, o que não era realizado anteriormente pelas lógicas de segurança baseadas em lógica modal. Vigano mostrou com esta lógica o ataque *man-in-the-middle* ao protocolo Needham-Schroeder.

2.4 Verificação pela álgebra de termos

Parte dos trabalhos atuais de verificação formal de protocolos criptográficos está baseada na utilização de sistemas algébricos. Nesta abordagem, as primitivas criptográficas são geralmente idealizadas como tipos com certas propriedades e o protocolo é considerado como um sistema que utiliza as propriedades destes tipos para prover as garantias requeridas. Muitas provas de segurança são, portanto, realizadas provando-se matematicamente que é impossível que um adversário subverta os objetivos do protocolo.

Paulson, em [45], utilizou o sistema de provas *Isabelle* para construir provas indutivas de corretude de protocolos criptográficos. E Millen, em [46], utilizou o sistema *PVS* para obter provas baseadas nas idéias de Paulson. Thayer e equipe, em [47], criaram um modelo teórico de causalidade em um protocolo, chamado *strand spaces*. E, Dolev e Yao, em [6], propuseram um modelo algébrico de segurança de protocolos criptográficos. Em particular, introduziram uma classe de *protocolos em cascata*, de chaves públicas, nos quais um participante pode aplicar operações de encriptação ou decriptação em várias camadas para formar mensagens. Abordaremos o modelo Dolev-Yao em detalhes nos capítulos **Capítulo** 4 e **Capítulo** 5.

2.5 Verificação por outros paradigmas

Além dos quatro paradigmas clássicos apresentados anteriormente, existem outras metodologias mais recentes de verificação de protocolos criptográficos, descritas a seguir.

2.5.1 Ferramenta Híbrida na verificação de ataques multi-protocolos

Protocolos criptográficos, em sua maioria, são projetados para uma troca segura de chaves de sessão após um processo de autenticação [14]. Um dos problemas mais comuns em segurança, entretanto, é que as vulnerabilidades aparecem na fronteira entre duas tecnologias de proteção [16]. Neste sentido, a verificação formal de *ataques multi-protocolos* é importante, pois, tipicamente, a modelagem formal e verificação de segurança são realizadas considerando-se que um protocolo é executado isoladamente, sem compartilhamento da rede com outros protocolos [17].

De fato, considerar que apenas um protocolo executa em uma rede não-confiável não é realístico. E, dois protocolos isolados considerados corretos, podem apresentar falhas quando utilizados sob a mesma rede. O principal resultado sobre este tipo de ataque aparece em [18]. Em [17], descobriu-se que, de 30 protocolos da literatura, 23 são vulneráveis a ataques multi-protocolos. O total de ataques descobertos foi 163. Os testes foram realizados considerando protocolos criptográficos executando, em paralelo, dois a dois e três a três, utilizando a ferramenta *Scyther* [44], que, dada uma descrição de um conjunto de protocolos, tenta construir um contra-exemplo (ataque). Esta ferramenta utiliza um algoritmo híbrido de *provas de teoremas* e *verificação de modelos*.

Com o Scyther, 17 ataques dois a dois foram observados modelando-se os protocolos de modo que os participantes podem, de alguma forma, verificar os tipos dos dados que recebem e, então, aceitar somente termos de tipos corretos. Este é o modelo mais restritivo dos três modelos avaliados em [17]. A verificação de tipos é possível até mesmo para um conjunto *Nonce* de todos os valores randômicos. Os ataques descobertos são de violação da autenticação entre participantes e falhas em requisitos de sigilo. A principal causa dos ataques é a forma como os *mecanismos de desafios* para autenticação são construídos. Um participante *X* prova sua identidade a outro participante pela aplicação de uma chave que somente *X* conhece. Este procedimento de se aplicar uma chave pode ser para encriptação ou decriptação, e estes dois métodos, por serem complementares, tornam possível que a encriptação em um protocolo seja seguida por uma decriptação no outro protocolo em paralelo. Isso gera a quebra do segredo de valores randômicos e, conseqüentemente, a quebra do processo de autenticação.

2.5.2 Segurança baseada em simulação

Atualmente, vários trabalhos têm utilizado idéias como a *indistinguibilidade* e a *funcionalidade ideal* para provar propriedades de segurança de protocolos criptográficos. Os principais projetos são de Canetti com sua abordagem da *Universal Composability* [48] [49] e de Backes, Pfitzmann, e Waidner, que produziram trabalhos relacionados com o de Canetti, mas usando também a *Simulabilidade Caixa-Preta* [1] [7] [50].

2.5.2.1 Universal Composability

Uma das principais razões para a existência de falhas em protocolos criptográficos é a ocorrência de interações inexperadas entre instâncias de diferentes protocolos que executam paralelamente em sistemas compostos [52]. A *composição* visa garantir a obtenção de um protocolo seguro pela composição de protocolos seguros.

Em [48], Canetti introduz a *Universal Composability* (*UC*), onde o modelo de execução de um protocolo consiste de um conjunto de máquinas de Turing Interativas (*ITMs - Interactive Turing Machines*), representando os participantes do protocolo e mais uma *ITM* para representar o adversário. A *UC* envolve a comparação entre um protocolo real e uma funcionalidade ideal, a comparação entre um adversário real e um ideal e um ambiente. O protocolo real realiza a funcionalidade ideal se, para todo ataque de um adversário real ao protocolo real, existe um ataque de um adversário ideal à funcionalidade ideal, tal que o comportamento observável do protocolo real sob ataque é o mesmo comportamento da funcionalidade ideal sob ataque. Cada conjunto de observações do sistema é obtido sobre mesmo ambiente. Em outras palavras, o sistema consistindo do ambiente, o adversário real e o protocolo real devem ser *indistinguíveis* do sistema consistindo do ambiente, o adversário ideal, e a funcionalidade ideal.

Em [52], Canetti descreve o *UC security framework* como uma estrutura para representação e análise de protocolos criptográficos. Neste paradigma, formulou-se uma metodologia geral e uniforme para expressar requisitos de segurança de praticamente qualquer ferramenta criptográfica. Além disso, define-se um método geral para composição de protocolos, mostrando que as definições de segurança geradas neste *framework* preservam a segurança sob as operações de composição.

2.5.2.2 Simulabilidade Caixa-Preta

O objetivo principal desta abordagem, segundo Pfitzmann e Waidner em [51], é ligar dois diferentes paradigmas de verificação de protocolos criptográficos: o de provas matematicamente rigorosas e o de provas mecanizadas por algum sistema de provas formais, de modo que se possam obter provas que permitam abstrações e a utilização de métodos formais, mantendo as semânticas criptográficas robustas.

Em [51], Pfitzmann e Waidner utilizam uma variante da *UC* e a noção de *Simulabilidade Caixa-Preta* baseada em *PIOA* (*Probabilistic Polynomial-Time IO Automata*). O modelo segue a idéia da *Simulabilidade*, onde a segurança de um sistema é definida relativamente a outro sistema. Com base na Simulabilidade e outras definições auxiliares, chega-se à definição de *Simulabilidade Caixa-Preta* e *Simulabilidade Universal*, que são a base do modelo de segurança proposto em [51] para sistemas de segurança reativos em redes assíncronas.

A *Simulabilidade* (com omissão de detalhes) é definida para dois sistemas Sis_1 e Sis_2 quando:

$$Sis_1 \geq_{\text{sec}}^{f,perf} Sis_2$$

Isso significa que Sis_1 é perfeitamente pelo menos tão seguro quanto Sis_2 se existe um mapeamento f válido de Sis_1 para Sis_2 e, para toda configuração de Sis_1, existe uma configuração de Sis_2 tal que um usuário H tenha uma visão indistinguível do sistema ao interagir com a configuração de Sis_1 ou de Sis_2.

O sistema Sis_1 é freqüentemente um sistema real que utiliza primitivas criptográficas concretas, enquanto Sis_2 é o sistema ideal, isto é, uma especificação que não depende de qualquer detalhe de implementação criptográfica específica e é não realística.

Capítulo 3.

Fundamentos do PVS

O PVS [3] é um sistema que permite a especificação e verificação formal de modelos, baseada em um sistema de tipos robusto com uma lógica de ordem superior [2]. Basicamente, pode ser visto como composto de três partes: uma linguagem de especificação, um provador de teoremas interativo e uma livraria de teorias já provadas, o *prelude*, contendo várias definições e lemas utilitários.

3.1 A lógica implementada no PVS [8] [62] [63]

O PVS utiliza o *cálculo de seqüentes* na manipulação e prova de lemas e teoremas. Um seqüente é uma expressão da forma $\Gamma \vdash \Delta$, onde Γ e Δ são seqüências de fórmulas, chamadas de *antecedente* e *conseqüente*, respectivamente. O símbolo $\vdash$ denota que, assumindo Γ, Δ é *dedutível* ou *provável*.

De modo usual, utilizam-se as letras gregas Γ e Δ para representar seqüências de fórmulas e letras latinas A, B e C para representar fórmulas individuais. Em um seqüente $\Gamma \vdash \Delta$, as fórmulas A_i de Γ e as fórmulas B_k de Δ são representadas como $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$, para algum n e m, com $0 < i \leq n$ e $0 < k \leq m$. Entretanto, em cada lado de $\vdash$, as fórmulas são interpretadas de modo diferente, isto é, Γ é uma *conjunção* de fórmulas A_i e Δ é uma *disjunção* de fórmulas B_k. Assim, $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$ é interpretado como $A_1 \wedge A_2 \wedge \ldots \wedge A_n \vdash B_1 \vee B_2 \vee \ldots \vee B_m$.

No cálculo de seqüentes, premissas e conclusões são construídas simultaneamente através de um conjunto de *regras de inferência* (possuem a forma $\frac{premissa}{conclusão}$ $\boldsymbol{nome\ da\ regra}$) básicas, utilizado para derivar ou deduzir seqüentes a partir de outros seqüentes. Intuitivamente, uma regra de inferência da forma $\frac{\Gamma \vdash \Delta}{\Gamma_1 \vdash \Delta_1}$ significa: *Se a premissa* $\Gamma \vdash \Delta$ *é verdadeira, então a conclusão* $\Gamma_1 \vdash \Delta_1$ *é verdadeira*.

No provador interativo do PVS, comandos de prova correspondem à aplicação de regras de inferência do cálculo de seqüentes. Ao iniciar uma prova, temos o objetivo a ser provado, representado como $\vdash \Delta$. Assim, uma prova é iniciada aplicando-se a Δ uma regra de inferência de *baixo para cima*. Em qualquer parte de uma prova, a aplicação de uma regra de inferência (comando de prova do PVS) é representada genericamente por $\frac{\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\mathbf{R}$, ou seja, a aplicação da regra $\mathbf{R}$ a uma folha da *árvore de prova* da forma $\Gamma \vdash \Delta$, pode produzir uma árvore com n novas folhas. Ou, se uma folha corresponde a um axioma, o ramo é considerado provado.

As regras de inferência podem ser organizadas em grupos, conforme a seguir. Mostraremos apenas alguns grupos e regras.

- Regras estruturais: Um seqüente tem sua estrutura rearranjada ou *enfraquecida* (*weakened*) pela adição de novas fórmulas antecedentes ou consequentes, o que é expresso pela regra $\mathbf{W}$, abaixo.

  $$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\mathbf{W}, \text{ se } \Gamma_1 \subseteq \Gamma_2 \text{ e } \Delta_1 \subseteq \Delta_2.$$

  Podemos ver que as regras de contração $\frac{A,A,\Gamma \vdash \Delta}{A,\Gamma \vdash \Delta}\mathbf{C} \vdash$ e $\frac{\Gamma \vdash \Delta,A,A}{\Gamma \vdash \Delta,A} \vdash \mathbf{C}$ são absorvidas pela regra $\mathbf{W}$.

- Regra do corte: Amplamente utilizada em nossas provas, como será mostrado no capítulo **Capítulo** 5, esta regra corresponde à introdução de uma fórmula no seqüente através de comandos do tipo *case* do PVS. Assim, a regra do corte, a partir de um seqüente $\Gamma \vdash \Delta$, produz dois ramos na árvore de prova ao introduzir uma fórmula A. Em um ramo assume-se A e no outro ramo a negação de A.

  $$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash \Delta}\mathbf{Corte}$$

- Axiomas proposicionais: Quando um axioma proposicional pode ser aplicado a um seqüente-folha da árvore de prova, então o ramo correspondente é considerado provado. A aplicação de regras deste tipo, no PVS, é feita automaticamente pelo sistema em casos triviais ou através do comando `assert`. A regra $\mathbf{Ax}$ garante que, dada a fórmula A, A é verdadeira. E as regras $\mathbf{FALSE} \vdash$ e $\vdash \mathbf{TRUE}$ mostram que um seqüente com uma fórmula falsa no antecedente ou uma fórmula verdadeira no conseqüente é um axioma.

$$\frac{}{\Gamma, A \vdash A, \Delta}\ \mathbf{Ax} \qquad \frac{}{\Gamma, FALSE \vdash \Delta}\ \mathbf{FALSE}\vdash \qquad \frac{}{\Gamma \vdash TRUE, \Delta}\ \vdash\mathbf{TRUE}$$

- Regras condicionais: Utilizadas para eliminar, em uma prova, a estrutura **IF-THEN-ELSE**. Representa-se como $\mathbf{IF}(A, B, C)$ a declaração $\mathbf{IF}\ A\ \mathbf{THEN}\ B\ \mathbf{ELSE}\ C$. Esta regra gera dois novos ramos na árvore de prova e é aplicada com o comando `prop` do PVS.

$$\frac{\Gamma, A, B \vdash \Delta \quad \Gamma, C \vdash A, \Delta}{\Gamma, \mathbf{IF}(A, B, C) \vdash \Delta}\ \mathbf{IF}\vdash \qquad \frac{\Gamma, A \vdash B, \Delta \quad \Gamma \vdash A, C, \Delta}{\Gamma \vdash \mathbf{IF}(A, B, C), \Delta}\ \vdash \mathbf{IF}$$

3.2 O assistente de provas PVS

Em PVS, um seqüente $\Gamma \vdash \Delta$, com $A_i \in \Gamma$ e $B_k \in \Delta$ $(0 < i \leq n$ e $0 < k \leq m)$, é mostrado na forma:

```
  [-1] A1
     ⋮
  [-n] An
|----------
  [1] B1
     ⋮
  [m] Bm
```

Na árvore de prova mantida pelo provador do PVS, cada nó filho é um seqüente gerado a partir da aplicação de algum comando PVS. Apenas um seqüente-folha é ativado por vez, ou seja, a aplicação de um comando do provador é realizada a somente um sub-objetivo. O término da prova de um seqüente passa o controle para a próxima folha não provada da árvore. A prova termina quando todos os seqüentes da árvore são provados.

A linguagem do PVS contém tipos primitivos, como inteiros e booleanos, bem como tuplas, registros e construtores para funções. Por exemplo, `[bool, int -> nat]` define um tipo de função que recebe como argumentos um par booleano/inteiro e retorna um natural. Além disso, tipos não interpretados também podem ser definidos. Por exemplo, para representar um usuário de um protocolo criptográfico, podemos definir um tipo não-vazio `U`, da seguinte forma: `U : TYPE+`. Portanto, dois usuários `x` e `y` podem ser declarados como `x, y : U`. Outra possibilidade importante é a definição de tipos enumerados em PVS, exemplificada pela definição dos tipos de operadores criptográficos (encriptação e decriptação), como a seguir:

```
cryptType : TYPE = {decrypt, encrypt}
```

Em PVS, uma especificação consiste de uma coleção de *teorias* que podem ser adequadamente utilizadas para modularização de especificações, onde teorias complexas podem ser provadas a partir de teorias mais simples. Cada teoria pode ser parametrizada e é composta essencialmente de um conjunto de declarações, que podem ser a introdução de nomes de tipos, constantes, variáveis, axiomas, fórmulas e *IMPORTING*s, que permitem importar outras teorias.

A parametrização de teorias permite construções genéricas, como mostrado no código abaixo. A teoria `finite_sequences_extras` importa a teoria `finite_sequences` para manipulação de seqüências finitas.

```
finite_sequences_extras[T: TYPE] : THEORY
BEGIN
  IMPORTING finite_sequences[T]
```

`T` é tratado como um tipo não interpretado. Conseqüentemente, quando a teoria `finite_sequences_extras` é invocada por outra teoria, `T` precisa ser especificado. Por exemplo, `finite_sequences_extras[int]` mostra a utilização da teoria `finite_sequences_extras` para o tipo `int`.

Toda especificação em PVS deve passar por um passo de *checagem de tipos*, que resolve referências e procura por erros semânticos, como nomes não declarados e tipos ambíguos, garantindo a consistência de tipos da especificação. A checagem de tipos é um procedimento indecidível, já que o usuário pode definir expressões booleanas arbitrárias em uma especificação. Assim, a checagem de tipos pode produzir *TCC*s (*Type Correctness Conditions*), que são declarações que devem ser validadas para que uma teoria seja considerada completa. Através de estratégias pré-definidas, o PVS pode provar automaticamente diversos tipos de TCCs, mas em alguns casos, geralmente em definições recursivas ou subtipos, é necessário que o usuário dirija a prova no PVS.

3.2.1 Exemplo de especificação em PVS: Soma de PA.

Para ilustrar a linguagem e os procedimentos de especificação e verificação formal no PVS, utilizaremos uma prova indutiva para a soma de termos de uma *Progressão Aritmética* (*PA*).

Defina-se uma PA, tal que os números naturais $a_1 \geq 0,\ r \geq 0,\ n > 0$ representem, respectivamente, o primeiro termo, a razão e o número de termos desta PA. A soma dos n termos da PA é dada por:

$$S_n = a_1 + a_2 + ... + a_n = \frac{n(a_1 + a_n)}{2} = \frac{n(a_1 + (a_1 + r(n-1)))}{2}, \text{ tal que}$$

$$a_2 = a_1 + r$$
$$a_3 = a_2 + r = a_1 + 2r$$
$$\vdots$$
$$a_n = a_{n-1} + r = a_1 + r(n-1)$$

Equação 1: A soma dos n termos de uma PA.

A expressão anterior pode ser especificada como uma teoria em PVS, conforme abaixo.

```
1. progressaoaritmetica : THEORY
2.
3. BEGIN
4.
5.   somaPA(a1 : nat, r : nat, n : posnat) : RECURSIVE nat =
6.     IF(n = 1) THEN a1
7.     ELSE a1 + somaPA(a1+r,r,n-1)
8.     ENDIF
9.   MEASURE n
10.
11.  exprSomaPA : LEMMA FORALL(a1 : nat, r : nat, n : posnat) :
12.    somaPA(a1,r,n) = n*(a1 + (a1 + r*(n-1))) / 2
13.
14.END progressaoaritmetica
```

A teoria `progressaoaritmetica` define a soma de termos da PA $S_n = a_1 + a_2 + ... + a_n$ na forma da função recursiva `somaPA`, que retorna um número natural. Isto é definido na função por **`RECURSIVE`** `nat`. Os parâmetros de `somaPA` são os números naturais $a_1 \geq 0,\ r \geq 0,\ n > 0$ da **Equação 1**, conforme assinatura da função dada por `somaPA(a1 : nat, r : nat, n : posnat)`. O PVS verifica automaticamente que `somaPA` é terminante, pois a medida da recursividade (**`MEASURE`** `n`) é estritamente decrescente para cada chamada recursiva.

Provaremos a **Equação 1** com o lema `exprSomaPA`, nas linhas 11 e 12 do código da teoria `progressaoaritmetica`. A palavra-chave **`LEMMA`** define o lema a ser provado e **`FORALL`** é o quantificador universal. O lema expressa a seguinte propriedade:

$$\forall(a_1 \mid a_1 \in N, r \mid r \in N, n \mid n \in N^*) : a_1 + (a_1 + r) + ... + (a_1 + r(n-1)) = \frac{n(a_1 + (a_1 + r(n-1)))}{2}$$

Mostraremos os passos principais da prova em PVS, indicando o comando utilizado com descrição de seu efeito. A prova se inicia através do comando *prove*. O objetivo inicial é mostrado a seguir. Os seqüentes serão rotulados como (S1), (S2), ..., para posterior identificação dos seqüentes na árvore de prova.

(S1)
```
  |-------
{1}  FORALL (a1: nat, r: nat, n: posnat):
       somaPA(a1, r, n) = n * (a1 + (a1 + r * (n - 1))) / 2
```

- `(induct n)`: Define a indução em `n`, produzindo dois ramos principais na prova: o ramo para a base de indução e o ramo para o passo de indução. Na base, temos:

(S2)
```
{-1}  n = 0
  |-------
[1]   n > 0
[2]   FORALL (a1: nat, r: nat):
       somaPA(a1, r, 1 + n) =
        (2 * a1 + r * n + (2 * (a1 * n) + r * n * n)) / 2
```

- `(grind)`: O objetivo é provar que, para $n=0$, ou vale $n>0$ (claramente, este termo é falso), ou vale o termo identificado por `[2]`. O comando `(grind)` é uma estratégia que tenta, repetidamente, aplicar *skolemização*, expansão de termos, instanciação, e outros. Com o valor de $n=0$, a expansão e validação pelo PVS do termo `somaPA` em `[2]`, retornará `a1`, que é igual ao lado direito da expressão. Assim, completa-se a base da indução, e o passo de indução é apresentado:

(S3)
```
  |-------
{1}  FORALL n:
       (n > 0 =>
         (FORALL (a1: nat, r: nat):
            somaPA(a1, r, n) = n * (a1 + (a1 + r * (n - 1))) / 2))
        =>
        n + 1 > 0 =>
         (FORALL (a1: nat, r: nat):
            somaPA(a1, r, n + 1) =
             (n + 1) * (a1 + (a1 + r * (n + 1 - 1))) / 2)
```

- `(skolem 1 n)`: Introduz a constante *Skolem* `n`.

(S4)
```
  |-------
{1}  (n > 0 =>
       (FORALL (a1: nat, r: nat):
          somaPA(a1, r, n) = n * (a1 + (a1 + r * (n - 1))) / 2))
      =>
      n + 1 > 0 =>
       (FORALL (a1: nat, r: nat):
          somaPA(a1, r, n + 1) =
           (n + 1) * (a1 + (a1 + r * (n + 1 - 1))) / 2)
```

- `(assert)`: Procedimento de decisão que, neste caso, validou a inequação `n + 1 > 0` e expandiu as equações da fórmula `{1}`.

(S5)
```
|-------
{1}   (n > 0 =>
       (FORALL (a1: nat, r: nat):
          somaPA(a1, r, n) = (2 * (a1 * n) - r * n + r * n * n) / 2))
      =>
      (FORALL (a1: nat, r: nat):
        somaPA(a1, r, 1 + n) =
         (2 * a1 + r * n + (2 * (a1 * n) + r * n * n)) / 2)
```

- `(prop)`: Realiza a simplificação proposicional, principalmente pela construção de sub-objetivos que não possuem elementos conectivos proposicionais. Neste caso, `prop` divide em dois sub-objetivos o termo `{1}`. O primeiro é trivial e contempla um caso onde $n = 0$. O segundo sub-objetivo é mostrado a seguir. No termo `{2}` abaixo, utilizamos o comando `(expand somaPA)` para expandir a função `somaPA` em um passo da recursividade, o que equivale dizer que

(S6) $a_1 + (a_1 + r) + \cdots + (a_1 + rn) = a_1 + [(a_1 + r) + \cdots + ((a_1 + r) + r(n-1))]$.

```
[-1]  FORALL (a1: nat, r: nat):
       somaPA(a1, r, n) = (2 * (a1 * n) - r * n + r * n * n) / 2
  |-------
{1}   n = 0
{2}   FORALL (a1: nat, r: nat):
       a1 + somaPA(a1 + r, r, n) =
        (2 * a1 + r * n + (2 * (a1 * n) + r * n * n)) / 2
```

- *Skolemizamos* o termo `{2}` para eliminação do quantificador universal através do comando `(skolem 2 ("a1" "r"))`. Em seguida, o termo `[-1]` é instanciado `(inst -1 "a1 + r" "r")`, gerando-se a hipótese da indução dada por $(a_1 + r) + (a_1 + 2r) + \cdots + (a_1 + rn) = \dfrac{n((a_1 + r) + (a_1 + rn))}{2}$.

(S7)
```
{-1} somaPA(a1 + r, r, n) = (2 * ((a1 + r) * n) - r * n + r * n * n)/2
  |-------
[1]   n = 0
[2]   a1 + somaPA(a1 + r, r, n) =
       (2 * a1 + r * n + (2 * (a1 * n) + r * n * n)) / 2
```

- Assim, queremos demonstrar que `[2]` vale; utilizando a hipótese em `{-1}`. Com o comando `(replace -1 2)` realizamos a aplicação da hipótese de indução, substituindo o termo `somaPA(a1 + r, r, n)`, em `[2]`, pelo lado direito deste termo em `{-1}`. Finalmente, com um comando `(assert)`, o PVS conclui a prova verificando a igualdade do termo produzido após a substituição em `[2]`.

A **Figura 2**, a seguir, mostra a árvore de prova para a soma da PA. Os seqüentes analisados acima estão mostrados em nós correspondentes da árvore.

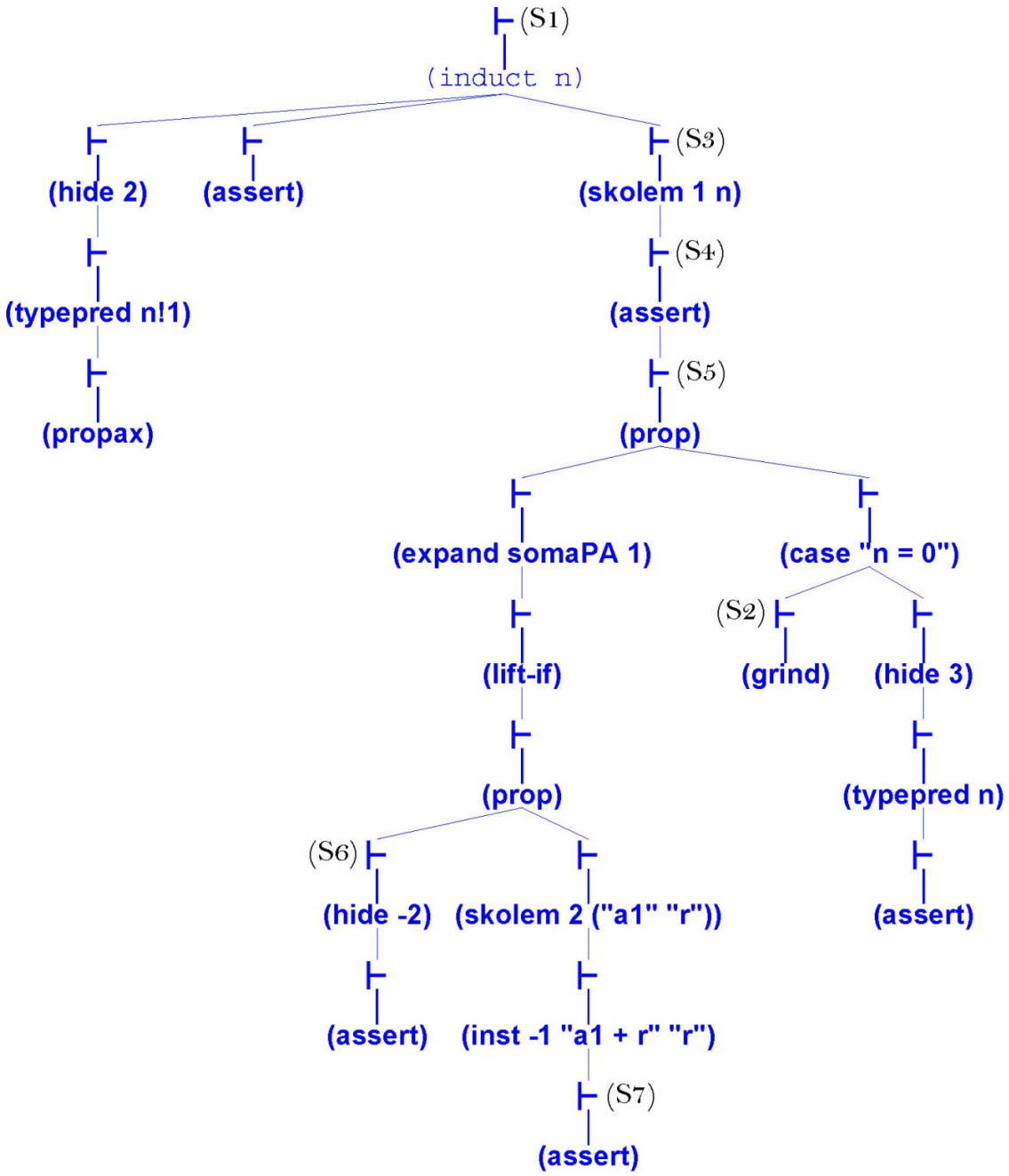

Figura 2: Árvore de prova para a soma da PA.

3.2.2 Outros detalhes de especificação e verificação em PVS.

Apresentamos acima alguns comandos importantes do provador do PVS, e aspectos da linguagem de especificação. Em nossa formalização do modelo Dolev-Yao para protocolos em cascata, utilizamos vários outros elementos, dos quais destacamos os

mostrados a seguir. Não serão considerados detalhes definicionais dos fragmentos de provas apresentados.

Algumas definições de tipos

Registros: têm a forma `[# a1:t1, ..., an:tn #]`, onde `ai` é um campo do registro e `ti` é o tipo de `ai`.

Conjuntos: são definidos no *prelude* do PVS como `setof[T]`, onde `T` é o tipo dos elementos do conjunto. Uma utilização comum é em funções que retornam um conjunto a partir de um argumento. Por exemplo, podemos especificar uma função `conjuntomenorque(n : nat)`, que retorna o conjunto de números naturais menores que `n`, conforme a seguir.

```
conjuntomenorque(n : nat) : setof[nat] = {num : nat | num < n}
```

Através a função `member` pode-se verificar se um elemento pertence a um conjunto. `member` recebe um atributo a, de tipo `T`, e um conjunto C, de tipo `setof[T]`, retornando verdadeiro caso $a \in C$, e falso caso contrário. Por exemplo, é verdadeira a expressão `member(10,conjuntomenorque(100))`.

Alguns comandos do provador

- `measure-induct+`: Em nosso contexto, este comando foi principalmente utilizado em provas por indução sobre o comprimento de seqüências, que são compostas por dois componentes: `length` e `seq`. Por exemplo, para provar que

  ```
  |-------
   FORALL(seq: reduzibleseq): first_cancelable(seq) < seq`length - 1,
  ```

  utilizamos `(measure-induct+ "seq`length" "seq")`. Obtemos, então, a seguinte estrutura indutiva:

  ```
  {-1}  FORALL (y: reduzibleseq):
           y`length < seq `length => first_cancelable(y) < y`length - 1
  |-------
  {1}   first_cancelable(seq) < seq `length - 1
  ```

  Isto é, se para toda seqüência `y` do tipo `reduzibleseq`, com comprimento menor que outra sequencia `seq` do mesmo tipo, vale a propriedade `first_cancelable(y) < y`length - 1`, então vale a propriedade da fórmula `{1}`.

- `typepred`: Explicita um predicado que define o tipo de um elemento. Geralmente, este elemento é uma variável *skolemizada*. Se aplicado ao exemplo apresentado para

o comando anterior, `(typepred seq)` acrescentaria uma fórmula `reduzibleseq?(seq)` como premissa. Este predicado é precisamente o que define o tipo de `seq`.

- `decompose-equality`: Decompõe uma igualdade de elementos como os registros, gerando novas igualdades (fórmulas) para cada um dos campos do registro. Por exemplo, um registro de tipo `op` (operador criptográfico) é definido, em PVS, como uma composição de um tipo de operação `crTyp` (encriptação ou decriptação) e um usuário `user`, como a seguir:

```
op : TYPE = [# crTyp : cryptType, user : U #].
```

  Sendo `op1` e `op2` duas variáveis do tipo `op`, a igualdade `op1 = op2` pode ser decomposta pelo comando `decompose-equality`, resultando nas duas igualdades abaixo:

```
op1`crTyp = op2`crTyp, e
op1`user = op2`user
```

- `case`: Na utilização mais usual deste comando, criam-se dois sub-objetivos a partir do ponto onde é executado. No primeiro, a fórmula especificada no `case` aparece no antecedente, no segundo, aparece no conseqüente. Em uma prova por contradição, podemos utilizar o comando `case` da seguinte forma:

```
{-1}  alpha0ContainsE?(prot, x, y)
{-2}  balanced_cascade_protocol?(prot)
|-------
{1}   secure_protocol?(prot, x, y, z)
```

A fórmula em `{1}` é um predicado que avalia se o protocolo criptográfico `prot` é seguro. Utilizando o comando `(case "insecure_protocol?(prot,x,y,z)")` forma-se a estrutura para uma prova por contradição, pois adiciona-se esta fórmula ao antecedente da prova. `insecure_protocol?` representa, logicamente, o inverso da definição expressa por `secure_protocol?`.

O `case` aplica-se também em situações onde é necessário tratar casos específicos durante uma prova. Por exemplo, considere-se o seguinte estado de uma prova:

```
{-1}  normalizeseq(seq1 o seq2) = empty_seq
|-------
{1}   areseqcomplements?(seq1, seq2)
```

Na situação acima, `seq1` e `seq2` são seqüências finitas. Cada seqüência possui um atributo que representa seu comprimento (`length`). `empty_seq` denota a seqüência vazia. Caso seja

necessário considerar um caso em que o comprimento de `seq1` é zero, pode-se utilizar o comando `case`: `(case "seq1`length = 0")`, que criará dois sub-objetivos: o primeiro, onde a fórmula `seq1`length = 0` aparece no antecedente, e o segundo, onde esta mesma fórmula aparece no conseqüente. Assim, o primeiro sub-objetivo forma o caso em que se deseja um tratamento especifico para `seq1`length = 0`. O segundo sub-objetivo forma o caso geral, onde `seq1`length = 0` é falso, ou seja, `seq1`length > 0`.

Capítulo 4.
O Modelo Dolev-Yao para Protocolos em Cascata

A seguir, apresentamos a formalização analítica para a segurança de protocolos cascata, mostrando também as condições sob as quais se define essa segurança. Em alguns aspectos, nossa modelagem difere daquela originalmente proposta por Dolev e Yao em [6], principalmente porque a formalização do modelo original apresenta imprecisões, o que foi verificado através da mecanização do modelo em PVS. As definições, lemas e provas, analiticamente apresentados a seguir, foram baseados na especificação e verificação realizadas em PVS.

4.1 Conceitos e definições básicos

Em um sistema de chaves públicas, seja U um conjunto enumerável de usuários. Um *participante* $x \in U$ tem um par de operadores E_x e D_x que denotam, respectivamente, a função de x para encriptação e a função de x para decriptação de mensagens. O conjunto de operadores $E = \{E_x | x \in U\}$ é dito público, ou seja, é conhecido por todos os usuários. Por outro lado, operadores $D = \{D_x | x \in U\}$ são privados, de modo que D_x é conhecido somente pelo usuário x. Restrições deste tipo, assumidas sempre como válidas, são chamadas de *primitivas criptográficas*.

A aplicação de operadores de E e de D, a uma mensagem M, é representada por $E_x(M)$ e $D_x(M)$, para todo usuário $x \in U$. Para melhor legibilidade, escrevemos $\eta_1\left(\eta_2\left(\eta_3(\ldots\eta_n(M)\ldots)\right)\right)$ como $\eta_1\eta_2\eta_3\ldots\eta_n M$, onde η_i $(i = 1..n)$ é um operador. As funções de encriptação e decriptação são inversas para todos os usuários, ou seja, para qualquer mensagem M e qualquer $x \in U$, vale

$$E_x\big(D_x(M)\big) = D_x\big(E_x(M)\big) = M$$

Seja $\Sigma = E \cup D$, e Σ^* o conjunto de palavras finitas sobre Σ, com λ denotando a palavra vazia. Utilizaremos o monóide gerado pelos símbolos de Σ, considerando-se o seguinte conjunto de congruências:

$$E_x D_x = D_x E_x = \lambda, \forall x \in U$$

Neste monóide módulo as congruências acima, para toda palavra $\delta \in \Sigma^*$, existe uma única forma canônica $\bar{\delta}$, onde não existe nenhuma subpalavra da forma $E_x D_x$ ou $D_x E_x$, para qualquer $x \in U$. Desta forma, definimos a *normalização* e, sempre que $\delta = \bar{\delta}$, δ está na *forma normal* ou *reduzida*; caso contrário, δ é chamado de *normalizável* ou *redutível*. Considerando um usuário $a \in U$, dizemos que $\delta|_a$ é normal com respeito a a, quando não existe nenhuma subpalavra $E_a D_a$ ou $D_a E_a$ em δ.

Alguns lemas simples, como a propriedade a seguir, são provados na formalização em PVS, e suas provas, ainda que rotineiras, são tecnicamente elaboradas.

Lema 1 [Uma propriedade de normalização] Para todo $\delta_1, \delta_2, \delta_3 \in \Sigma^*$, vale $\overline{\delta_1 \delta_2 \delta_3} = \overline{\delta_1 \overline{\delta_2} \delta_3}$.

Prova. Por indução no comprimento de δ_2.

BI. Caso $\delta_2 = \lambda$, $\bar{\lambda} = \lambda$, e $\overline{\delta_1 \lambda \delta_3} = \overline{\delta_1 \delta_3}$.

PI. Vamos assumir que, para $\delta_2 = \gamma_1 \gamma_2$ $(\gamma_1, \gamma_2 \in \Sigma^*)$, vale $\overline{\delta_1 \delta_2 \delta_3} = \overline{\delta_1 \overline{\delta_2} \delta_3}$. Seja δ_2' da forma $\gamma_1 E_x D_x \gamma_2$ ou $\gamma_1 D_x E_x \gamma_2$. Pela definição de normalização, $\overline{\delta_2'} = \overline{\gamma_1 \gamma_2}$, então temos que $\overline{\delta_1 \delta_2' \delta_3} = \overline{\delta_1 \gamma_1 \gamma_2 \delta_3}$. Assim, por HI, $\overline{\delta_1 \delta_2' \delta_3} = \overline{\delta_1 \overline{\delta_2'} \delta_3}$.

Para todo $\delta \in \Sigma^*$, $|\delta|$ denota o comprimento da seqüência δ, ou seja, a quantidade de operadores em δ. E, para todo $0 \leq j < n$, onde $|\delta| = n \in \mathbb{N}$, δ_j é o $(j+1)$-ésimo símbolo em δ. Para $0 \leq j \leq k < n$, $\delta_{[j,k]}$ denota a subpalavra de δ composta pelos símbolos $(j+1)$-ésimo a $(k+1)$-ésimo.

O *complemento* de um operador é definido, para todo $x \in U$, como $E_x^c = D_x$ e $D_x^c = E_x$. Para todo $\delta \in \Sigma^*$, δ^c denota o *complemento* de δ, definido como:

$$\lambda^c = \lambda$$
$$\forall 0 \leq i < |\delta| : (\delta^c)_i = \left(\delta_{|\delta|-1-i}\right)^c$$

Por exemplo, $(E_x E_x D_x)^c = E_x D_x D_x$.

Por premissas estabelecidas anteriormente para um sistema de chaves públicas no modelo Dolev-Yao, um usuário $x \in U$ não possui informações sobre operadores privados de um usuário $y \in U \mid y \neq x$. Assim, definimos $\forall x, y \in U$ a função $\Phi(x, y)$ que, para cada par de usuários, retorna o conjunto de operadores que podem ser aplicados por x a uma mensagem M enviada a y. Assim, $\Phi(x, y) = \{D_x, E_y, E_x\}$.

Com esses elementos, podemos apresentar a noção de protocolos em cascata entre dois usuários. Protocolos desta classe consistem de uma seqüência de passos de comunicação de ida e volta, alternadamente, entre dois usuários, utilizando os operadores de Φ em cada passo.

4.2 Formalização analítica do modelo

4.2.1 Protocolo em cascata e linguagem do adversário

Nesta seção, apresentamos as definições formais que descrevem um protocolo em cascata e, com base nestas definições, a linguagem utilizada pelo adversário na interação com esse protocolo.

Definição 1 [Passo de protocolo] Para a comunicação entre dois usuários quaisquer, definimos a função $\alpha\beta: U \times U \to \Sigma^*$, de modo que, $\forall x, y \in U \mid x \neq y$, temos:

$$
\begin{cases}
1.\ \alpha\beta(x,y) \neq \lambda \\
2.\ \alpha\beta(x,y) = \overline{\alpha\beta(x,y)} \\
3.\ \alpha\beta(x,y) \in \Phi(x,y)^* \\
4.\ \forall u, v \in U: \\
\quad 4.1.\ |\alpha\beta(x,y)| = |\alpha\beta(u,v)| \\
\quad 4.2.\ \forall 0 \leq j < |\alpha\beta(x,y)|: \\
\quad\quad 4.2.1.\ \alpha\beta(x,y)_j = E_x \ sse\ \alpha\beta(u,v)_j = E_u \\
\quad\quad 4.2.2.\ \alpha\beta(x,y)_j = E_y \ sse\ \alpha\beta(u,v)_j = E_v \\
\quad\quad 4.2.3.\ \alpha\beta(x,y)_j = D_x \ sse\ \alpha\beta(u,v)_j = D_u \\
\quad\quad 4.2.4.\ \alpha\beta(x,y)_j = D_y \ sse\ \alpha\beta(u,v)_j = D_v
\end{cases}
$$

Isto é, um $\alpha\beta(x,y)$ é uma seqüência não vazia (condição 1) e normal (condição 2) de operadores utilizados por x ao enviar uma mensagem qualquer a y (condição 3). Além disso, todos os pares de usuários, distintos entre si, trocam mensagens segundo as mesmas regras (condição 4), ou seja, $\forall u, v \in U$, temos que um $\alpha\beta(u,v)$ possui a mesma seqüência de operadores de $\alpha\beta(x,y)$, exceto pelo renomeamento dos usuários.

Definição 2 [Protocolo em cascata] Um protocolo em cascata P é uma seqüência não vazia de passos de protocolo. Para a comunicação entre dois usuários $x, y \in U$ quaisquer, P aplica os passos de protocolo de forma que a comunicação alterna-se entre os usuários x e y; isto é, $\forall 0 \leq i < |P|$, o passo P_i aplica-se segundo os seguintes casos:

1. $P_i(x,y)$, para i par (x envia mensagem para y)
2. $P_i(y,x)$, para i ímpar (y envia mensagem para x)

Por abuso de notação, representaremos um $P_i(x,y)$ ou $P_i(y,x)$ por P_i, conforme seja conveniente. Uma subseqüência $P_i \dots P_j$ $(0 \leq i \leq j < |P|)$ de P será representada por $P_{[i,j]}$.

Na definição acima, a comunicação entre um par de usuários x e y alterna em cada passo de comunicação a sua direção; nas posições pares x envia mensagens para y e, nas posições ímpares, y envia mensagens para x. Isso é formalizado ao se intercambiar os argumentos x e y dos passos de protocolo. Na comunicação entre estes dois usuários, cada passo de protocolo P_i é aplicado seqüencialmente a uma mensagem M, a partir de P_0. Ou seja, considerando que $x \to y$ represente o envio de uma mensagem M de x para y, temos:

$x \to y$: $P_0 M = \alpha\beta_0(x,y)M$
$y \to x$: $P_1 P_0 M = \alpha\beta_1(y,x)\alpha\beta_0(x,y)M$
$\vdots$
$x \to y$: $P_{|P|-1} \dots P_0 M = \alpha\beta_{|P|-1}(x,y) \dots \alpha\beta_0(x,y)M$, se $|P| > 2$ $ímpar$; **ou**
$y \to x$: $P_{|P|-1} \dots P_0 M = \alpha\beta_{|P|-1}(y,x) \dots \alpha\beta_0(x,y)M$, se $|P| > 2$ par.

Definição 3 [Linguagem admissível do adversário] Um adversário z, que tenta subverter um protocolo em cascata P, dispõe de uma linguagem admissível $(\Sigma_1(z)^* \cup \Sigma_2)^*$, onde

$\Sigma_1(z) = E \cup \{D_z\}$, e
$\Sigma_2 = \{P_i(x,y) \mid 1 \leq i < |P| \text{ e } x, y \in U, x \neq y\}$.

A definição acima descreve as possibilidades de interação de um adversário z, no modelo Dolev-Yao, com o protocolo P. Assume-se que z seja capaz de observar todo o tráfego da rede e que seja limitado pelas primitivas criptográficas, ou seja, z não possui nenhum conhecimento sobre o conjunto $D\backslash\{D_z\}$. Desta forma, para se obter um γ dentro da linguagem admissível, z pode:

- Utilizar a linguagem $\left(\Sigma_1(z)\right)^*$: *$z$ faz qualquer coisa que um usuário honesto é capaz de fazer* e, assim, possui um operador privado e tem conhecimento de todo operador público.
- Utilizar passos de protocolo $\{P_i(x,y) \mid i \text{ par e } 1 < i < |P| \text{ e } x,y \in U\}$: *$z$ é capaz de criar, interceptar e alterar mensagens*. Assim, para obter $P_i(x,y)$ aplicado a uma mensagem, z intercepta a $(i-1)$-ésima mensagem M de y para x. Então, *z se passa por y*, o que denotamos por $z(y)$, e atua ativamente na comunicação com x, enviando uma mensagem M' arbitrariamente escolhida, ou seja,

  $z(y) \to x$: M' (passo P_{i-1} é executado por z como se fosse y)
  $x \to z(y)$: $P_i(x,y)M'$ (passo i de P),

  A restrição de que z não dispõe de P_0 deixa claro que a comunicação ocorre por iniciativa do usuário x, que envia $P_0(x,y)M$ a y, ou seja, z deve esperar que x inicie uma conversa com y. Apesar da possibilidade de isto não acontecer, consideramos o pior caso para fins de definição de segurança.
- Utilizar passos de protocolo $\{P_i(y,x) \mid i \text{ ímpar e } 1 \leq i < |P| \text{ e } x,y \in U\}$: Neste caso, z inicia uma conversa com y, passando-se por x. Para que z obtenha um $P_i(y,x)$ aplicado a uma mensagem M' escolhida, as seguintes operações sobre M' são executadas a partir do passo $i-1$ de P:

  $z(x) \to y$: M' (passo $i-1$ de P)
  $y \to z(x)$: $P_i(y,x)M'$ (passo i de P)

4.2.2 Definindo a segurança de protocolos em cascata

Mostraremos, a seguir, as propriedades e condições que possibilitam a definição da segurança para protocolos em cascata.

Definição 4 [Propriedade de balanceamento (PB)] Seja $\boldsymbol{\pi} \in \{\boldsymbol{E}, \boldsymbol{D}\}^*$. $\boldsymbol{\pi}$ possui a PB com respeito a $\boldsymbol{z} \in \boldsymbol{U}$, quando a existência de um operador privado de $\boldsymbol{z}$ em $\boldsymbol{\pi}$ implica na existência de um operador público de $\boldsymbol{z}$ em $\boldsymbol{\pi}$, ou seja,

$$\exists 0 \leq i < |\pi| : \pi_i = D_z \Rightarrow \exists 0 \leq j < |\pi| : \pi_j = E_z$$

Definição 5 [Protocolo balanceado] Um protocolo em cascata P é balanceado se

$$\begin{cases} \forall x, y \in U \text{ e } \forall 0 < i < |P|: \\ \quad P_i(x, y) \text{ tem a PB com respeito a } x, \text{ se } i \text{ é par.} \\ \quad P_i(y, x) \text{ tem a PB com respeito a } y, \text{ se } i \text{ é ímpar.} \end{cases}$$

Por exemplo, supondo que $P_2(x, y) = E_y D_x E_y$, então o protocolo P não é balanceado, uma vez que $P_2(x, y)$ não tem a PB com respeito a x.

Definição 6 [Condição inicial de segurança] Dizemos que um protocolo em cascata P satisfaz a *condição inicial de segurança* quando, $\forall x, y \in U$, temos:

$$P_0(x, y) \cap \{E_x, E_y\} \neq \phi$$

Sem esta condição, um adversário $z \in U$ pode trivialmente obter uma mensagem M transmitida entre os usuários x e y, já que x dispõe somente do operador D_x.

Definição 7 [Protocolo seguro] Um protocolo em cascata P é seguro quando, para quaisquer três usuários distintos entre si $x, y, z \in U$, $\forall \gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$ e $0 \leq i < |P|$, vale

$$\overline{\gamma P_i \dots P_0} \neq \lambda$$

Caso contrário, o protocolo é inseguro.

Por esta definição, nenhum adversário z pode inverter uma seqüência de passos de protocolo, a fim de obter uma mensagem M a ser transmitida em sigilo entre os usuários x e y.

4.2.3 Balanceamento da linguagem admissível do adversário

Nesta seção, provamos algumas propriedades importantes relativas à linguagem admissível do adversário. Para isso, inicialmente, definimos o a-balanceamento e a propriedade de enlace, descritos a seguir.

Definição 8 [a-balanceamento] Seja uma seqüência de operadores π e um usuário $a \in U$. Dizemos que π é a-balanceada quando o seguinte é satisfeito: Se $\exists x, y \in U \mid x, y \neq a$ com $\pi_0 = D_x$ e $\pi_{|\pi|-1} = D_y$ e $\pi_{[1,|\pi|-2]}|_a$ contém D_a, então $\pi_{[1,|\pi|-2]}|_a$ tem a PB com respeito a a.

Definição 9 [Propriedade de enlace – PE] Seja um usuário $z \in U$. Uma seqüência de operadores η tem a PE com respeito a z quando toda substring de $D_z \eta D_z$ é, para todo usuário $a \in U \mid a \neq z$, a-balanceada.

Lema 2 [$\Sigma_1^*(z)$ e Σ_2 satisfazem PE]: Seja P um protocolo em cascata balanceado, z um usuário e δ uma seqüência de operadores. Se $\delta \in \Sigma_1^*(z)$ ou $\delta \in \Sigma_2$, então δ tem a PE com respeito a z.

Prova. Precisamos demonstrar que, para todo natural $i, j \mid i < j < |\delta| + 2$ e todo usuário $a \in U \mid a \neq z$, $(D_z \delta D_z)_{[i,j]}$ (substrings de $D_z \delta D_z$) é a-balanceada, ou seja, que o seqüente a seguir é verdadeiro:

$\delta \in \Sigma_1^*(z)$ ou $\delta \in \Sigma_2$,
$\exists x, y \in U \mid x, y \neq a$, tais que $\left((D_z \delta D_z)_{[i,j]}\right)_0 = D_x$ e
$\left((D_z \delta D_z)_{[i,j]}\right)_{|(D_z \delta D_z)_{[i,j]}|-1} = D_y$,
$D_a \in \left((D_z \delta D_z)_{[i,j]}\right)_{[1,|(D_z \delta D_z)_{[i,j]}|-2]}|_a$,

implica

$\left((D_z \delta D_z)_{[i,j]}\right)_{[1,|(D_z \delta D_z)_{[i,j]}|-2]}|_a$ tem a PB com respeito a a.

Consideremos, separadamente, os casos $\delta \in \Sigma_1^*(z)$ e $\delta \in \Sigma_2$.

- $\delta \in \Sigma_1^*(z)$: Como $D_z \in \Sigma_1(z)$, vale $D_z \delta D_z \in \Sigma_1^*(z)$. Assim, temos que $(D_z \delta D_z)_{[i,j]} \in \Sigma_1^*(z)$ implica $D_a \notin \left((D_z \delta D_z)_{[i,j]}\right)_{[1,|(D_z \delta D_z)_{[i,j]}|-2]}|_a$, pois $D_a \notin \Sigma_1^*(z)$.

- $\delta \in \Sigma_2$: Para este caso, consideramos o protocolo P entre dois usuários $u, v \in U$, com $\delta = P(u,v)_k$, onde $0 < k < |P|$. Assim, diferenciamos dois casos: $a = u$ e $a \neq u$. Os casos em que $a = v$ e $a \neq v$ são análogos.
  - ❖ $a = u$: Para o caso em que $i = 0$ e $j = |\delta| + 1$, $\left((D_z \delta D_z)_{[i,j]}\right)_{[1,|(D_z \delta D_z)_{[i,j]}|-2]}|_a = \delta$, de modo que $x = z$ e $y = z$, na definição de a-balanceamento. Com $D_a \in \delta$, temos a PB para δ, pois P é balanceado. Para $i > 0$ (e, analogamente, para $j < |\delta| + 1$), não existe um usuário $x \neq a$ tal que $\left((D_z \delta D_z)_{[i,j]}\right)_0 = D_x$, já que $\left((D_z \delta D_z)_{[i,j]}\right)_0 = \delta_{i-1}$ e $\delta_{i-1} \in \{E_u, E_v, D_u\}$.
  - ❖ $a \neq u$: Neste caso, $D_a \notin \left((D_z \delta D_z)_{[i,j]}\right)_{[1,|(D_z \delta D_z)_{[i,j]}|-2]}|_a$, uma vez que qualquer operador de $\left((D_z \delta D_z)_{[i,j]}\right)_{[1,|(D_z \delta D_z)_{[i,j]}|-2]}|_a$ corresponde a um operador de $\delta \in \{E_u, E_v, D_u\}^*$.

Lema 3 [Preservação da PE na composição de duas seqüências] Seja um protocolo em cascata balanceado P, um usuário $z \in U$ e as seqüências de operadores μ e η. Se μ e η têm a PE com respeito a z, então a seqüência $\mu\eta$ tem a PE com respeito a z.

Prova. Devemos mostrar que, para todo usuário $a \neq z$, toda substring de $D_z \mu \eta D_z$ é a-balanceada. Sabemos que toda substring de $D_z \mu D_z$ ou $D_z \eta D_z$ é a-balanceada. Então, precisamos mostrar somente que são a-balanceadas as substrings δ de $D_z \mu \eta D_z$, que iniciem em alguma posição de $D_z \mu$ e terminem em alguma posição de ηD_z. Conforme a definição de a-balanceamento: Se $\exists x, y \in U \mid x, y \neq a$ com $\delta_0 = D_x$ e $\delta_{|\delta|-1} = D_y$, e $\delta_{[1,|\delta|-2]}|_a$ contém D_a, então $\delta_{[1,|\delta|-2]}|_a$ tem a PB com respeito a a. Assim, $\delta = D_x \mu_{[i,|\mu|-1]} \eta_{[0,j]} D_y$ $(i, j \in \mathbb{N} \mid i < |\mu| \text{ e } j < |\eta|)$. Vamos supor que $\delta_{[1,|\delta|-2]}|_a$ não tem a PB com respeito a a. Deste modo, $D_a \in \delta_{[1,|\delta|-2]}|_a$, mas $E_a \notin \delta_{[1,|\delta|-2]}|_a$. Isto leva a uma contradição, pois resulta na ocorrência de pelo menos um dos dois casos a seguir.

1. $D_a \in \mu_{[i,|\mu|-1]}|_a$: Como $E_a \notin \delta_{[1,|\delta|-2]}|_a$, temos que $E_a \notin \mu_{[i,|\mu|-1]}|_a$ e, assim, a substring $D_x \mu_{[i,|\mu|-1]} D_z$ de $D_z \mu D_z$ não é a-balanceada.
2. $D_a \in \eta_{[0,j]}|_a$: Como $E_a \notin \delta_{[1,|\delta|-2]}|_a$, temos que $E_a \notin \eta_{[0,j]}|_a$ e, assim, a substring $D_z \eta_{[0,j]} D_y$ de $D_z \eta D_z$ não é a-balanceada.

Note que os dois casos acima são simultaneamente verdadeiros, caso não surjam pares normalizáveis E_aD_a ou D_aE_a, ao se compor a seqüência $\mu\eta$, a partir de μ e η. Se um par normalizável é criado na fronteira entre μ e η, então existe um operador E_a em $\mu_{[i,|\mu|-1]}|_a$ ou $\eta_{[0,j]}|_a$. Se $E_a \in \mu_{[i,|\mu|-1]}|_a$, então somente o caso 2, acima, é verdadeiro. O caso 1 é verdadeiro quando $E_a \in \eta_{[0,j]}|_a$.

Lema 4 [PE para linguagem admissível do adversário] Seja P um protocolo em cascata balanceado, z um usuário e $\forall\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$. Então, γ tem a PE com respeito a z.

Prova. Podemos escrever $\gamma = \delta_0\delta_1 \ldots \delta_i$, onde $0 \leq i$ e $\delta_i \in \Sigma_1^*(z)$ ou $\delta_i \in \Sigma_2$. Por indução no número de termos δ_i de γ, temos:

BI. $\gamma = \lambda$ tem a PE com respeito a z, pois não existem usuários $x, y \in U$, conforme a **Definição 8**.

PI. Vamos supor que $\gamma = \delta_0\delta_1 \ldots \delta_i$ tenha a PE com respeito a z. Seja $\delta_{i+1} \in \Sigma_1^*(z)$ ou $\delta_{i+1} \in \Sigma_2$, tal que $\gamma' = \gamma\delta_{i+1}$. Pelo **Lema 2**, δ_{i+1} tem a PE com respeito a z. Considerando-se a hipótese de indução e o **Lema 3**, com $\mu = \gamma$ e $\eta = \delta_{i+1}$, temos que γ' tem a PE com respeito a z.

Lema 5 [PE de uma seqüência implica PE da normalização desta seqüência] Seja $\eta \in \Sigma^*$ e $z \in U$. Se η tem a PE com respeito a z, então $\bar{\eta}$ tem a PE com respeito a z.

Prova. Para a construção de $\bar{\eta}$, podemos supor a existência de um método iterativo que elimine, em cada *passo de normalização*, um par $\rho_u\rho_u^c$ de η, para todo $u \in U$, e onde $\rho_u = D_u$ ou $\rho_u = E_u$. Assim, podemos provar este lema por indução no número de passos para a construção de $\bar{\eta}$. Representaremos por η^i a aplicação de i passos de normalização a uma seqüência η.

BI. Se $\bar{\eta}$ é construído em zero passos, então $\eta = \bar{\eta}$. Assim, $\bar{\eta}$ tem a PE com respeito a z.

PI. Suponhamos que, para η^i $(i > 0)$, seja verdade que, se η^i tem a PE com respeito a z, então $\overline{\eta^i}$ tem a PE com respeito a z. No passo $i+1$ vale que $\overline{\eta^i} = \overline{\eta^{i+1}}$, então $\overline{\eta^{i+1}}$ tem a PE com respeito a z. Assim, basta mostrar que, se $\eta^{i+1} = \eta'\eta''$ tem a PE com respeito a z, então $\eta^i = \eta'\rho_u\rho_u^c\eta''$ tem a PE com respeito a z, ou seja, toda substring de $D_z\eta^iD_z$ é a-balanceada para todo usuário $a \in U \mid a \neq z$. Verificaremos, então, se as substrings de

η^i, que não existem em η^{i+1}, são a-balanceadas. Nessas substrings encontramos operadores ρ_u e/ou ρ_u^c. Quando $u = a$, o a-balanceamento é válido para η^i, uma vez que $\eta^i|_a = \eta^{i+1}|_a$. Para $u \neq a$, e sabendo que toda substring de $D_z\eta^{i+1}D_z$ é a-balanceada, temos como substrings de η^i, que não existem em η^{i+1} (consideremos que $i, j \in \mathbb{N} \mid i < |\eta'|$ e $j < |\eta''|$):

- $\eta'_{[i,|\eta'|-1]}\zeta E_u$ e $E_u\zeta\eta''_{[0,j]}$ (com $\zeta = \lambda$ ou $\zeta = D_u$): são a-balanceadas, pela definição de a-balanceamento.
- $\eta'_{[i,|\eta'|-1]}\zeta D_u$ e $D_u\zeta\eta''_{[0,j]}$: (com $\zeta = \lambda$ ou $\zeta = E_u$): a-balanceamento válido, pois, como $\eta'_{[i,|\eta'|-1]}$ e $\eta''_{[0,j]}$ são a-balanceadas, se $D_a \in \eta'_{[i,|\eta'|-1]}\zeta D_u|_a$ ou $D_a \in D_u\zeta\eta''_{[0,j]}|_a$, então estas duas seqüências têm a PB com respeito a a
- $\eta'_{[i,|\eta'|-1]}\rho_u\rho_u^c\eta''_{[0,j]}$: o a-balanceamento também é válido para esta subseqüência, pois $\eta'_{[i,|\eta'|-1]}\eta''_{[0,j]}$ é a-balanceada e $D_a \notin \rho_u\rho_u^c$.

Lema 6 [Balanceamento da linguagem admissível] Seja P um protocolo em cascata balanceado. Temos que, para um usuário $z \in$ U, $\forall\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$ e $\forall a \in U \mid a \neq z$, vale que $\bar{\gamma}$ tem a PB com respeito a a.

Prova. Pelo **Lema 4**, γ tem a PE. Além disso, pelo **Lema 5**, temos a PE para $\bar{\gamma}$. Deste modo, pela definição de PE, $D_z\bar{\gamma}D_z$ é A-balanceada para todo $a \in U \mid a \neq z$, porque $\exists x, y \in U \mid x, y = z$, com $D_z\bar{\gamma}D_{z_0} = D_x$ e $D_z\bar{\gamma}D_{z_{|D_z\bar{\gamma}D_z|-1}} = D_y$. Além disso, $\bar{\gamma} = D_z\bar{\gamma}D_{z_{[1,|D_z\bar{\gamma}D_z|-2]}}|_a$ contém D_a, então $\bar{\gamma}$ tem a PB com respeito a a. Assim, concluímos a prova.

4.2.4 Provando a segurança de protocolos em cascata

A seguir, provamos o teorema que estabelece as condições de segurança para um protocolo em cascata.

Teorema 1 [Segurança de Protocolos em Cascata] Um protocolo em cascata **P** é seguro se, e somente se,

(i) satisfaz a condição inicial de segurança, e
(ii) é balanceado.

A prova do **Teorema 1** divide-se em **A. Necessidade** e **B. Suficiência** e, antes ou durante o desenvolvimento de cada uma destas provas, enunciaremos lemas e definições necessários, sempre considerando, onde aplicável, quaisquer três usuários $x, y, z \in U$, distintos entre si.

A. Necessidade Se P é seguro, então (i) e (ii) são válidos. Provaremos a contra-recíproca desta implicação, ou seja, assumindo que (i) ou (ii) não valem, então P é inseguro.

I. (i) não vale, então P é inseguro.

Prova. Como (i) não vale, $P_0 \in \{D_x\}^*$. Assim, P é inseguro, uma vez que podemos fazer $\gamma = P_0^c \in \{E_x\}^* \in \Sigma_1^*(z)$, a fim de se obter $\overline{\gamma P_0} = \lambda$.

II. (ii) não vale, então P é inseguro.

Prova. Neste caso, a insegurança de P será mostrada por indução no comprimento de P_0, tomando-se como hipótese que $\exists \gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$, tal que $\overline{\gamma P_0} = \lambda$. Consideremos, portanto, o **Lema 7** a seguir.

Lema 7 [*Extração* de operador privado] Sejam dois usuários quaisquer $u, v \in U \mid u \neq v$. Se um passo de protocolo $\alpha\beta(u, v)$ é desbalanceado com respeito a u, então existem duas seqüências de operadores $\tau_1, \tau_2 \in \Sigma_1^*(v)$, tais que $\overline{\tau_1 \alpha\beta(u, v)\tau_2} = D_u$.

Prova. Pelo desbalanceamento, sabemos que $\alpha\beta \in \{D_u, E_v\}^*$. Então, $\alpha\beta^c \in \{E_u, D_v\}^* \in \Sigma_1^*(v) = \{E_u, E_v, D_v\}^*$ e, assim, para todo $0 \leq i \leq j < |\alpha\beta|$, temos $\alpha\beta_{[i,j]}^c \in \Sigma_1^*(v)$. Como, pelo desbalanceamento, $\exists 0 \leq k < |\alpha\beta|$ tal que $\alpha\beta_k =$

D_u , podemos fazer $\tau_1 = \alpha\beta^c_{[0,k-1]}$ e $\tau_2 = \alpha\beta^c_{[k+1,|\alpha\beta^c|-1]}$ e, deste modo, $\overline{\tau_1\alpha\beta(u,v)\tau_2} = D_u$.

Continuação da prova indutiva de **A. Necessidade** - II:

BI Se $|P_0(x,y)| = 1$, então temos três possibilidades para a formação de $P_0(x,y)$. Ou $P_0(x,y) = E_x$ ou $P_0(x,y) = E_y$ ou $P_0(x,y) = D_x$. Para o primeiro caso, considerando-se o **Lema 7**, pode-se obter $\gamma = D_x$. O segundo caso é análogo ao primeiro e no terceiro caso basta escolher $\gamma = E_x$.

PI Assume-se como verdadeiro que $\exists\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$, tal que $\overline{\gamma(P_0)_{[1,|P_0|-1]}} = \lambda$. Então, deve ser mostrado que $\exists\gamma' \in (\Sigma_1^*(z) \cup \Sigma_2)^*$, tal que $\overline{\gamma'\rho(P_0)_{[1,|P_0|-1]}} = \lambda$, sendo $P_0 = \rho(P_0)_{[1,|P_0|-1]}$ e $\rho \in \{E_x, E_y, D_x\}$. Assim, podem-se distinguir três casos:

a) $P_0 = E_x(P_0)_{[1,|P_0|-1]}$: O desbalanceamento de P ocorre em algum P_i $(0 < i < |P|)$. Vamos assumir que i seja par, ou seja, o desbalanceamento de $P_i(x,y)$ é com respeito a x. Pelo **Lema 7**, $\exists\, \tau_1, \tau_2 \in \Sigma_1(z)$ tais que $\overline{\tau_1 P_i(x,z)\tau_2} = D_x$. Assim, por hipótese de indução, o protocolo é inseguro, pois $\gamma' = \overline{\gamma\overline{\tau_1 P_i(x,z)\tau_2}E_x(P_0)_{[1,|P_0|-1]}} = \overline{\gamma(P_0)_{[1,|P_0|-1]}} = \lambda$. Para um valor ímpar de i, temos um desbalanceamento de $P_i(y,x)$ com respeito a y e a prova é análoga.

b) $P_0 = D_x(P_0)_{[1,|P_0|-1]}$: Neste caso, podemos fazer $\gamma' = \gamma E_x$ e, deste modo, concluir que o protocolo é inseguro por hipótese de indução, $\overline{\gamma' P_0} = \overline{\gamma E_x D_x(P_0)_{[1,|P_0|-1]}} = \overline{\gamma(P_0)_{[1,|P_0|-1]}} = \lambda$.

c) $P_0 = E_y(P_0)_{[1,|P_0|-1]}$: Este passo da prova é análogo ao do item a), acima.

B. Suficiência Sendo verdade (i) e (ii), temos que demonstrar que P é seguro. Suponhamos, por contradição, que (i) e (ii) valem, mas que P é inseguro, ou seja, $\exists\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$ tal que $\overline{\gamma P_i \ldots P_0} = \lambda$ $(0 \le i < |P|)$.

Lema 8 [P inseguro implica existe γ complemento de P_0] Se P é inseguro, então $\exists\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$ tal que $\bar{\gamma}^c = P_0(x,y)$.

Sub-Lema 8.1 [Inversos normais são complementos] Para qualquer par de seqüências de operadores normais δ e δ', vale que $\overline{\delta\delta'} = \lambda$ implica $\delta^c = \delta'$.

Prova. A prova é por indução no comprimento de δ.

BI Se $\delta = \lambda$, i.e., $|\delta| = 0$, então, por definição, $\delta^c = \lambda$. Como δ'é normal, $\overline{\lambda\delta'} = \delta' = \lambda$.

PI Para $\delta \neq \lambda$, suponhamos que seja verdadeiro para a subseqüência normal $\delta_{[0,|\delta|-2]}$; i.e., $\overline{\delta_{[0,|\delta|-2]}\,\delta'_{[1,|\delta'|-1]}} = \lambda$ implica $\delta^c_{[0,|\delta|-2]} = \delta'_{[1,|\delta'|-1]}$. Como $\overline{\delta\delta'} = \lambda$ e δ e δ' são normais, então $\delta^c_{|\delta|-1} = \delta'_0$. Logo, $\overline{\delta\delta'} = \overline{\delta_{[0,|\delta|-2]}\delta_{|\delta|-1}\delta'_0\delta'_{[1,|\delta'|-1]}} = \overline{\delta_{[0,|\delta|-2]}\,\delta'_{[1,|\delta'|-1]}} = \lambda$ e, por hipótese de indução, $\delta^c = \delta'$.

Prova do Lema 8: Sendo P inseguro, temos um $\gamma' \in (\Sigma_1^*(z) \cup \Sigma_2)^*$ tal que $\overline{\gamma' P_i \ldots P_0} = \lambda$, para algum $0 \leq i < |P|$. Como $P_i \ldots P_1 \in \Sigma_2^*$, podemos ter um $\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$ tal que $\gamma = \gamma' P_i \ldots P_1$. Assim, considerando o **Lema 1** com $\delta_1 = \lambda$, $\delta_2 = \gamma$ e $\delta_3 = P_0$, podemos escrever $\overline{\gamma' P_i \ldots P_0} = \overline{\gamma' P_i \ldots P_1 P_0} = \overline{\bar{\gamma} P_0} = \lambda$. Pelo **Sub-Lema 8.1**, e sabendo que P_0 é normal, temos que $\bar{\gamma}^c = P_0$.

Prova de **B. Suficiência**: Como o protocolo P é inseguro, então, pelo **Lema 8**, $\exists\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$ tal que $\bar{\gamma}^c = P_0$. Além disso, como P é balanceado, então, pelo **Lema 6**, $\bar{\gamma}$ tem a PB com respeito a x e y. A contradição entre esta última afirmativa e o fato de que $\bar{\gamma}^c = P_0$, pode ser mostrada considerando-se as duas situações seguintes:

a. $(P_0)_i = E_y$, para algum $0 \leq i < |P_0|$: Como P_0 é normal e $\bar{\gamma}^c = P_0$, $\bar{\gamma}_{|\bar{\gamma}|-1-i} = D_y$. E, como $\bar{\gamma}$ tem a PB com respeito a y, existe um j $(0 \leq j \neq i < |\bar{\gamma}|)$, tal que $\bar{\gamma}_j = E_y$. Isto implica em $(P_0)_{|P_0|-1-j} = D_y$. Mas, pela condição 3 da **Definição 1**, $D_y \notin P_0$, o que leva a uma contradição de que P é um protocolo em cascata, conforme a **Definição 2**.

b. $(P_0)_i \neq E_y$, para algum $0 \leq i < |P_0|$: Como P_0 é normal e não contém um D_y, então, todo $(P_0)_i = E_x$. Logo, $\bar{\gamma}_{|\bar{\gamma}|-1-i} = D_x$, o que contradiz o fato de que $\bar{\gamma}$ tem a PB com respeito a x.

Com base nas provas realizadas sobre as três diferentes formas de P_0, em **A. Necessidade**, podemos ilustrar como um adversário $z \in U$ é capaz de obter uma mensagem M da comunicação entre os usuários x e y, utilizando-se do

desbalanceamento do protocolo P. Como z é capaz de *escutar* o canal de comunicação, z intercepta $P_0(x,y)M$, objetivando extrair M. Seja P_i ($0 < i < |P|$) um passo de protocolo desbalanceado de P. Temos que $P_i(x,z)$ (em uma comunicação entre x e z) pode ser escrito na forma $\delta' D_x \delta''$, onde $\delta' \in \{E_z, D_x\}^*$ e $\delta'' \in \{E_z\}^*$. Observe que não importa se i é par ou ímpar, já que o termo $P_i(x,z)$ existe em ambos os casos, mudando apenas o usuário que inicia a comunicação usando P. z obtém M executando os três tipos de procedimentos a seguir, conforme o operador mais à esquerda de uma substring de $P_0(x,y)$ seja E_x, D_x ou E_y. Assim, um a um, os operadores de $P_0(x,y)$ são invertidos até que M esteja disponível. Sejam j e k naturais tais que $0 < j \leq k < |P_0|$.

- $P_{0_{[j,k]}} = E_x P_{0_{[j+1,k]}}$: z mantém uma comunicação com x, enviando mensagens quaisquer até chegar ao passo $i-1$ do protocolo. Neste passo, z envia a x $\tau_2 P_{0_{[j,k]}} M$, onde $\tau_2 = \delta''^c$. No passo i, a resposta de x é $P_i \tau_2 P_{0_{[j,k]}} M$. Assim, z aplica a este termo $\tau_1 = \delta'^c$, obtendo $\overline{\tau_1 P_i \tau_2 P_{0_{[j,k]}}} M = P_{0_{[j+1,k]}} M$, eliminando, então, o operador E_x.
- $P_{0_{[j,k]}} = D_x P_{0_{[j+1,k]}}$: Neste caso, z simplesmente aplica o operador E_x para obter $\overline{E_x P_{0_{[j,k]}}} M = P_{0_{[j+1,k]}} M$.
- $P_{0_{[j,k]}} = E_y P_{0_{[j+1,k]}}$: Análogo ao caso em que $P_{0_{[j,k]}} = E_x P_{0_{[j+1,k]}}$, exceto que a comunicação acontece entre y e z.

Assim, completamos a formalização analítica do modelo de segurança Dolev-Yao para protocolos em cascata.

Capítulo 5.

Formalização do Modelo *Dolev-Yao* em PVS

A formalização da segurança do modelo Dolev-Yao, em PVS, está dividida em uma estrutura hierárquica de 9 *teorias*, conforme a **Figura 3**. Esta divisão simplifica a visão das provas e especificações.

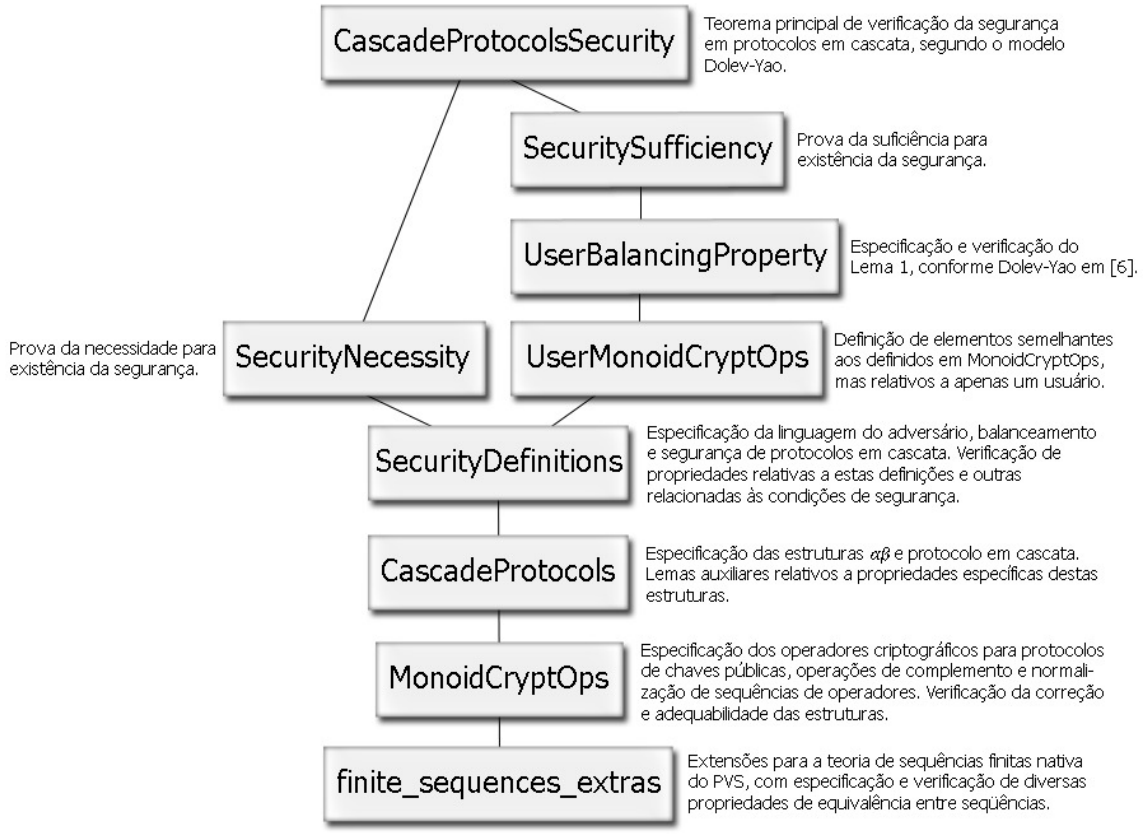

Figura 3: Teorias que compõem a formalização do modelo Dolev-Yao.

Descrevemos, a seguir, as teorias mais importantes da formalização do modelo Dolev-Yao. Alguns lemas utilizados durante as provas serão apenas citados, assim como

algumas fórmulas podem ser omitidas dos seqüentes para facilitar a compreensão. Consideraremos apenas os casos principais de uma prova, omitindo, quando conveniente, provas de *TCC*s gerados. Lemas e definições apresentados serão relacionados aos mostrados no capítulo anterior. Nos fragmentos de código a seguir, o símbolo `%` significa um comentário de linha, em PVS.

5.1 *Teoria* MonoidCryptOps: monóides em Σ^*

Para definir um operador E_x ou D_x ($\forall x \in U$), utilizamos as seguintes especificações:

```
U : TYPE+ % Non-empty type modeling a user.
cryptType : TYPE = {decrypt, encrypt} % The type of a crypt operation.
op : TYPE = [# crTyp : cryptType, user : U #] % Definition of an operator
                                               with a crypt type and an user.
seqOps : TYPE = finite_sequence[op] % A sequence of operators.
```

`seqOps` define um tipo de dados, que é uma seqüência finita contendo elementos do tipo `op`. Este é um tipo de registro contendo dois elementos: um tipo de operador `cryptType` e um usuário `U`. Em uma seqüência `seq`, do tipo `seqOps`, podemos acessar o elemento de uma posição $0 \leq$ `i` $<$ `seq`length`, escrevendo `seq1(i)`, e obter uma subseqüência com as posições de `i` a `j` ($0 \leq$ `i` $\leq$ `j` $<$ `seq`length`), através de `seq^(i,j)`.

Não é necessário formalizar a aplicação de uma seqüência de operadores a uma mensagem, já que a modelagem não considera os algoritmos criptográficos utilizados na transformação (encriptação ou decriptação) de uma mensagem, mas somente o protocolo.

Dados dois operadores `op1` e `op2`, do tipo `op`, define-se o predicado `areopcomplements?`, que retorna um valor booleano verdadeiro, caso os dois operadores sejam complementares. Os predicados `sameUsOp?` e `oppOp?` retornam valores verdadeiros caso, respectivamente, os usuários de `op1` e `op2` são iguais e os operadores de `op1` e `op2` são complementares. A partir destas especificações, podemos definir o predicado `areseqcomplements?`, que verifica se duas seqüências `seq1` e `seq2` são complementares.

```
oppOp?(op1, op2 : op) : bool = crTyp(op1)/=crTyp(op2)

sameUsOp?(op1, op2 : op) : bool = user(op1)=user(op2)

areopcomplements?(op1: op, op2: op) : bool =
                  sameUsOp?(op1, op2) AND oppOp?(op1, op2)
```

```
% Checks if two sequences are complements, considering they're symmetric,
% that is, an op1 in seq1(i) has a complement op2 in seq2(seq2`length - 1 - i),
% where seq1 and seq2 have the same length. For example, the sequences
% Ey,Dy,Ex and Dx,Dx,Ex are complements when normalized but they are not
% symmetric.
areseqcomplements?(seq1 : seqOps, seq2 : seqOps) : bool =
  IF seq1`length /= seq2`length THEN FALSE
  ELSE LET lth = seq1`length - 1 IN
    FORALL(i : nat | i < seq1`length) :
      areopcomplements?(seq1(i),seq2(lth - i))
  ENDIF
```

A verificação de que uma seqüência `seq` é normal, é feita pelo predicado `normalseq?`, que avalia se cada par de elementos de `seq` não é complementar, retornando verdadeiro neste caso.

```
% Checks if a sequence is normal, that is, its size is less than 2 or
% there is no two complementary operators in the sequence.
normalseq?(seq : seqOps) : bool =
  IF seq`length < 2 THEN TRUE
  ELSE FORALL (i : posnat | i < seq`length) :
    NOT ( areopcomplements?(seq(i-1),seq(i)) )
ENDIF
```

A partir da definição acima, podemos definir tipos de dados que representem seqüências normais (`normalseq`) ou redutíveis (`reduzibleseq`), como a seguir. Os tipos são definidos a partir dos predicados que modelam a propriedade correspondente.

```
normalseq: TYPE = (normalseq?) % A type describing a normal sequence.

% A sequence is reduzible when it is not normal.
reduzibleseq?(seq : seqOps): bool = NOT normalseq?(seq)

reduzibleseq: TYPE = (reduzibleseq?) % A reduzible sequence.
```

Para normalizar uma seqüência redutível `seq`, primeiramente definimos uma função recursiva `first_cancelable`, que retorna a posição do primeiro par normalizável de `seq`. O argumento desta função é uma seqüência de tipo `reduzibleseq`, definida acima. A utilização deste tipo de dados, definido a partir de um predicado, mostra que `first_cancelable` é definida para seqüências `seqOps` que satisfazem o predicado `reduzibleseq?`.

```
% A recursive function that finds the first cancelable position
% of a pair of complementary operators.
first_cancelable(seq : reduzibleseq) : RECURSIVE nat =
  IF areopcomplements?(seq(0),seq(1)) THEN 0
    ELSE 1 + first_cancelable(^(seq,(1,seq`length-1)))
  ENDIF
MEASURE seq`length-1
```

Com esta definição, podemos especificar a função de normalização `normalizeseq` de uma seqüência `seq`. Esta função, na primeira cláusula `IF`, verifica se a seqüência é normal; neste

caso, a normalização de `seq` é a própria seqüência `seq`. Senão, um par de operadores complementares é eliminado de `seq`, na posição determinada pela aplicação de `first_cancelable`.

```
% Definition of a recursive function that normalizes a sequence.
normalizeseq(seq : seqOps) : RECURSIVE seqOps =
IF normalseq?(seq) THEN seq
ELSE LET (firstCancPos : nat) = first_cancelable(seq) IN
  IF firstCancPos=0 THEN normalizeseq(seq^(2,seq`length-1)) ELSE
    normalizeseq(seq^(0,firstCancPos-1) o seq^(firstCancPos+2,seq`length-1))
  ENDIF
ENDIF
MEASURE seq`length
```

5.1.1 Lema geral de normalização

Com o lema `normalize_general`, a seguir, foi possível validar as estruturas de dados e funções utilizadas para representar os monóides em Σ^*. Este lema é a base para a prova do **Lema 1** em PVS, e mostra uma propriedade básica, mas não trivial, dos monóides em Σ^*.

```
normalize_general : LEMMA FORALL(seq : seqOps, i : nat | i < seq`length - 1) :
   areopcomplements?(seq(i),seq(i+1)) =>
   normalizeseq(seq) = normalizeseq(IF i = 0 THEN empty_seq
                            ELSE seq^(0,i-1) ENDIF o seq^(i+2,seq`length-1))
```

Este lema pode ser analiticamente escrito como $\overline{\delta_\iota \delta_{\iota+1}} \Rightarrow \bar{\delta} = \overline{\delta_{[0,\iota-1]} \delta_{[\iota+2,|\delta|-1]}}$, $\forall \delta \in \Sigma^*$ e $0 \leq i < |\delta| - 1$. A prova de `normalize_general` é por indução no comprimento de `seq`. O objetivo inicial é o seguinte:

```
   |-------
{1}   FORALL (seq: seqOps, i: nat | i < seq`length - 1):
       areopcomplements?(seq`seq(i), seq`seq(i + 1))
        =>
        normalizeseq(seq) =
         normalizeseq(IF i = 0 THEN empty_seq ELSE seq ^ (0, i - 1) ENDIF
                     o seq ^ (i + 2, seq`length - 1))
```

- `(measure-induct+ "seq`length" "seq")`: Indução no comprimento de `seq`, gerando um seqüente cuja premissa `{-1}` representará, após devida instanciação, a *hipótese de indução*. Mostraremos, no *passo de indução*, que o conseqüente `{1}` é verdadeiro.

```
{-1} FORALL (y: seqOps):
      FORALL (i: nat | i < y`length - 1):
       y`length < seq`length => areopcomplements?(y`seq(i), y`seq(i + 1))
          =>
        normalizeseq(y) =
          normalizeseq(IF i = 0 THEN empty_seq ELSE y ^ (0, i - 1) ENDIF
                       o y ^ (i + 2, y`length - 1))
  |-------
{1}   FORALL (i: nat | i < seq`length - 1):
       areopcomplements?(seq`seq(i), seq`seq(i + 1))
        =>
       normalizeseq(seq) =
        normalizeseq(IF i = 0 THEN empty_seq ELSE seq ^ (0, i - 1) ENDIF
                     o seq ^ (i + 2, seq`length - 1))
```

- `(skeep)`: Skolemização de `i` no sequente `{1}`. Abaixo, a hipótese de indução em `{-1}`, do seqüente anterior, foi omitida.

```
{-2}  areopcomplements?(seq`seq(i),
                        seq`seq(i + 1))
  |-------
{1}   normalizeseq(seq) =
      normalizeseq(IF i = 0 THEN empty_seq ELSE seq ^ (0, i - 1) ENDIF o
                   seq ^ (i + 2, seq`length - 1))
```

Considerando a primeira posição normalizável de `seq`, e as posições `i` e `i + 1` de `seq`, onde existem operadores complementares, dividimos a prova em três casos, apresentados a seguir. Referenciaremos o termo `first_cancelable(seq)`, a primeira posição normalizável, como `fc`.

1 `fc = i`. A prova para este caso é direta, bastando aplicar o lema `first_normalize`, onde provamos que $\texttt{fc} = \texttt{i} \rightarrow \overline{\delta} = \overline{\delta_{[0,\texttt{i}-1]}\delta_{[\texttt{i}+2,|\delta|-1]}}$, $\forall \delta \in \Sigma^*$. A prova para `first_normalize` segue, facilmente, das definições de `first_normalize` e `normalizeseq`.

2 `fc > i`. Por contradição, pode-se verificar que não é possível existir uma posição `i` anterior a uma posição `fc`.

3 `fc < i`. Parte principal da prova, onde verificaremos que $\overline{\delta_{[0,\texttt{fc}-1]}\delta_{[\texttt{fc}+2,|\delta|-1]}} = \overline{\delta_{[0,\texttt{i}-1]}\delta_{[\texttt{i}+2,|\delta|-1]}}$, $\forall \delta \in \Sigma^*$. Dividiremos esta prova em dois subcasos, `fc < i - 1` e `fc = i - 1`.

As três situações acima foram provadas, em PVS, conforme mostrado nos casos abaixo.

Caso 1:

- `(case "fc = i")`

```
[-1]  fc = i
  |-------
[1]   normalizeseq(seq) =
      normalizeseq(IF i = 0 THEN empty_seq ELSE seq ^ (0, i - 1) ENDIF o
                   seq ^ (i + 2, seq`length - 1))
```

- `(lemma first_normalize ("seq" "seq" "i" "i"))`: Com este comando, invocamos e instanciamos o lema `first_normalize`, obtendo o antecedente `{-1}`, abaixo.

```
{-1}  fc = i =>
      normalizeseq(seq) =
       normalizeseq(IF i = 0 THEN empty_seq ELSE seq ^ (0, i - 1) ENDIF o
                    seq ^ (i + 2, seq`length - 1))
```

Com um comando `(assert)` concluímos este caso da prova.

Fim do caso 1.

Caso 2:

- `(case "fc > i")`

```
[-1]  fc > i
[-2]  areopcomplements?(seq`seq(i), seq`seq(i + 1))
  |-------
```

- `(lemma characterization_f_canc)`: Invocando este lema e instanciando-o adequadamente, temos que $\overline{\delta_{[0,i]}} \wedge \left(\delta_i = \overline{\delta_{i+1}}\right) \rightarrow$ `fc` $=$ `i`, o que é mostrado em `{-1}`, a seguir.

```
{-1}  normalseq?(seq ^ (0, i)) AND
      areopcomplements?(seq`seq(i), seq`seq(i + 1))
      => fc = i
[-2]  fc > i
[-3]  areopcomplements?(seq`seq(i), seq`seq(i + 1))
  |-------
```

- `(assert)`: Simplificando o seqüente anterior, temos:

```
[-1]  fc > i
[-2]  areopcomplements?(seq`seq(i), seq`seq(1 + i))
  |-------
[1]   normalseq?(seq ^ (0, i))
```

A expressão em `[1]` é verdadeira, considerando-se `[-1]`. Conclui-se, então, este caso utilizando o lema `first_canc_greater_imp_normal` que prova o seguinte:

`fc` $\geq$ `i` $\rightarrow \delta_{[0,i]}$ é normal.

Fim do caso 2.

Caso 3:

Caso 3.1: `fc < i - 1`

Neste caso, obtemos um passo da prova onde os termos mais importantes são os abaixo apresentados. `[2]` é o termo que provaremos ser verdadeiro; em `[-4]` temos a hipótese de indução não instanciada.

```
{-1}  fc < i - 1
[-4]  FORALL (y: seqOps):
       FORALL (i: nat | i < y`length - 1):
         y`length < seq`length => areopcomplements?(y`seq(i), y`seq(1 + i))
          =>
           normalizeseq(y) =
            normalizeseq(IF i = 0 THEN empty_seq ELSE y ^ (0, i - 1) ENDIF
                         o y ^ (2 + i, y`length - 1))
[-5]  areopcomplements?(seq`seq(i), seq`seq(1 + i))
  |-------
[1]   seq`length <= 2
[2]   normalizeseq(IF fc = 0 THEN empty_seq ELSE seq ^ (0, fc - 1) ENDIF o
                   seq ^ (2 + fc, seq`length - 1))
      = normalizeseq(seq ^ (0, i - 1) o seq ^ (2 + i, seq`length - 1))
```

Em `[2]` é necessário considerar os casos onde `fc = 0` e `fc > 0`. Mostraremos apenas o segundo caso, cuja fórmula em `[2]`:

```
[2]   normalizeseq(seq ^ (0, fc - 1) o seq ^ (2 + fc, seq`length - 1))
      = normalizeseq(seq ^ (0, i - 1) o seq ^ (2 + i, seq`length - 1))
```

- `(inst -4 "seq ^ (0, fc - 1) o seq ^ (2 + fc, seq`length - 1)" "i - 2")`:

  Instanciando a hipótese de indução e realizando operações de reescrita adequadas, obtemos `[2]` na seguinte forma:

  `[2]  normalizeseq(seq1 ^ (0, i - 3) o seq1 ^ (i, seq1`length - 1))`
  `= normalizeseq(seq2 ^ (0, fc - 1) o seq2 ^ (2 + fc, seq2`length - 1))`, onde

  `seq1 = (seq ^ (0, fc - 1) o seq ^ (2 + fc, seq`length - 1))`, e
  `seq2 = (seq ^ (0, i - 1) o seq ^ (2 + i, seq`length - 1))`.

  Provando-se que os argumentos das funções `normalizeseq` acima são iguais, temos que `[2]` é verdadeiro.

  Fim do caso 3.1.

Caso 3.2: `fc = i - 1`

De modo semelhante ao do caso anterior, utiliza-se a hipótese de indução, obtendo

```
[-1]  fc = i - 1
[-3]  areopcomplements?(seq`seq(i), seq`seq(1 + i))
  |-------
[1]   seq`length <= 2
[2]   normalizeseq(IF fc = 0 THEN empty_seq ELSE seq ^ (0, fc - 1) ENDIF o
                   seq ^ (2 + fc, seq`length - 1))
       = normalizeseq(seq ^ (0, i - 1) o seq ^ (2 + i, seq`length - 1))
```

Para `fc = 0`, a prova é facilmente finalizada. Considerando somente o caso em que `fc > 0`, temos, em [2],

```
normalizeseq(seq ^ (0, i - 2) o seq ^ (1 + i, seq`length - 1)) =
       normalizeseq(seq ^ (0, i - 1) o seq ^ (2 + i, seq`length - 1)),
```

o que é verdadeiro, dados `[-1]` e `[-3]`.

Fim do caso 3.2.

Fim do caso 3.

Q.E.D.

A **Figura 4**, a seguir, mostra a árvore de prova do lema `normalize_general`. Os nós-pai de cada caso descrito na prova acima estão indicados na figura. A prova conta com 193 comandos.

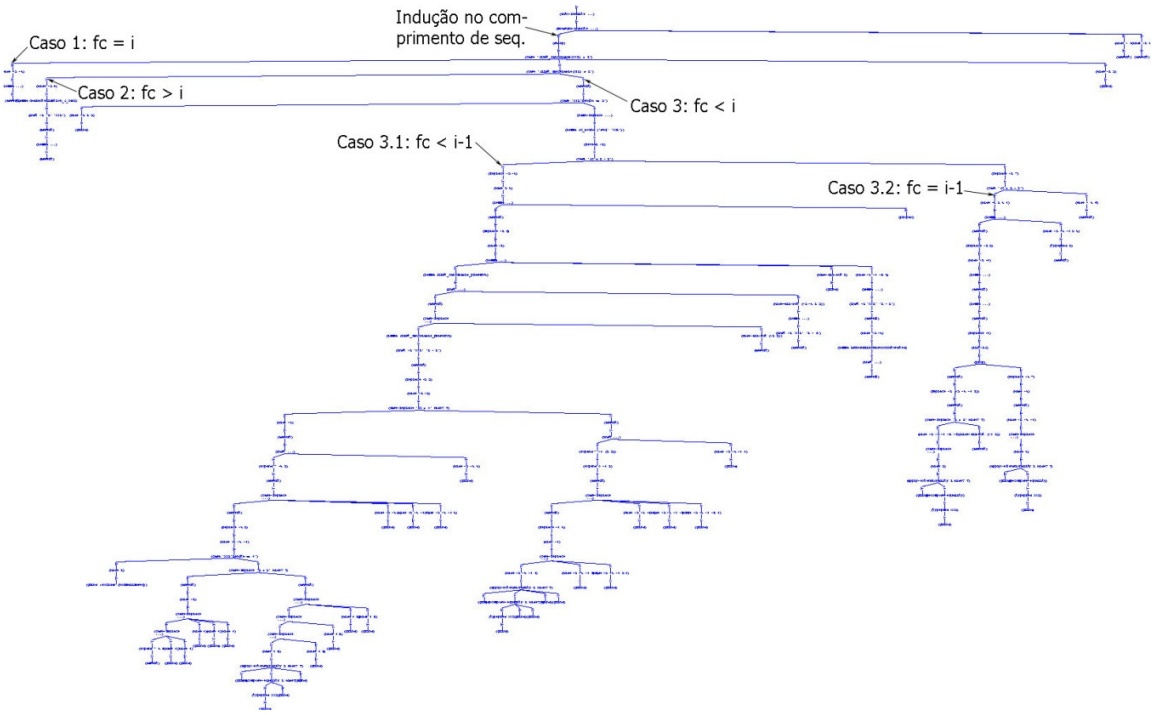

Figura 4: Árvore de prova do lema `normalize_general`.

5.1.2 Verificação do Lema 1

O **Lema 1**, apresentado no capítulo **Capítulo** 4, foi especificado como um corolário do lema `normalize_general`, conforme mostrado abaixo.

```
normalize_general_seq : COROLLARY FORALL (seq1, seq2, seq3 : seqOps) :
     normalizeseq(seq1 o seq2 o seq3) =
       normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

A prova é por indução no comprimento de `seq2`, com o seguinte objetivo:

```
  |-------
[1]   FORALL (seq1, seq2, seq3: seqOps):
       normalizeseq(seq1 o seq2 o seq3) =
        normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

- `(measure-induct+ "seq2`length" "seq2")`: A indução sobre `seq2`length` resulta no seqüente a seguir.

```
{-1}  FORALL (y: seqOps):
       FORALL (seq1, seq3: seqOps):
         y`length < seq2`length =>
          normalizeseq(seq1 o y o seq3) =
           normalizeseq(seq1 o normalizeseq(y) o seq3)
  |-------
{1}   FORALL (seq1, seq3: seqOps):
       normalizeseq(seq1 o seq2 o seq3) =
        normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

- `(skeep)`: Ao *skolemizar* `{1}`, obtemos no conseqüente:

```
{1}   normalizeseq(seq1 o seq2 o seq3) =
      normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

Para o caso onde `seq2` é normal, temos que `normalizeseq(seq2) = seq2`, e a fórmula `{1}` é trivialmente verdadeira. Caso contrário, considerando a definição de `normalseq?` e com $0 <$ `i` $<$ `seq2`length` podemos obter a fórmula abaixo como um antecedente.

```
{-1}  (areopcomplements?(seq2`seq(i - 1), seq2`seq(i)))
```

- `(lemma normalize_general ("seq" "seq1 o seq2 o seq3" "i" "i-1+seq1`length"))`: Utilizando o lema `normalize_general`, obtemos em `{-1}`:

```
{-1}  areopcomplements?((seq1 o seq2 o seq3)`seq(seq1`length - 1 + i),
                        (seq1 o seq2 o seq3)`seq(seq1`length + i))
      =>
      normalizeseq(seq1 o seq2 o seq3) =
       normalizeseq((seq1 o seq2 o seq3) ^ (0, seq1`length-2+i) o
         (seq1 o seq2 o seq3)^(1+seq1`length+i, (seq1 o seq2 o seq3)`length-1))
[-2]  (areopcomplements?(seq2`seq(i - 1), seq2`seq(i)))
  |-------
[3]   normalizeseq(seq1 o seq2 o seq3) =
      normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

Temos que `(seq1 o seq2 o seq3)`seq(seq1`length - 1 + i) = seq2`seq(i - 1)` e `(seq1 o seq2 o seq3)`seq(seq1`length + i) = seq2`seq(i)`. Assim, considerando `[-2]` obtém-se `{-1}` na seguinte forma:

```
{-1}  normalizeseq(seq1 o seq2 o seq3) =
       normalizeseq((seq1 o seq2 o seq3) ^ (0, seq1`length-2+i) o
         (seq1 o seq2 o seq3)^(1+seq1`length+i, (seq1 o seq2 o seq3)`length-1))
```

Substitui-se, então, esta última fórmula em `[3]` do seqüente anterior e, após considerar um ramo particular para o caso `i <= 1`, teremos o caso principal com `i > 1`. Assim, vale

```
(seq1 o seq2 o seq3) ^ (0, seq1`length - 2 + i) = seq1 o seq2 ^ (0, i - 2), e
(seq1 o seq2 o seq3) ^ (1 + seq1`length + i, (seq1 o seq2 o seq3)`length - 1) =
       seq2^(i+1,seq2`length-1) o seq3
```

E obtemos:

```
[-3]  (areopcomplements?(seq2`seq(i - 1), seq2`seq(i)))
  |-------
{5}  normalizeseq(seq1 o seq2^(0, i-2) o (seq2^(i+1, seq2`length-1) o seq3))
      = normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

Temos omitido, até este ponto da prova, a hipótese de indução, ainda não instanciada.

- `(inst -4 "seq2 ^ (0, i - 2) o seq2 ^ (i + 1, seq2`length - 1)" "seq1" "seq3")`:

  Instancia-se a hipótese de indução com a seqüência `seq2`, sem os operadores complementares das posições `i` e `i + 1`, obtendo

```
[-3]  (areopcomplements?(seq2`seq(i - 1), seq2`seq(i)))
[-4]  normalizeseq(seq1 o seq2^(0, i-2) o seq2^(1+i, seq2`length-1) o seq3) =
      normalizeseq(seq1 o normalizeseq(seq2 ^ (0, i - 2) o
                                seq2 ^ (i + 1, seq2`length - 1)) o seq3)
  |-------
{5}   normalizeseq(seq1 o seq2^(0, i-2) o seq2^(1+i, seq2`length-1) o seq3)
      = normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

Reescrevendo `{5}` com `[-4]`, obtém-se a fórmula

```
{5}  normalizeseq(seq1 o normalizeseq(seq2 ^ (0, i - 2) o
                          seq2 ^ (i + 1, seq2`length - 1)) o seq3) =
       normalizeseq(seq1 o normalizeseq(seq2) o seq3)
```

Considerando o antecedente `[-3]` do seqüente anterior, e pelo lema `normalize_general`, sabemos que

```
normalizeseq(seq2) = normalizeseq(seq2^(0, i-2) o seq2^(i+1, seq2`length-1))
```

Deste modo, `[5]` é verdadeiro e a prova está completa.

5.2 *Teoria* CascadeProtocols: protocolo em cascata

5.2.1 Passo de protocolo

Na **Definição 1**, mostramos a função que representa um passo de protocolo: $\alpha\beta: U \times U \rightarrow \Sigma^*$. Em PVS, esta função é especificada como `alphabeta : TYPE = [[U, U] -> seqOps]`. Para representar um passo de protocolo, primeiramente, define-se o predicado `abUsers?`, que modela a condição 4 da **Definição 1**.

```
abUsers?(ab : alphabeta, x, y : U) : bool = % condição 4 da definição 1
  FORALL(u, v : U) :
  ab(x,y)`length = ab(u,v)`length AND       % condição 4.1
    FORALL(i : nat | i < ab(x,y)`length) :  % condição 4.2
      (user(ab(x,y)(i)) = x OR user(ab(x,y)(i)) = y) AND
      (crTyp(ab(x,y)(i)) = crTyp(ab(u,v)(i))) AND
      (user(ab(x,y)(i)) = x IFF user(ab(u,v)(i)) = u) AND
      (user(ab(x,y)(i)) = y IFF user(ab(u,v)(i)) = v)
```

Uma vez definido `abUsers?`, temos o predicado `alphabeta_welldef?`, que retorna verdadeiro caso um `alphabeta` esteja bem definido para dois usuários quaisquer, segundo condições da **Definição 1**.

```
alphabeta_welldef?(ab : alphabeta, x, y : U) : bool =
  ab(x,y)`length > 0 AND                      % condição 1
  normalseq?(ab(x,y)) AND                     % condição 2
  ( FORALL(j : nat | j < ab(x,y)`length) :    % condição 3
      member(ab(x,y)(j),validSetxy(x,y)) ) AND
  abUsers?(ab, x, y)                          %condição 4
```

5.2.2 Protocolo em cascata

Um protocolo em cascata é definido como uma seqüência finita de elementos `alphabeta`, isto é, `protocol : TYPE = finite_sequence[alphabeta]`. Definimos, então, o predicado `protocol_welldef?` que, dado um protocolo em cascata do tipo `protocol`, retorna verdadeiro caso este protocolo seja *bem-definido*, ou seja, é tal como mostrado pela **Definição 2**. Além disso, definimos também um tipo de dados `welldefined_protocol`, baseado no predicado `protocol_welldef?`.

```
protocol_welldef?(prot : protocol) : bool =
  prot`length > 0 AND
  FORALL(x : U, y : U | x /= y) : FORALL(i : nat | i < prot`length) :
    IF even?(i) THEN alphabeta_welldef?(prot(i),x,y)
    ELSE alphabeta_welldef?(prot(i),y,x) % odd?(i)
    ENDIF
```

```
welldefined_protocol: TYPE = (protocol_welldef?)
```

Em algumas definições ou lemas, um protocolo em cascata deve ser tratado como uma seqüência de operadores criptográficos, para que seja possível aplicar as especificações mostradas na seção **5.1**. Assim, definimos a função recursiva `extract_eN`, que *lineariza* os passos de um protocolo em cascata `prot`, de modo que se forme uma única seqüência de operadores criptográficos.

```
extract_eN(prot : protocol, i : below[prot`length], x, y : U) :
  RECURSIVE seqOps =
    IF i = 0 THEN prot(0)(x,y)
      ELSE (prot(i)(x,y) o extract_eN(prot, i-1, x, y))
    ENDIF
  MEASURE i
```

5.3 *Teoria* SecurityDefinitions: Linguagem do Adversário e Definições de Segurança

Nesta seção, especificamos as estruturas e funções que definem a linguagem admissível do adversário (**Definição 3**), propriedade de balanceamento (**Definição 4**), protocolo balanceado (**Definição 5**), condição inicial de segurança (**Definição 6**) e a noção de protocolo seguro (**Definição 7**).

5.3.1 Especificação da Definição 3: Linguagem admissível do adversário

O tipo de dados utilizado para representar os termos construídos por um adversário é `gammaT : TYPE = finite_sequence[seqOps]`. A linguagem admissível do adversário é definida no predicado `gamma_welldef?`, que retorna verdadeiro caso um termo do tipo `gammaT`, construído pelo adversário `z`, esteja em $(\Sigma_1(z)^* \cup \Sigma_2)^*$. A definição da linguagem $\Sigma_1(z)^*$ é dada pela função `sigma1`, que retorna o conjunto de operadores disponíveis ao usuário `z`. Em `wellDefInSigma1`, temos uma função que retorna se uma seqüência de operadores `seq` está na linguagem $\Sigma_1(z)^*$. A linguagem Σ_2 é especificada por `sigma2_3`, que retorna, a partir de um protocolo em cascata bem definido `prot`, um conjunto com termos de `prot` acessíveis a `z`.

```
sigma1(z : U) : setof[op] = {op1 : op | crTyp(op1) = encrypt OR user(op1) = z}
```

```
sigma2_3(prot : protocol) : setof[seqOps] =
  {ab : seqOps | EXISTS(i : posnat | i < prot`length) :
                 EXISTS(a : U, b : U | a /= b) : ab = prot(i)(a,b) }

wellDefInSigma1(seq : seqOps, z : U) : bool = FORALL(i : nat | i < seq`length) :
                                              member(seq(i), sigma1(z))

gamma_welldef?(prot : welldefined_protocol, gamma : gammaT, z : U) :
  bool = FORALL(i : nat | i < gamma`length) :
    ( member(gamma(i),sigma2_3(prot)) OR
      wellDefInSigma1(gamma(i),z) )
```

Dada uma estrutura `gammaT`, a função `extract_gamma`, a seguir, *lineariza* essa estrutura, retornando uma seqüência de operadores.

```
extract_gamma(gamma : gammaT) :
   RECURSIVE seqOps =
     IF gamma`length = 0 THEN empty_seq
     ELSE (gamma(0) o extract_gamma(gamma^(1,gamma`length-1)))
     ENDIF
   MEASURE gamma`length
```

5.3.2 Especificação da Definição 4: PB

Sejam dois usuários distintos `x` e `y` trocando uma estrutura `alpbet`, do tipo `alphabeta`. A PB para `alpbet` foi especificada na forma do predicado `balanced_wrt?`, que retorna verdadeiro quando `alpbet` tem a PB com respeito a `x` ou `y`. Para se afirmar que um operador de `alpbet` é, por exemplo, um D_z, utiliza-se a expressão `(crTyp(alpbet(x,y)(i)) = decrypt AND user(alpbet(x,y)(i)) = z)`, ou seja, verifica-se se o tipo de operador de `alpbet(x,y)(i)` é `decrypt` e o usuário é `z`. Apesar de a **Definição 4** mostrar uma definição genérica para a PB, é suficiente defini-la em termos de estruturas `alphabeta`, já que estaremos tratando sempre de protocolos em cascata em nossa especificação.

```
balanced_wrt?(alpbet : alphabeta, x : U, y : U | x/=y,  z : U | z=x OR z=y) :
  boolean =
    ( EXISTS (i : nat | i < alpbet(x,y)`length) :
      ( crTyp(alpbet(x,y)(i)) = decrypt AND user(alpbet(x,y)(i)) = z))
        =>
    ( EXISTS (j : nat | j < alpbet(x,y)`length) :
      ( crTyp(alpbet(x,y)(j)) = encrypt AND user(alpbet(x,y)(j)) = z))
```

5.3.3 Especificação da Definição 5: Protocolo balanceado

Um protocolo bem definido é balanceado sempre que cada termo `alphabeta` do protocolo possuir a PB com respeito ao usuário que *construiu* esse termo. Se um `alphabeta` está em uma posição `i` par (`even?(i)`) da seqüência do protocolo, então o balanceamento

deve acontecer com respeito a `x`, ou seja, `balanced_wrt?(prot(i),x,y,x)`. Analogamente, para o caso em que `i` é ímpar.

```
balanced_cascade_protocol?(prot : welldefined_protocol) : boolean =
  FORALL(x : U ,y : U | x /= y, i : posnat | i < prot`length) :
    IF even?(i) THEN balanced_wrt?(prot(i),x,y,x)
    ELSE balanced_wrt?(prot(i),y,x,y)
    ENDIF
```

5.3.4 Especificação da Definição 6: Condição inicial de segurança

Dados um protocolo em cascata em definido `prot` e dois usuários `x` e `y`, definimos a condição inicial de segurança como o predicado `alpha0ContainsE?`, que retorna verdadeiro caso exista um operador com tipo `encrypt` em `prot(0)`.

```
alpha0ContainsE?(prot : welldefined_protocol, x : U, y : U) : bool =
  EXISTS(i : nat | i < prot(0)(x,y)`length) : member(prot(0)(x,y)(i),
                     defineopset(encrypt, {us : U | us=x OR us=y}))
```

Assim, a condição inicial de segurança é satisfeita quando existir um `i` menor que o comprimento de `prot(0)`, tal que o operador em `prot(0)(x,y)(i)` pertença ao conjunto definido por `defineopset`. Esta função retorna um conjunto de operadores de um tipo específico para cada usuário de um conjunto de usuários. Neste caso, `defineopset` retorna o conjunto $\{E_x, E_y\}$.

5.3.5 Especificação da Definição 7: Protocolo seguro

Definimos os conceitos de protocolo seguro e inseguro, conforme abaixo. As funções `extract_gamma` e `extract_eN` foram devidamente utilizadas para gerarem seqüências de operadores que, após concatenadas, são normalizadas e comparadas com a seqüência vazia `empty_seq`, para determinar se o protocolo em cascata bem definido `prot` é ou não seguro.

```
secure_protocol?(prot : welldefined_protocol, x : U, y : U | x /= y, z : U) :
  bool =
    FORALL (gamma : gammaT | gamma_welldef?(prot, gamma, z),
            i : nat | i < prot`length) :
    (normalizeseq(extract_gamma(gamma) o extract_eN(prot, i, x, y))
       /= empty_seq)

insecure_protocol?(prot : welldefined_protocol, x : U, y : U | x /= y, z : U) :
  bool =
    EXISTS (gamma : gammaT | gamma_welldef?(prot,gamma,z),
            i : nat | i < prot`length) :
      normalizeseq(extract_gamma(gamma) o extract_eN(prot,i,x,y)) = empty_seq
```

O lema a seguir estabelece que efetivamente a negação do predicado `secure_protocol?` é o predicado `insecure_protocol?`, mas não mostraremos aqui a prova.

```
secure_eq_not_insecure_prot : LEMMA
  FORALL(prot : welldefined_protocol, x : U, y : U | x /= y, z : U) :
    secure_protocol?(prot,x,y,z) IFF NOT insecure_protocol?(prot,x,y,z)
```

Para a segurança de protocolos, mostramos a especificação de dois predicados que definem os conceitos logicamente opostos de segurança e insegurança. Além disso, provamos um lema que estabelece a relação entre estes predicados. A mesma idéia foi aplicada às definições da PB e de protocolo balanceado, porém, omitimos aqui os lemas de relacionamento entre os pares correspondentes destas definições.

5.4 *Teoria* SecurityNecessity: Prova da necessidade do Teorema 1

Conforme descrito no capítulo **Capítulo** 4, seção **A. Necessidade**, a prova da necessidade para o **Teorema 1** consiste em demonstrar a contra-recíproca de: *se um protocolo bem definido é seguro, então valem as propriedades* (i) *e* (ii). Para esta prova, expressaremos primeiramente os lemas fundamentais mais importantes utilizados na prova final da necessidade. Em alguns casos, lemas serão descritos apenas com relação a suas funcionalidades em uma parte de qualquer prova.

Na seção **5.4.1**, a seguir, provaremos o lema `secProt_imp_alpha0ContainsE`, que corresponde à prova mostrada em **A. Necessidade-I**. E, na seção **5.4.2**, provaremos o lema `secure_impl_balanced`, correspondente à prova **A. Necessidade-II**.

5.4.1 Verificação: Protocolo em cascata *P* seguro → Condição inicial de segurança.

O lema `secProt_imp_alpha0ContainsE` é enunciado abaixo.

```
secProt_imp_alpha0ContainsE : LEMMA FORALL (prot : welldefined_protocol,
                                            x : U, y : U | x /= y, z : U) :
  secure_protocol?(prot, x, y, z) => alpha0ContainsE?(prot, x, y)
```

A seguir, temos o objetivo a ser provado por contra-recíproca.

```
  |-------
[1]   FORALL (prot: welldefined_protocol, x: U, y: U | x /= y, z: U):
       secure_protocol?(prot, x, y, z) => alpha0ContainsE?(prot, x, y)
```

- `(skeep)`: Skolemizando `[1]` e expandindo a definição de `secure_protocol?`:

```
{-1}  FORALL (gamma: gammaT | gamma_welldef?(prot, gamma, z),
            i: nat | i < prot`length):
       (normalizeseq(extract_gamma(gamma) o extract_eN(prot, i, x, y)) /=
         empty_seq)
  |-------
[1]   alpha0ContainsE?(prot, x, y)
```

- `(inst -1 "defgammawithcomplementseq(prot`seq(0)(x, y))" "0")`: Neste comando, a função `defgammawithcomplementseq` contrói um `gammaT` com uma única seqüência de operadores, que é o complemento do primeiro passo de protocolo de `prot`. Esta instanciação gera dois ramos principais, I e II, a seguir. Chamaremos de `gamComp`, o termo `defgammawithcomplementseq(prot`seq(0)(x, y))`.

I.
```
{-2}  normalizeseq(extract_gamma(gamComp) o prot`seq(0)(x, y))
         /= empty_seq
   |-------
```

II.
```
|-------
 {1}   gamma_welldef?(prot, gamComp, z)
 [2]   alpha0ContainsE?(prot, x, y)
```

- Em I, temos que a fórmula `{-2}` é falsa, pois a normalização mostrada resulta na seqüência vazia, pois `extract_gamma(gamComp)` = $\overline{\texttt{prot`seq(0)(x, y)}}$.

  Resta provar o seqüente em II, cuja prova é dada a seguir.

- Em II, a negação lógica de `alpha0ContainsE?`, que aparece na fórmula `[2]`, é o predicado `Nalpha0ContainsE?`. O lema `alpha0FunctionsEq` prova a equivalência `Nalpha0ContainsE?`↔`NOT alpha0ContainsE?`. Então, considerando este lema temos:

```
[-1]  Nalpha0ContainsE?(prot, x, y)
  |-------
{1}   gamma_welldef?(prot, gamComp, z)
```

Conforme podemos ver no seqüente acima, devemos demonstrar que `gamComp` é bem definido, dada a não existência de operadores do tipo `encrypt` em `prot`seq(0)(x, y)`. Demonstrar que a fórmula `{1}` é verdadeira equivale a demonstrar que `prot` é inseguro, o que fizemos em I. Assim, o seqüente acima pode ser visto como a contra-recíproca do objetivo inicial da prova deste lema, ou seja,

`Nalpha0ContainsE?(prot, x, y)` → `insecure_protocol?(prot, x, y, z)`.

- No seqüente anterior, em `{1}`, vamos provar que `gamComp` $\in \Sigma_1^*(z)$, isto é,

```
[-1]  Nalpha0ContainsE?(prot, x, y)
  |-------
{1}   crTyp(gamComp`seq(i)`seq(j)) = encrypt
{2}   user(gamComp`seq(i)`seq(j)) = z
```

- A fórmula `[-1]` implica na existência somente de operadores D_x em `prot`seq(0)(x, y)`. Expandindo `Nalpha0ContainsE?`, obtemos:

```
[-2]  FORALL (k: nat | k < prot`seq(0)(x, y)`length):
       prot`seq(0)(x, y)`seq(k)`crTyp = decrypt AND
        prot`seq(0)(x, y)`seq(k)`user = x
  |-------
[1]   crTyp(gamComp`seq(i)`seq(j)) = encrypt
[2]   user(gamComp`seq(i)`seq(j)) = z
```

  Como `gamComp` é o complemento de `prot`seq(0)(x, y)`, e `prot`seq(0)(x, y)` possui somente operadores D_x, então `k`, em `[-2]`, deve ser instanciado com o valor `prot`seq(0)(x, y)`length` $-$ `1` $-$ `j`. Assim, concluímos a prova desta seção.

5.4.2 Verificação: Protocolo em cascata P seguro $\rightarrow$ P balanceado.

Nesta seção, mostraremos inicialmente três lemas fundamentais, dos quais depende o lema principal, `secure_impl_balanced`. Em ordem de dependência, os lemas a serem apresentados são:

```
unbalanced_implies_complement_in_sigma1,
 └extractable_decUser,
   └unbalanced_impl_insecure_At_0 e
     └secure_impl_balanced.
```

Lema `unbalanced_implies_complement_in_sigma1`: Em parte da prova do **Lema 7**, no capítulo **Capítulo** 4, assumimos que, sendo um passo de protocolo `ab(x,y)` desbalanceado com respeito a `y`, então o complemento de toda sub-palavra de `ab(x,y)` está em $\Sigma_1^*(y)$. Com o lema `unbalanced_implies_complement_in_sigma1` provamos a validade desta afirmação, mas não apresentaremos a prova aqui.

```
unbalanced_implies_complement_in_sigma1 : LEMMA FORALL(x : U, y : U | x /= y,
        ab : alphabeta | alphabeta_welldef?(ab, x, y),
        i : nat, j : below[ab(x,y)`length] | i <= j) :
  unbalanced_wrt?(ab, x, y, x) =>
  wellDefInSigma1?(complementseq(ab(x, y)^(i, j)), y)
```

No enunciado do lema acima, identificamos o seguinte:

- `unbalanced_wrt?`: predicado logicamente oposto a `balanced_wrt?`.

- `wellDefInSigma1?(tau : seqOps, u : U)`: predicado que retorna verdadeiro caso `tau` $\in \Sigma_1^*(u)$.
- `complementseq(seq : seqOps)`: função que retorna o complemento de uma seqüência `seq`.

Este lema é necessário para a prova do lema `extractable_decUser`, a seguir.

Lema `extractable_decUser`: Corresponde ao **Lema 7** do capítulo **Capítulo** 4. Abaixo, apresentamos o enunciado do lema, seguido pelo objetivo que será demonstrado.

```
extractable_decUser : LEMMA FORALL(x : U, y : U | x /= y,
                  ab : alphabeta | alphabeta_welldef?(ab, x, y)) :
 unbalanced_wrt?(ab, x, y, x) =>
 EXISTS (tau1, tau2 : seqOps) :
   wellDefInSigma1?(tau1, y) AND wellDefInSigma1?(tau2, y) AND
   normalizeseq(tau1 o ab(x,y) o tau2) = defalphbetawithop(opDef(decrypt,x), 1)

  |-------
{1}   FORALL (x: U, y: U | x /= y,
            ab: alphabeta | alphabeta_welldef?(ab, x, y)):
       unbalanced_wrt?(ab, x, y, x) =>
        (EXISTS (tau1, tau2: seqOps):
          wellDefInSigma1?(tau1, y) AND wellDefInSigma1?(tau2, y) AND
           normalizeseq(tau1 o ab(x, y) o tau2) =
            defalphbetawithop(opDef(decrypt, x), 1))
```

Onde, `defalphbetawithop(oper : op, n : nat)` é uma função que cria um passo de protocolo (de tipo `alphabeta`) com `n` operadores `op`.

- `(skeep)`: Após a *skolemização* de `x`, `y` e `ab` no objetivo mostrado acima, obtemos o seqüente:

```
{-1}  unbalanced_wrt?(ab, x, y, x)
{-2}  ab(x, y)`seq(i)`crTyp = decrypt
{-3}  ab(x, y)`seq(i)`user = x
  |-------
{1}   EXISTS (tau1, tau2: seqOps):
       wellDefInSigma1?(tau1, y) AND wellDefInSigma1?(tau2, y) AND
         normalizeseq(tau1 o ab(x, y) o tau2) =
          defalphbetawithop(opDef(decrypt, x), 1)
```

  `{-2}` e `{-3}` são inferidas a partir do desbalanceamento em `{-1}`, ou seja, existe uma posição $0 \leq i <$ `ab(x, y)`length, em `ab(x, y)`, onde existe um operador D_x.

- `(lemma unbalanced_implies_complement_in_sigma1)`: Considerando `{-1}` acima, utilizamos por duas vezes o lema `unbalanced_implies_complement_in_sigma1`, para gerar as fórmulas `{-1}` e `{-2}` a seguir.

```
{-1}  wellDefInSigma1?(complementseq(ab(x, y)^(0, i - 1)), y)
{-2}  wellDefInSigma1?(complementseq(ab(x, y)^(1 + i, ab(x, y)`length - 1)), y)
```

```
{-3}  ab(x, y)`seq(i)`crTyp = decrypt
{-4}  ab(x, y)`seq(i)`user = x
  |-------
{1}   EXISTS (tau1, tau2: seqOps):
       wellDefInSigma1?(tau1, y) AND wellDefInSigma1?(tau2, y) AND
         normalizeseq(tau1 o ab(x, y) o tau2) =
          defalphbetawithop(opDef(decrypt, x), 1)
```

Representados nas fórmulas `{-1}` e `{-2}` acima, temos $\tau_1 \in \Sigma_1^*(y)$ e $\tau_2 \in \Sigma_1^*(y)$, respectivamente, conforme definição de τ_1 e τ_2 na prova analítica do **Lema 7**.

- No seqüente anterior, considerando `{-1}` e `{-2}` e instanciando `[1]` com τ_1 e τ_2 descritos acima, obtemos

```
{-1}  ab(x, y)`seq(i)`crTyp = decrypt
{-2}  ab(x, y)`seq(i)`user = x
  |-------
{1}  normalizeseq(complementseq(ab(x, y)^(0, i - 1)) o ab(x, y) o
              complementseq(ab(x, y)^(1 + i, ab(x, y)`length - 1))) =
                defalphbetawithop(opDef(decrypt, x), 1)
```

A função de normalização em `{1}`, claramente, resulta em uma seqüência com apenas um termo, `ab(x, y)`seq(i)`, que é igual a um operador $D_x$, conforme as premissas em `{-1}` e `{-2}`. Então, concluímos a prova do lema `extractable_decUser`.

Lema `unbalanced_impl_insecure_At_0`: Neste lema, mostramos a contra-recíproca da prova principal desta seção: Se um protocolo bem definido `prot` é desbalanceado, então existe um γ bem definido, tal que $\overline{\gamma_{\texttt{prot(0)(x,y)}}} = \lambda$. A prova é por indução no comprimento de `prot(0)`.

```
unbalanced_impl_insecure_At_0 : LEMMA FORALL (prot : welldefined_protocol,
                    x : U, y : U | x /= y, z : U | z /= x AND z /= y) :
  unbalanced_cascade_protocol?(prot) =>
    EXISTS (gamma : gammaT | gamma_welldef?(prot,gamma,z)) :
      normalizeseq(extract_gamma(gamma) o prot(0)(x,y)) = empty_seq
```

O objetivo a ser provado é o seguinte:

```
  |-------
[1]   FORALL (prot: welldefined_protocol, x: U, y: U | x /= y,
            z: U | z /= x AND z /= y):
       unbalanced_cascade_protocol?(prot) =>
        (EXISTS (gamma: gammaT | gamma_welldef?(prot, gamma, z)):
           normalizeseq(extract_gamma(gamma) o prot(0)(x,y))
            = empty_seq)
```

- `(measure-induct+ "length(x, y, prot(0))" ("prot" "x" "y"))`: Com este comando iniciamos a prova por indução no comprimento de `prot(0)`.

```
{-1}  FORALL (hProt: welldefined_protocol, u: U, v: {y: U | u /= y}):
       FORALL (z: U | z /= u AND z /= v):
         length(u, v, hProt(0)) < length(x, y, prot(0))
          IMPLIES
          unbalanced_cascade_protocol?(hProt) =>
           (EXISTS (gamma: gammaT | gamma_welldef?(hProt, gamma, z)):
              normalizeseq(extract_gamma(gamma) o hProt(0)(u, v)) = empty_seq)
  |-------
{1}   FORALL (z: U | z /= x AND z /= y):
       unbalanced_cascade_protocol?(prot) =>
        (EXISTS (gamma: gammaT | gamma_welldef?(prot, gamma, z)):
          normalizeseq(extract_gamma(gamma) o prot(0)(x,y))
           = empty_seq)
```

Omitiremos a hipótese de indução, em `{-1}`, de alguns seqüentes a seguir, para melhor visualização.

- `(lemma first_crypt_elem)`: Com este lema devidamente instanciado, temos que o primeiro operador de `prot(0)(x,y)` pertence ao conjunto $\{D_x, E_y, E_x\}$. Assim, consideraremos um caso para cada um destes três possíveis valores.

I. `prot(0)(x,y)(0) = Ex`: Representaremos esta igualdade como `isEx`.

- Sendo `prot` desbalanceado, temos uma posição $0 <$ `i` $<$ `prot`length` de `prot`, onde ocorre o desbalanceamento. Podemos supor, usando o comando `case`, que o desbalanceamento seja para um usuário `x` do protocolo, ou seja, `unbalanced_wrt?(prot`seq(i), x, z, x)`.
- `(lemma extractable_decUser)`: Com este lema, em `{-1}` a seguir, e considerando a descrição acima temos:

```
{-1}  FORALL (x: U, y: U | x/=y, ab: alphabeta | alphabeta_welldef?(ab, x, y)):
       unbalanced_wrt?(ab, x, y, x) =>
        (EXISTS (tau1, tau2: seqOps):
          wellDefInSigma1?(tau1, y) AND wellDefInSigma1?(tau2, y) AND
            normalizeseq(tau1 o ab(x, y) o tau2) =
             defalphbetawithop(opDef(decrypt, x), 1))
[-2]  unbalanced_wrt?(prot`seq(i), x, z, x)
[-3]  isEx
  |-------
[4]   EXISTS (gamma: gammaT | gamma_welldef?(x!1, gamma, z)):
       normalizeseq(extract_gamma(gamma) o prot`seq(0)(x, y)) = empty_seq
```

A fórmula `{-3}` representa o caso a ser tratado neste ramo da prova do lema `unbalanced_impl_insecure_At_0`. Em `[4]` está expressa a existência de um `gammaT` que normaliza o primeiro passo de protocolo de `prot` à seqüência vazia.

- `(inst -1 "x" "z" "prot`seq(i)")`: Instanciamos `{-1}` do seqüente anterior com os usuários `x` e `z` e um passo de protocolo na posição `i` de `prot`. Assim, considerando `{-2}`, teremos em `{-1}`:

```
{-1}  EXISTS (tau1, tau2: seqOps):
       wellDefInSigma1?(tau1, z) AND wellDefInSigma1?(tau2, z) AND
         normalizeseq(tau1 o prot`seq(i)(x, z) o tau2) =
          defalphbetawithop(opDef(decrypt, x), 1)
```

Skolemizando a fórmula acima, e aplicando uma *simplificação disjuntiva* com o comando `flatten`, teremos três novas fórmulas no seqüente, `{-1}`, `{-2}` e `{-3}`, como mostradas a seguir:

```
{-1}  wellDefInSigma1?(tau1, z)
{-2}  wellDefInSigma1?(tau2, z)
{-3}  normalizeseq(tau1 o prot`seq(i)(x, z) o tau2) =
      defalphbetawithop(opDef(decrypt, x), 1)
[-4]  isEx
  |-------
[4]   EXISTS (gamma: gammaT | gamma_welldef?(x!1, gamma, z)):
       normalizeseq(extract_gamma(gamma) o prot`seq(0)(x, y)) = empty_seq
```

- `(lemma triplet_welldef)`: Este lema, que não provaremos aqui, é enunciado como mostrado abaixo.

```
triplet_welldef : LEMMA FORALL (prot : welldefined_protocol,
                          tau1, tau2 : seqOps, a : U, b : U | a /= b, z : U,
                          i:posnat | i < prot`length) :
  wellDefInSigma1?(tau1, z) AND wellDefInSigma1?(tau2, z) =>
  gamma_welldef?(prot, defgammawithtriplet(tau1, prot(i)(a, b), tau2), z))
```

A função `defgammawithtriplet` realiza a concatenação das três seqüências para constituir um `gammaT`. Com o lema `triplet_welldef`, provamos o seguinte: Para todo $\tau_1, \tau_2 \in \Sigma_1^*(z)$ (`wellDefInSigma1?(tau1, z) AND wellDefInSigma1?(tau2, z)`), temos que $\tau_1 P_i(a,b) \tau_2$ (`defgammawithtriplet(tau1, prot(i)(a, b), tau2)`) é um `gammaT` bem definido. $P =$ `prot`, `i`, `a`, `b` e `z` são conforme mostrados no enunciado do lema. Após a correta instanciação de `triplet_welldef` teremos, considerando `{-1}` e `{-2}` do seqüente anterior:

```
gamma_welldef?(prot, defgammawithtriplet(tau1, prot`seq(i)(x, z), tau2), z)
```

- `(expand isEx)`: A expansão de `isEx` produz $P_0(x,y) = E_x P_0(x,y)_{[1,|P_0(x,y)|-1]}$.

```
prot`seq(0)(x, y) = defalphbetawithop(opDef(encrypt, x), 1) o
   prot`seq(0)(x, y)^(1, prot`seq(0)(x, y)`length - 1)
```

Assim, reescrevemos `prot`seq(0)(x, y)` na fórmula `[4]` do seqüente anterior:

```
[4] EXISTS (gamma: gammaT | gamma_welldef?(x!1, gamma, z)):
  normalizeseq(extract_gamma(gamma) o (defalphbetawithop(opDef(encrypt,x),1) o
      prot`seq(0)(x, y)^(1, prot`seq(0)(x, y)`length - 1))) = empty_seq
```

- `(inst -1 "suff_ab_prot(prot,1)" "x" "y" "z")`: Recuperando-se a hipótese de indução, esta instanciação deve ser realizada, de modo que obtenhamos o seqüente a seguir.

```
{-1}  EXISTS (gamma: gammaT | gamma_welldef?(suff_ab_prot(prot, 1), gamma, z)):
       normalizeseq(extract_gamma(gamma) o suff_ab_prot(prot, 1)`seq(0)(x, y))
         = empty_seq
[-2]  gamma_welldef?(prot,defgammawithtriplet(tau1,prot`seq(i)(x,z),tau2),z)
[-5]  normalizeseq(tau1 o prot`seq(i)(x, z) o tau2) =
            defalphbetawithop(opDef(decrypt, x), 1)
  |-------
[5]   EXISTS (gamma: gammaT | gamma_welldef?(x!1, gamma, z)):
       normalizeseq(extract_gamma(gamma) o
         (defalphbetawithop(opDef(encrypt, x), 1) o
          prot`seq(0)(x, y) ^ (1, prot`seq(0)(x, y)`length - 1))) = empty_seq
```

  O termo `suff_ab_prot(prot, 1)`, utilizado na instanciação, representa o protocolo `prot` com o primeiro passo de protocolo sem operador da posição 0, ou seja, $P_0(\mathrm{x},\mathrm{y})_{[1,|P_0(\mathrm{x},\mathrm{y})|-1]}$. Em nossa especificação, isto é

```
suff_ab_prot(prot, 1)`seq(0)(x, y) =
     prot`seq(0)(x, y)^(1, prot`seq(0)(x, y)`length - 1)
```

- `(skolem -1 "gammap")`: Skolemizamos `{-1}`, acima, com um `gammaT` que chamaremos de `gammap`, assumido como bem definido.

- `(inst 5 "gammap o defgammawithtriplet(tau1, prot`seq(i)(x, z), tau2)")`: Instanciamos o `gammaT`, em `[5]`, do seqüente anterior. Este `gammaT` é bem definido, pois `gammap` e `defgammawithtriplet(tau1, prot`seq(i)(x, z), tau2)` são bem definidos e, de acordo com o lema `gammas_welldefs_imp_o_is_welldef`, que não descreveremos em detalhes, a composição de dois objetos `gammaT` bem definidos resulta em um objeto `gammaT` bem definido. Assim, o seqüente anterior se torna o seguinte:

```
[-1]  normalizeseq(extract_gamma(gammap) o suff_ab_prot(prot, 1)`seq(0)(x, y))
        = empty_seq
[-2]  normalizeseq(tau1 o prot`seq(i)(x, z) o tau2) =
      defalphbetawithop(opDef(decrypt, x), 1)
  |-------
{3}   normalizeseq(extract_gamma(gammap) o
        extract_gamma(defgammawithtriplet(tau1, prot`seq(i)(x, z), tau2)) o
        defalphbetawithop(opDef(encrypt, x), 1) o
        prot`seq(0)(x, y) ^ (1, prot`seq(0)(x, y)`length - 1)) = empty_seq
```

  Em `{3}`, temos o seguinte: $\overline{\mathtt{gammap}\,\tau_1 P_i(\mathrm{x},\mathrm{z})\tau_2\, E_{\mathrm{x}}\, P_0(\mathrm{x},\mathrm{y})_{[1,|P_0(\mathrm{x},\mathrm{y})|-1]}} = \lambda$. Como $\overline{\tau_1 P_i(\mathrm{x},\mathrm{z})\tau_2} = D_{\mathrm{x}}$, pela fórmula `[-2]`, obtemos em `{3}`: $\overline{\mathtt{gammap}\, P_0(\mathrm{x},\mathrm{y})_{[1,|P_0(\mathrm{x},\mathrm{y})|-1]}} = \lambda$. Pela hipótese de indução em `[-1]`, vemos que `{3}` é verdadeiro e concluímos este ramo da prova.

II. `prot(0)(x,y)(0) = Dx`: Neste caso, temos o seguinte seqüente:

```
{-1}  prot`seq(0)(x, y) = defalphbetawithop(opDef(decrypt, x), 1) o
       prot`seq(0)(x, y)^(1, prot`seq(0)(x, y)`length - 1)
  |-------
[5]   EXISTS (gamma: gammaT | gamma_welldef?(x!1, gamma, z)):
       normalizeseq(extract_gamma(gamma) o prot`seq(0)(x, y)) = empty_seq
```

Em `{-1}`, temos que $P_0(x,y) = D_x P_0(x,y)_{[1,|P_0(x,y)|-1]}$ e, em `[5]`, existe γ bem-definido, tal que $\overline{\gamma P_0(x,y)} = \lambda$.

- `(replace -1 5)`: Substituindo `{-1}` em `[5]`, obtemos o seqüente:

```
  |-------
{5} EXISTS (gamma: gammaT | gamma_welldef?(x!1, gamma, z)):
    normalizeseq(extract_gamma(gamma) o defalphbetawithop(opDef(decrypt,x),1) o
       prot`seq(0)(x, y)^(1, prot`seq(0)(x, y)`length - 1))) = empty_seq
```

- Considerando-se a hipótese de indução, instanciada com `suff_ab_prot(prot,1)`, temos em `{-1}`:

```
{-1}  EXISTS (gamma: gammaT | gamma_welldef?(suff_ab_prot(prot, 1), gamma, z)):
       normalizeseq(extract_gamma(gamma) o suff_ab_prot(prot, 1)`seq(0)(x, y))
        = empty_seq
  |-------
[6]   EXISTS (gamma: gammaT | gamma_welldef?(x!1, gamma, z)):
       normalizeseq(extract_gamma(gamma) o
         (defalphbetawithop(opDef(decrypt, x), 1) o
          prot`seq(0)(x, y)^(1, prot`seq(0)(x, y)`length - 1))) = empty_seq
```

- `(skolem -1 "gammap")`: Com a skolemização de `{-1}`, esta fórmula se torna:

```
{-1} normalizeseq(extract_gamma(gammap) o suff_ab_prot(prot, 1)`seq(0)(x, y))
       = empty_seq
```

- `(inst 6 "gammap o defGammaWithOp(opDef(encrypt, x), 1)")`: A instanciação de `[6]` com `gammap` concatenado a uma seqüência com um único operador E_x, produz o seqüente:

```
[-1]  normalizeseq(extract_gamma(gammap) o suff_ab_prot(prot, 1)`seq(0)(x, y))
        = empty_seq
  |-------
{6} normalizeseq(extract_gamma(gammap) o
     extract_gamma(defGammaWithOp(opDef(encrypt, x), 1)) o
     (defalphbetawithop(opDef(decrypt, x), 1) o
      prot`seq(0)(x, y)^(1, prot`seq(0)(x, y)`length - 1))) = empty_seq
```

Em `{6}`, temos o seguinte: $\overline{\texttt{gammap}\, E_x D_x\, P_0(x,y)_{[1,|P_0(x,y)|-1]}} = \lambda$. Claramente, `{6}` é verdadeiro, considerando-se a hipótese de indução.

III. `prot(0)(x,y)(0) = Ey`: Este caso é análogo ao caso I e não será mostrado.

Lema `secure_impl_balanced`: Com este lema concluiremos esta seção, provando que um protocolo em cascata seguro implica no balanceamento deste protocolo. O lema `secure_impl_balanced` é enunciado abaixo.

```
secure_impl_balanced : LEMMA FORALL (prot : welldefined_protocol,
                        x : U, y : U | x /= y, z : U | z /= x AND z /= y) :
  secure_protocol?(prot, x, y, z) => balanced_cascade_protocol?(prot)
```

Queremos provar o seguinte objetivo:

```
  |-------
{1}  FORALL (prot: welldefined_protocol, x: U, y: U | x /= y,
                                          z: U | z /= x AND z /= y):
       secure_protocol?(prot, x, y, z) => balanced_cascade_protocol?(prot)
```

- `(skeep)`: A skolemização do objetivo acima produz o seqüente:

```
[-2]  secure_protocol?(prot, x, y, z)
  |-------
[1]   balanced_cascade_protocol?(prot)
```

- `(lemma unbalanced_impl_insecure_At_0)`: Invocando este lema e fazendo a instanciação com as *variáveis de skolem* geradas pelo comando anterior, obtemos o seqüente mostrado a seguir.

```
{-1}  EXISTS (gamma: gammaT | gamma_welldef?(prot, gamma, z)):
       normalizeseq(extract_gamma(gamma) o prot`seq(0)(x, y))
         = empty_seq
[-3]  secure_protocol?(prot, x, y, z)
  |-------
```

- `(lemma secure_eq_not_insecure_prot)`: Após a instanciação adequada deste lema, temos, no conseqüente, uma fórmula com o predicado `insecure_protocol?(prot, x, y, z)`. Expandindo a definição de `insecure_protocol?`, obtemos em `{1}`:

```
[-1]  EXISTS (gamma: gammaT | gamma_welldef?(prot, gamma, z)):
       normalizeseq(extract_gamma(gamma) o prot`seq(0)(x, y))
        = empty_seq
  |-------
{1}   EXISTS (gamma: gammaT | gamma_welldef?(prot, gamma, z),
            i: nat | i < prot`length):
       normalizeseq(extract_gamma(gamma) o extract_eN(prot, i, x, y))
          = empty_seq
```

  Claramente, o seqüente acima pode ser provado com a skolemização de `[-1]`, e instanciação de `{1}` com o `gammaT` skolemizado em `[-1]` e `i = 0`. Deste modo, completamos a prova do lema `secure_impl_balanced`.

5.5 *Teoria* SecuritySufficiency: Prova da Suficiência do Teorema 1

A seguir, apresentamos a prova de suficiência do **Teorema 1**. Mostraremos somente a prova para o lema principal `alpha0_and_bal_secure`, e apresentaremos os enunciados de lemas importantes para esta prova.

Lema `gammaAlpha0Complements`: Neste lema, provamos o **Lema 8** mostrado na prova de suficiência do **Teorema 1**.

```
gammaAlpha0Complements : LEMMA FORALL(prot : welldefined_protocol, x : U,
                                      y : U | x /= y, z : U) :
  insecure_protocol?(prot,x,y,z) =>
   EXISTS(gamma : gammaT | gamma_welldef?(prot,gamma,z)) :
     areseqcomplements?(normalizeseq(extract_gamma(gamma)),prot`seq(0)(x, y))
```

Lema `userBalancing`: Correspondente ao **Lema 6** do capítulo **Capítulo** 4, `userBalancing` é formalizado conforme abaixo. Este lema representa uma importante parte da prova de `alpha0_and_bal_secure` e depende dos lemas **Lema 2**, **Lema 3**, **Lema 4** e **Lema 5**, como mostrado na seção **4.2.3**. A teoria `UserBalancingProperty` apresenta as especificações de definições e lemas que permitem a prova de `userBalancing`.

```
userBalancing : LEMMA FORALL (prot : welldefined_protocol, z : U,
            gamma : gammaT | gamma_welldef?(prot,gamma, z), w : U | w /= z) :
  balanced_cascade_protocol?(prot) =>
    balancedseq_wrt?(normalizeseq(extract_gamma(gamma)), w)
```

Lema `alpha0WithoutEyContainsOnlyEx`: Se não existe um E_y em `prot`seq(0)(x,y)`, então todos os operadores desta seqüência são E_x. Esta situação é descrita na prova analítica da suficiência, onde analisamos o caso $(P_0)_i \neq E_y$.

```
alpha0WithoutEyContainsOnlyEx : LEMMA FORALL(prot : welldefined_protocol,
                                             x : U, y : U | x /= y) :
  (NOT memberAlphaBeta(opDef(encrypt,y),x,y,prot(0))
       AND alpha0ContainsE?(prot, x, y)) =>
  FORALL(i : below[prot(0)(x,y)`length]) : prot(0)(x,y)(i) = opDef(encrypt,x)
```

O lema principal da prova de suficiência do **Teorema 1** é mostrado a seguir.

```
alpha0_and_bal_secure : LEMMA FORALL (prot : welldefined_protocol,
                    x : U, y : U | x /= y, z : U | z /= x AND z /= y) :
  ( alpha0ContainsE?(prot, x, y) AND
    balanced_cascade_protocol?(prot) ) => secure_protocol?(prot, x, y, z)
```

E o objetivo a ser provado é o seguinte:

```
  |-------
[1]   FORALL (prot: welldefined_protocol, x: U, y: U | x /= y,
            z: U | z /= x AND z /= y):
       (alpha0ContainsE?(prot, x, y) AND balanced_cascade_protocol?(prot))
        => secure_protocol?(prot, x, y, z)
```

- `(skeep)`: Skolemizando o objetivo acima, temos:

```
{-1}  alpha0ContainsE?(prot, x, y)
{-2}  balanced_cascade_protocol?(prot)
  |-------
{1}   secure_protocol?(prot, x, y, z)
```

- `(case "insecure_protocol?(prot,x,y,z)")`: Conforme mostrado no capítulo **Capítulo 4**, **B. Suficiência**, geraremos uma prova por contradição, supondo que o protocolo em cascata `prot` é inseguro.

```
{-1}  insecure_protocol?(prot, x, y, z)
[-2]  alpha0ContainsE?(prot, x, y)
[-3]  balanced_cascade_protocol?(prot)
  |-------
```

- `(lemma gammaAlpha0Complements)`: Com este lema, e considerando `{-1}` anterior, obtemos uma fórmula onde afirma-se que existe um `gammaT` que é o complemento de `prot`seq(0)(x, y)`.

```
{-1}  EXISTS (gamma: gammaT | gamma_welldef?(prot, gamma, z)):
       areseqcomplements?(normalizeseq(extract_gamma(gamma)),
           prot`seq(0)(x, y))
```

- `(skeep -1)`: Após a skolemização da fórmula acima, obtemos o seqüente:

```
{-1}  areseqcomplements?(normalizeseq(extract_gamma(gamma)),prot`seq(0)(x, y))
[-2]  insecure_protocol?(prot, x, y, z)
[-3]  alpha0ContainsE?(prot, x, y)
[-4]  balanced_cascade_protocol?(prot)
  |-------
```

- `(lemma userBalancing)`: Com este lema adequadamente instanciado, e nomeando `normalizeseq(extract_gamma(gamma))` como `normGamma`, temos uma nova fórmula em `{-1}`.

```
{-1}  FORALL (w: U | w /= z): balancedseq_wrt?(normGamma, w)
{-2}  areseqcomplements?(normGamma, prot`seq(0)(x, y))
[-3]  alpha0ContainsE?(prot, x, y)
  |-------
```

Assim como na prova analítica da suficiência, geramos uma contradição entre `{-1}` e `{-2}`, considerando dois casos relativos aos operadores contidos em `prot(0)(x,y)`.

I. `(case "memberAlphaBeta(opDef(encrypt,y),x,y,prot(0))")`: Neste caso, `prot(0)(x,y)` possui um operador E_y, conforme em `{-1}`, no seqüente a seguir. A fórmula em `[-2]`, abaixo, é a fórmula `{-1}` do seqüente anterior, instanciada com `y`:

```
{-1}  memberAlphaBeta(opDef(encrypt, y), x, y, prot`seq(0))
[-2]  balancedseq_wrt?(normGamma, y)
[-3]  areseqcomplements?(normGamma, prot`seq(0)(x, y))
  |-------
```

- Expandindo as definições de `balancedseq_wrt?` e `areseqcomplements?`, podemos skolemizar e instanciar novas fórmulas, gerando:

```
[-1]  normGamma`length = prot`seq(0)(x, y)`length
{-2}  areopcomplements?(normGamma`seq(i),
                        prot`seq(0)(x, y)`seq(normGamma`length - 1 - i))
[-3]  normGamma`seq(i)`crTyp = encrypt AND normGamma`seq(i)`user = y
  |-------
```

`[-1]` e `{-2}` vêm da definição de `areseqcomplements?`, e `[-3]` resulta de `balancedseq_wrt?`. Considerando-se `{-2}` e `[-3]`, podemos dizer que `prot`seq(0)(x, y)`seq(normGamma`length - 1 - i)` = $D_y$. Mas `prot`seq(0)(x,y)` não contém, por definição, o operador privado de `y`. Logo, chegamos a uma contradição para este caso.

II. Este ramo da prova é conforme a seguir. Em `{1}`, temos a situação de que um operador E_y não existe em `prot`seq(0)(x, y)`.

```
[-1]  FORALL (w: U | w /= z): balancedseq_wrt?(normGamma, w)
[-2]  areseqcomplements?(normGamma, prot`seq(0)(x, y))
[-3]  alpha0ContainsE?(prot, x, y)
  |-------
{1}   memberAlphaBeta(opDef(encrypt, y), x, y, (prot)`seq(0))
```

- `(inst -1 x)`: Instanciando `[-1]`, acima, teremos que:

```
[-1]  balancedseq_wrt?(normGamma, x)
```

- `(lemma alpha0WithoutEyContainsOnlyEx)`: Em `{-1}`, a seguir, utilizamos o lema `alpha0WithoutEyContainsOnlyEx` para afirmar que todos os operadores de `prot`seq(0)(x, y)` são $E_x$. Isso quer dizer que, considerando `[-3]`, todos os operadores de `normGamma` são $D_x$, contradizendo `[-2]`, onde diz-se que `normGamma` tem a PB com respeito a `x`.

```
[-1]  FORALL (i: below[finseq_appl[alphabeta](prot)(0)(x, y)`length]):
       prot`seq(0)(x, y)`seq(i) = opDef(encrypt, x)
[-2]  balancedseq_wrt?(normGamma, x)
[-3]  areseqcomplements?(normGamma, prot`seq(0)(x, y))
  |-------
```

Deste modo, concluímos a prova da suficiência para o **Teorema 1**.

5.6 *Teoria* CascadeProtocolsSecurity: Prova final do Teorema 1

A especificação em PVS do **Teorema 1** é dada a seguir, e é válida para todo protocolo em cascata bem-definido e para todo $x, y, z \in U \mid x \neq y$ e $z \neq x, y$.

```
theorem1 : THEOREM FORALL (prot : welldefined_protocol,
                           x : U, y : U | x /= y, z : U | z /= x AND z /= y) :
    secure_protocol?(prot, x, y, z) IFF
    ( alpha0ContainsE?(prot, x, y) AND balanced_cascade_protocol?(prot) )
```

O `theorem1` pode ser trivialmente provado, considerando-se os lemas anteriores: `secProt_imp_alpha0ContainsE`, `secure_impl_balanced` e `alpha0_and_bal_secure`. E, assim, temos a prova final para o teorema da segurança de protocolos em cascata no modelo Dolev-Yao.

Uma aplicação imediata do **Teorema 1** pode ser feita pela definição de um protocolo em cascata simples, conforme mostrado abaixo:

Para dois participantes quaisquer $x, y \in U \mid x \neq y$, seja o protocolo P a seguir:

$$P_0(x, y) = E_x$$
$$P_1(y, x) = D_y E_x$$
$$P_2(x, y) = E_y$$

Em P, a condição inicial de segurança (**Definição 6**) é satisfeita, mas P_1 não possui a PB com respeito a y. Assim, utilizando `theorem1`, podemos provar que P é inseguro.

Capítulo 6.

Conclusões

Formalizamos, em PVS, a prova de segurança para protocolos em cascata no modelo Dolev-Yao, seguindo uma metodologia algébrica, onde os passos de protocolo são especificados como funções de pares de usuários em seqüências de operadores. Os resultados estatísticos com a quantidade de lemas e TCCs provados, em cada teoria especificada, estão no **Anexo 1**.

Apesar da prova de segurança, caracterizada pelo **Teorema 1**, ser de ordem superior, a maior parte de nossos esforços foram no sentido de verificar propriedades sobre seqüências finitas de operadores; em especial, validar as estruturas de dados e funções necessárias ao processo de normalização. O lema `normalize_general`, descrito no capítulo **Capítulo** 5 e correspondente ao **Lema 1** da prova analítica do capítulo **Capítulo** 4, destaca-se como o principal lema deste processo de validação.

Utilizamos, em nossas provas, diferentes técnicas disponíveis no PVS, como técnicas proposicionais (contradição, contraposição, etc), técnicas lógicas (skolemização, instanciação, etc) e indução sobre estruturas envolvidas na especificação, principalmente o comprimento de seqüências.

A metodologia aplicada na verificação dos protocolos em cascata no modelo *Dolev-Yao* é adequada para o tratamento formal da segurança, em geral. Pela utilização de um sistema para automatização de provas, todas as omissões e *pequenas falhas* de especificação são detectadas e necessitam ser provadas ou enunciadas. Neste trabalho, a verificação formal exibiu erros definicionais e omissões presentes na formalização analítica dos protocolos em [6]. Adicionalmente, o processo de formalização permitiu atingir estruturas de dados precisas e adequadas para implementação do modelo. Modificações simples das imprecisões mencionadas permitiram formalizar provas para os resultados apresentados em [6]. Os problemas mais relevantes, encontrados na formalização analítica realizada por Dolev e Yao em [6], são os seguintes:

- Passos de protocolo devem ser normalizados: Em [6], um passo de protocolo é considerado, **por conveniência**, normalizado. Ao formalizar o modelo em PVS, verificamos ser necessário que, **por definição**, um passo de protocolo seja normal, e definimos esta propriedade na condição 2 da **Definição 1**. Trata-se apenas de um rigor matemático necessário para a validade da prova do **Lema 7**, por exemplo; em outros lemas também se verifica o problema. A condição de que um passo de protocolo **deve** ser normalizado ocorre devido à definição da PB. Consideremos um passo de protocolo $\alpha\beta(u,v)$ $(u,v \in U \mid u \neq v)$, que não tem a PB com respeito a u. O passo de protocolo $\alpha\beta(u,v)_{[0,i]}E_uD_u\alpha\beta(u,v)_{[i+1,|\alpha\beta(u,v)|-1]}$ $(i < |\alpha\beta(u,v)|)$ não é normal e tem a PB com respeito a u, mas ainda assim é possível violar o protocolo encontrando os mesmos τ_1 e τ_2 relativos a $\alpha\beta(u,v)$, conforme o **Lema 7**, tais que $\overline{\tau_1\alpha\beta(u,v)_{[0,i]}E_uD_u\alpha\beta(u,v)_{[i+1,|\alpha\beta(u,v)|-1]}\tau_2} = D_u$. Assim, verificamos, durante a especificação em PVS, a necessidade de que um passo de protocolo seja normal. Para fins de formalização algébrica, esta propriedade precisa ser explícita, mas, na prática, a normalização ocorre de modo natural, já que cada operação de encriptação/decriptação ocorre sequencialmente, e um operador não é aplicado à transformação de uma mensagem sem que antes um operador, imediatamente anterior na seqüência de operadores, seja aplicado. Desta forma, um operador ρ_u^c $(u \in U)$ *desfaz* a transformação da mensagem resultante da aplicação de um operador ρ_u anterior na seqüência.
- Generalização de lemas: Nossa formalização permitiu especificar algumas definições e lemas de modo mais geral que o proposto por Dolev e Yao. Por exemplo, em [6], são apresentados os dois lemas a seguir

  (1) "*Seja P um protocolo em cascata balanceado. E seja μ qualquer seqüência de operadores tendo a* PE *com respeito a $z \in U$. Para toda seqüência de operadores $\eta \in (\Sigma_1^*(z) \cup \Sigma_2)^*$, $\mu\eta$ e $\eta\mu$ têm a* PE *com respeito a z.*"; e

  (2) "*Seja P um protocolo em cascata balanceado. Então toda seqüência de operadores $\eta \in (\Sigma_1^*(z) \cup \Sigma_2)^*$, tem a* PE *com respeito a $z \in U$.*"

  (2), acima, é provado a partir de (1). Há uma notável discrepância entre o maior esforço para se provar (1), em relação ao esforço para provar (2). Além disso,

existe certa redundância na consideração de seqüências $\eta \in (\Sigma_1^*(z) \cup \Sigma_2)^*$, por ambos os lemas, que poderia ser evitada. A partir dessa análise, formalizamos estes dois lemas de uma forma mais geral, conforme pode ser visto no **Lema 3** e no **Lema 4** (reescritos abaixo), e de modo independente, ou seja, para provar qualquer um destes lemas, não é necessário utilizar o outro. Assim, tornamos mais gerais duas propriedades fundamentais presentes na prova original em [6].

"**Lema 3** *Seja um protocolo em cascata balanceado* P, *um usuário* $z \in U$ *e as seqüências de operadores* μ *e* η. *Se* μ *e* η *têm a PE com respeito* a z, *então a seqüência* $\mu\eta$ *tem a PE com respeito a* z."

"**Lema 4** *Seja* P *um protocolo em cascata balanceado*, z *um usuário e* $\forall\gamma \in (\Sigma_1^*(z) \cup \Sigma_2)^*$. *Então*, γ *tem a PE com respeito a* z."

Afirmar que um protocolo executa suas atribuições com segurança, tem sido um desafio há mais de 25 anos e, ainda hoje, diversos paradigmas vêm sendo propostos e novas falhas de protocolos criptográficos detectadas.

Recentemente, trabalhos importantes como alguns descritos na seção 2.5 têm demonstrado a possibilidade de se representar matematicamente algumas características de implementações reais, de modo que se possam analisar adequadamente aspectos de segurança.

Enquanto se desenvolvem os métodos formais para verificação de protocolos criptográficos, estes também são modificados e criados para novas aplicações. E isso representa mais um desafio para a verificação por métodos formais.

6.1 Trabalhos Futuros

A utilização de diversas técnicas de provas em PVS, bem como a construção do conjunto de especificações sobre seqüências de operadores e propriedades da normalização, podem contribuir para a realização de trabalhos futuros, no sentido de se verificar a correção lógica de classes de protocolos mais gerais, ou mesmo de protocolos específicos.

Atualmente, temos um grande interesse na verificação da segurança de protocolos *oblivious transfer*, que representa uma importante primitiva de segurança para dois

participantes, e que implica em qualquer outra funcionalidade de segurança para dois participantes, sem a necessidade de primitivas adicionais [58].

Referências

[1] M. Backes. Cryptographically sound analysis of security protocols. Ph.D thesis, *Computer Science Department, Saarland University*, 2002.

[2] S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert. PVS Language Reference – Version 2.4. Nov 2001.

[3] J. Crow, S. Owre, J. M. Rushby, N. Shankar, M. Srivas. A Tutorial Introduction to PVS. April 1995.

[4] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993-999, 1978.

[5] 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*, volume 1055 of *Lecture Notes in Computer Science*, pages 147-166. Springer-Verlag, 1996.

[6] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198-208, 1983.

[7] M. Backes, B. Pfitzmann, and M. Waidner. A Universally Composable Cryptographic Library. *IACR Cryptology ePrint Archive, Report 2003/015*, January 2003.

[8] S. Owre, N. Shankar, J. M. Rushby. PVS Prover Guide – Version 2.4. Nov 2001.

[9] A. Rubin and P. Honeyman. Formal methods for the analysis of authentication protocols. *Technical Report CITI TR 93-7*, October 1993.

[10] Jonathan Herzog. Computational Soundness for Standard Assumptions of Formal Cryptography. PhD thesis, *Massachusetts Institute of Technology*, May 2004.

[11] C. J. F. Cremers, S. Mauw & E. P. de Vink, Formal Methods for Security Protocols: Three Examples of the Black-Box Approach. 2003.

[12] B. Blanchet. A computationally sound mechanized prover for security protocols. *In Proc. 27th IEEE Symposium on Security & Privacy*, 2006.

[13] C. Meadows, Formal Verification of Cryptographic Protocols: A survey, *Advances in Cryptology – ASIACRYPT'94* 135-150.

[14] Avinanta Tarigan, Survey in Formal Analysis of Security Properties of Cryptographic Protocol. *Universität Bielefeld*, May 2002.

[15] M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. *In Proc. 1st IFIP International Conference on Theoretical Computer Science, pages 3--22. Springer LNCS 1872*, 2000.

[16] Ross J. Anderson, Security Engineering: a guide to building dependable distributed system. John Wiley & Sons Inc., 2001.

[17] C. J. F. Cremers, Verification of Multi-Protocol Attacks. *Eindhoven University of Technology*, 2004.

[18] J. Kelsey, B. Schneier, D. Wagner, Protocol interactions and the chosen protocol attack, in: *Security Protocols Workshop*, 1997, pp. 91-104.

[19] G. Bella, F. Massacci, and L. C. Paulson. Verifying the SET purchase protocols. *Technical report, Cambridge University Computer Laboratory*, 2001.

[20] Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freedman. The interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274-288, Feb 1987.

[21] Levente Buttyan, Formal methods in the design of cryptographic protocols. *Technical Report SSC/1999/038, Swiss Federal Institute of Technology, ICA*, November 1999.

[22] Siraj Shaikh and Vicky Bush, Analysing the Woo-Lam Protocol Using CSP and Rank Functions. 2006.

[23] S. SCHNEIDER, Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering* 24(9):741–758. 1998.

[24] F. Wang, Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE 92(8)*. 2004.

[25] Vijay Varadharajan, Verification of network security protocols. *Computers and Security* 8(8): 693-708, 1989.

[26] Deepinder P. Sidhu, Authentication protocols for computer networks. *I Computer Networks and ISDN Systems*, 11: 297-310. 1986.

[27] T.Y.C. WOO and S.S. LAM, Authentication for distributed systems. *Computer* 25(1):39–52. 1992.

[28] C. H. West, General technique for communications protocol validation. *IBM Journal of Research and Development*, 22:393-404. 1978.

[29] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. *In 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 174- 190, 2001.

[30] J. Herzog, A computational interpretation of Dolev-Yao adversaries. *Theoretical Computer Science*, vol.340, pp. 57-81, June 2005.

[31] Udi Manber, Introduction to Algorithms: A Creative Approach, *Addison-Wesley Longman Publishing Co., Inc., Boston, MA*, 1989.

[32] S. Delaune and F. Jacquemard, Decision Procedures for the Security of Protocols with Probabilistic Encryption agains Offline Dictionary Attacks. *Journal of Automated Reasoning* 36: 85-124. Nov 2006.

[33] V. Cortier, S. Kremer, R. Küsters and B. Warinschi, Computationally Sound Symbolic Secrecy in the Presence of Hash Functions. *FSTTCS*. 2006.

[34] L. Mazaré, An NP Decision Procedure for Generic Dolev-Yao Constraints with Atomic Keys. Nov 2004.

[35] M. Bellare and P. Rogaway, The game-playing technique. *Cryptology ePrint Archive*, Report 2004/331, Dec 2004.

[36] Catherine Meadows, The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113-131, 1996.

[37] S. Rigby D. Longley, An automatic search for security flaws in key management schemes. In *Computers and Security*, volume 11, pages 75-89, 1992.

[38] M. Burrows, M. Abadi, and R. Needham, A logic of authentication. *ACM Transactions on Computer Systems*, Feb 1990.

[39] B. C. Neuman and S. G. Stubblebine, A note on the use of timestamps as nonces. *Operating System Review*, 27(2):10-14, Apr 1993.

[40] A. Aziz and W. Diffie, Privacy and authentication for wireless local area networks. *IEEE Personal Communications*, pages 25-31, 1994.

[41] Li Gong, Roger Needham and R. Yahalom, Reasoning about belief in cryptographic protocols. *Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 234-248, May 1990.

[42] L. E. Moser, A logic of knowledge and belief for reasoning about computer security. *Proceedings of the Computer Security Foundation Workshop II*, pages 57-63, 1989.

[43] D. Accorsi and L. Viganò, Towards an awareness-based semantics for security protocol analysis. *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[44] C. Cremers, Scyther documentation. http://people.inf.ethz.ch/cremersc/scyther/index.html.

[45] L. Paulson, The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1)85-128. 1998;

[46] H. Rueβ and J. Millen, Local Secrecy for State-Based Models. *Formal Methods in Computer Security*, CAV workshop. Jul 2000.

[47] J. Thayer, J. Herzog, J. Guttman, Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, Volume 7, Issue 2-3:191-230, 1999.

[48] Ran Canetti, Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symp. on the Foundations of Computer Science*. 2001.

[49] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai, Universally composable twoparty and multi-party secure computation. In *Proc. ACM Symp. on the Theory of Computing*, pages 494-503, 2002.

[50] M. Backes, B. Pfitzmann and M. Waidner, A General Composition Theorem for Secure Reactive Systems. In *Proceedings of the 1st Theory of Cryptography Conference (TCC 2004)*, volume 2951 of *Lecture Notes in Computer Science*, pages 336-354. Springer, 2004.

[51] B. Pfitzmann and M. Waidner, A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission, In *IEEE Symposium on Security and Privacy*, pages 184-200. *IEEE Computer Society Press*, 2001.

[52] Ran Canetti, Security and composition of cryptographic protocols: a tutorial (part I). *ACM SIGACT News* - Volume 37, Sep 2006.

[53] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). *In Proc. CCS'03, pages 220-230*, 2003.

[54] J. Bryans and S. Schneider. CSP, PVS, and a recursive authentication protocol. *In Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 3-5 1997. Available at http://dimacs.rutgers.edu/Workshops/Security/.

[55] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. *In TPHOLS'97*, 1997.

[56] Mohamed Layouni, Jozef Hooman, and Sofiène Tahar. Modeling and Verication of Leaders Agreement in the Intrusion-Tolerant Enclaves Using PVS. May 2003.

[57] I. Cervesato. The Dolev-Yao intruder is the most powerful attacker. *In Proceedings of the Sixteenth Annual Symposium on Logic in Computer Science*, June 2001.

[58] Joe Kilian. Founding Cryptography on Oblivious Transfer, *Proceedings*, *20th Annual ACM Symposium on the Theory of Computation (STOC)*, 1988.

[59] Bruno Blanchet and Avik Chaudhuri. Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage. *In IEEE Symposium on Security and Privacy*, May 2008. IEEE.

[60] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Report arXiv:0802.3444v1*, February 2008. Available at http://arxiv.org/abs/0802.3444v1.

[61] Steve Kremer. Formal Verification of Cryptographic Protocols. *Steve Kremer. MOdelling and VErifying parallel Processes (MOVEP'06)*, June 2006. Available at http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/kremer-movep06.pdf.

[62] A. S. Troelstra, H. Schwichtenberg, Basic Proof Theory. *Cambridge University Press* 1996.

[63] S. Owre and N. Shankar. The formal semantics of PVS. *Technical report, SRI-CSL-97-2, Computer Science Laboratory, SRI International*, Menlo Park, CA, August 1997. Available at http://pvs.csl.sri.com/.

[64] M. Backes, B. Pfitzmann, and M. Waidner. Limits of the Reactive Simulatability/UC of Dolev-Yao Models with Hashes. *Cryptology ePrint Archive, Report 2006/014* (http://eprint.iacr.org/2006/068), 2006.

[65] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. *In Proc. 20th STACS, vol 2607 of Lecture Notes in Computer Science*. Springer, 2003.

[66] M. Backes, C. Jacobi, and B. Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. *In Proc. 11th FME*, 2002.

Anexo 1 – Estatísticas da Implementação em PVS

A seguir, mostramos o status das teorias provadas. Estas informações são geradas pelo PVS através do comando `prove-importchain`, que executa todas as provas previamente realizadas em uma teoria, e em teorias *importadas*, inclusive provas de TCCs. A maioria dos TCCs é provada automaticamente pelo PVS, por meio de estratégias *built-in*. Entretanto, provamos 13 TCCs que não puderam ser provados pelas estratégias pré-definidas do PVS.

```
Proof summary for theory SecuritySufficiency
   prot_to_gamma_wo_alpha0_TCC1.....proved - complete   [shostak](0.04 s)
   gamma_from_prot_welldef..........proved - complete   [shostak](0.74 s)
   extractN_protToGamma_property_TCC1....proved - complete   [shostak](0.01 s)
   extractN_protToGamma_property.....proved - complete   [shostak](1.12 s)
   rewriteGammaWithProtocolElements......proved - complete   [shostak](0.07 s)
   gammaAlpha0Complements_TCC1.....proved - complete   [shostak](0.48 s)
   gammaAlpha0Complements..........proved - complete   [shostak](0.22 s)
   opInSeqImpExistsMinpos..........proved - complete   [shostak](0.67 s)
   alpha0WithoutEyContainsOnlyEx_TCC1....proved - complete   [shostak](0.47 s)
   alpha0WithoutEyContainsOnlyEx_TCC2....proved - complete   [shostak](0.04 s)
   alpha0WithoutEyContainsOnlyEx.....proved - complete   [shostak](1.21 s)
   alpha0_and_bal_secure...........proved - complete   [shostak](0.48 s)
   Theory totals: 12 formulas, 12 attempted, 12 succeeded (5.55 s)

Proof summary for theory UserBalancingProperty
   user_balanced?_TCC1.....proved - complete   [shostak](0.00 s)
   linkage_property?_TCC1..........proved - complete   [shostak](0.30 s)
   sigma1_sat_link_property........proved - complete   [shostak](2.76 s)
   sigma2_3_sat_link_property.....proved - complete   [shostak](4.50 s)
   admissible_language_sat_link_property...proved - complete   [shostak](0.02 s)
   balanced_prot_imp_linkage_in_sigmas...proved - complete   [shostak](0.90 s)
   userBalancing...................proved - complete   [shostak](1.27 s)
   Theory totals: 7 formulas, 7 attempted, 7 succeeded (9.75 s)

Proof summary for theory UserMonoidCryptOps
   seqDefWithOp_TCC1.....proved - complete   [shostak](0.01 s)
   normalseqZ?_TCC1.....proved - complete   [shostak](0.01 s)
   reduzibleZimpliessize..........proved - complete   [shostak](0.02 s)
   first_cancelableZ_TCC1..........proved - complete   [shostak](0.02 s)
   first_cancelableZ_TCC2..........proved - complete   [shostak](0.03 s)
   first_cancelableZ_TCC3..........proved - complete   [shostak](0.04 s)
   first_cancelableZ_TCC4..........proved - complete   [shostak](0.32 s)
   first_cancelableZ_TCC5..........proved - complete   [shostak](0.31 s)
   fc_boundZ.......................proved - complete   [shostak](0.51 s)
   normalizeseqZ_TCC1.....proved - complete   [shostak](0.01 s)
   normalizeseqZ_TCC2.....proved - complete   [shostak](0.04 s)
   normalizeseqZ_TCC3.....proved - complete   [shostak](0.30 s)
   normalizeseqZ_TCC4.....proved - complete   [shostak](0.03 s)
   normalizeseqZ_TCC5.....proved - complete   [shostak](0.03 s)
   normalizeseqZ_TCC6.....proved - complete   [shostak](0.30 s)
   normalizeseqZ_is_normalized.....proved - complete   [shostak](0.11 s)
   user_normalize_property.....proved - complete   [shostak](0.54 s)
```

```
   crTyp_normalize_property.............proved - complete   [shostak](0.43 s)
   subwords_preserve_operators..........proved - complete   [shostak](0.21 s)
   normalization_preserves_operators.....proved - complete   [shostak](2.08 s)
   fl_ops?_TCC1.........................proved - complete   [shostak](0.01 s)
   fl_ops?_TCC2.........................proved - complete   [shostak](0.03 s)
   Theory totals: 22 formulas, 22 attempted, 22 succeeded (5.39 s)

Proof summary for theory SecurityNecessity
   suff_ab_TCC1.........................proved - complete   [shostak](0.04 s)
   sub_alphabeta_ok.....................proved - complete   [shostak](2.11 s)
   suff_ab_prot_TCC1....................proved - complete   [shostak](0.60 s)
   suff_ab_prot_TCC2....................proved - complete   [shostak](0.46 s)
   suff_ab_prot_TCC3....................proved - complete   [shostak](2.29 s)
   alpha0_suff_eq_TCC1..................proved - complete   [shostak](0.37 s)
   alpha0_suff_eq_TCC2..................proved - complete   [shostak](0.11 s)
   alpha0_suff_eq.......................proved - complete   [shostak](0.31 s)
   extract_seq_from_triplet_TCC1........proved - complete   [shostak](0.01 s)
   extract_seq_from_triplet.............proved - complete   [shostak](0.04 s)
   extract_triplet_as_composition.......proved - complete   [shostak](0.84 s)
   triplet_welldef......................proved - complete   [shostak](0.14 s)
   gamma_in_prot_suff...................proved - complete   [shostak](0.82 s)
   unbalanced_implies_complement_in_sigma1...proved - complete [shostak](0.51 s)
   extractable_op_TCC1..................proved - complete   [shostak](0.07 s)
   extractable_op_TCC2..................proved - complete   [shostak](0.02 s)
   extractable_op.......................proved - complete   [shostak](2.10 s)
   extractable_decUser..................proved - complete   [shostak](0.95 s)
   suff_unbal_prot_is_unbal.............proved - complete   [shostak](1.04 s)
   secProt_imp_alpha0ContainsE..........proved - complete   [shostak](0.28 s)
   unbalanced_impl_insecure_At_0........proved - complete   [shostak](4.94 s)
   secure_impl_balanced.................proved - complete   [shostak](0.05 s)
   Theory totals: 22 formulas, 22 attempted, 22 succeeded (18.10 s)

Proof summary for theory SecurityDefinitions
   wellDefInSigma1_subalphas............proved - complete   [shostak](0.34 s)
   extract_gamma_TCC1...................proved - complete   [shostak](0.01 s)
   extract_gamma_TCC2...................proved - complete   [shostak](0.02 s)
   extract_gamma_TCC3...................proved - complete   [shostak](0.13 s)
   correspondence_unbal_bal_alpbet......proved - complete   [shostak](0.09 s)
   unbalanced_property..................proved - complete   [shostak](0.20 s)
   balanced_cascade_protocol?_TCC1......proved - complete   [shostak](0.01 s)
   correspondence_unbal_bal_prot........proved - complete   [shostak](0.15 s)
   unbalanced_prot_property.............proved - complete   [shostak](0.06 s)
   alpha0ContainsE?_TCC1................proved - complete   [shostak](0.76 s)
   alpha0FunctionsEq....................proved - complete   [shostak](0.16 s)
   defGammaWithOp_TCC1..................proved - complete   [shostak](0.04 s)
   defgammawithseq_TCC1.................proved - complete   [shostak](0.02 s)
   normalize_extract....................proved - complete   [shostak](0.26 s)
   gammas_welldefs_imp_o_is_welldef.....proved - complete   [shostak](0.87 s)
   extractGammaAssoc....................proved - complete   [shostak](0.75 s)
   normalizegammacomplementnormal.......proved - complete   [shostak](0.30 s)
   normalseqgamma.......................proved - complete   [shostak](0.38 s)
   secure_eq_not_insecure_prot..........proved - complete   [shostak](0.05 s)
   Theory totals: 19 formulas, 19 attempted, 19 succeeded (4.60 s)

Proof summary for theory CascadeProtocols
   abUsers?_TCC1........................proved - complete   [shostak](0.05 s)
   length_TCC1..........................proved - complete   [shostak](0.06 s)
   extract_eN_TCC1......................proved - complete   [shostak](0.01 s)
   extract_eN_TCC2......................proved - complete   [shostak](0.03 s)
   extract_eN_TCC3......................proved - complete   [shostak](0.01 s)
   ab_ok_imp_notempty...................proved - complete   [shostak](0.06 s)
   prot_welldef_imp_notempty............proved - complete   [shostak](0.11 s)
   prot_has_length_greater_than_0.......proved - complete   [shostak](0.13 s)
```

```
   welldef_prot_all_users..............proved - complete   [shostak](0.18 s)
   welldef_prot_welldef_alpbet..........proved - complete   [shostak](0.03 s)
   first_crypt_elem_TCC1................proved - complete   [shostak](0.27 s)
   first_crypt_elem.....................proved - complete   [shostak](1.51 s)
   Theory totals: 12 formulas, 12 attempted, 12 succeeded (2.45 s)

Proof summary for theory MonoidCryptOps
   complementseq_TCC1..................proved - complete   [shostak]( 0.01 s)
   areseqcomplements?_TCC1.............proved - complete   [shostak]( 0.07 s)
   normalseq?_TCC1.....................proved - complete   [shostak]( 0.01 s)
   reduzibleimpliessize................proved - complete   [shostak]( 0.04 s)
   isreduziblewhenconcatenated.........proved - complete   [shostak]( 0.25 s)
   opcomplements_eq....................proved - complete   [shostak]( 0.05 s)
   arecomplementswhenconcatenated_TCC1...proved - complete   [shostak]( 0.01 s)
   arecomplementswhenconcatenated_TCC2...proved - complete   [shostak]( 0.13 s)
   arecomplementswhenconcatenated_TCC3...proved - complete   [shostak]( 0.12 s)
   arecomplementswhenconcatenated.......proved - complete   [shostak]( 0.11 s)
   subSeqComplement_TCC1................proved - complete   [shostak]( 0.05 s)
   subSeqComplement_TCC2................proved - complete   [shostak]( 0.06 s)
   subSeqComplement.....................proved - complete   [shostak]( 1.02 s)
   subSeqComplementNormal...............proved - complete   [shostak]( 0.84 s)
   subSeqNormal.........................proved - complete   [shostak]( 0.31 s)
   complements_of_uniform_seqs..........proved - complete   [shostak]( 0.25 s)
   first_cancelable_TCC1................proved - complete   [shostak]( 0.03 s)
   first_cancelable_TCC2................proved - complete   [shostak]( 0.02 s)
   first_cancelable_TCC3................proved - complete   [shostak]( 0.02 s)
   first_cancelable_TCC4................proved - complete   [shostak]( 0.32 s)
   first_cancelable_TCC5................proved - complete   [shostak]( 0.30 s)
   fc_bound.............................proved - complete   [shostak]( 0.36 s)
   normalizeseq_TCC1....................proved - complete   [shostak]( 0.01 s)
   normalizeseq_TCC2....................proved - complete   [shostak]( 0.02 s)
   normalizeseq_TCC3....................proved - complete   [shostak]( 0.23 s)
   normalizeseq_TCC4....................proved - complete   [shostak]( 0.01 s)
   normalizeseq_TCC5....................proved - complete   [shostak]( 0.02 s)
   normalizeseq_TCC6....................proved - complete   [shostak]( 0.41 s)
   normal_normalized....................proved - complete   [shostak]( 0.78 s)
   first_canc_greater_imp_normal........proved - complete   [shostak]( 0.85 s)
   firstcancelablearecomplements_TCC1...proved - complete   [shostak]( 0.01 s)
   firstcancelablearecomplements_TCC2...proved - complete   [shostak]( 0.03 s)
   firstcancelablearecomplements........proved - complete   [shostak]( 0.91 s)
   firstcancelableimpliesreduzible......proved - complete   [shostak]( 1.08 s)
   first_cancelable_preserv2_TCC1.......proved - complete   [shostak]( 0.01 s)
   first_cancelable_preserv2............proved - complete   [shostak]( 0.83 s)
   first_cancelable_preserv3_TCC1.......proved - complete   [shostak]( 0.01 s)
   first_cancelable_preserv3............proved - complete   [shostak]( 0.74 s)
   characterization_f_canc_TCC1.........proved - complete   [shostak]( 0.10 s)
   characterization_f_canc_TCC2.........proved - complete   [shostak]( 0.07 s)
   characterization_f_canc..............proved - complete   [shostak]( 1.30 s)
   first_normalize_TCC1.................proved - complete   [shostak]( 0.06 s)
   first_normalize......................proved - complete   [shostak]( 0.09 s)
   normalize_id.........................proved - complete   [shostak]( 1.05 s)
   normalizedImpComplements.............proved - complete   [shostak]( 4.48 s)
   normalize_general_TCC1...............proved - complete   [shostak]( 0.01 s)
   normalize_general_TCC2...............proved - complete   [shostak]( 0.00 s)
   normalize_general_TCC3...............proved - complete   [shostak]( 0.09 s)
   normalize_general_TCC4...............proved - complete   [shostak]( 0.02 s)
   normalize_general....................proved - complete   [shostak](20.72 s)
   normalize_general_seq................proved - complete   [shostak]( 2.82 s)
   normalization_complements............proved - complete   [shostak]( 2.14 s)
   normalize_left.......................proved - complete   [shostak]( 0.02 s)
   normalize_right......................proved - complete   [shostak]( 0.02 s)
   normalizeRightCompl_TCC1.............proved - complete   [shostak]( 0.02 s)
   normalizeRightCompl_TCC2.............proved - complete   [shostak]( 0.10 s)
```

```
   normalizeRightCompl.................proved - complete   [shostak]( 0.37 s)
   Theory totals: 57 formulas, 57 attempted, 57 succeeded (43.81 s)

Proof summary for theory finite_sequences_extras
   first_TCC1..........................proved - complete   [shostak](0.02 s)
   rest_TCC1...........................proved - complete   [shostak](0.01 s)
   insert_TCC1.........................proved - complete   [shostak](0.09 s)
   insert_TCC2.........................proved - complete   [shostak](0.08 s)
   identseq............................proved - complete   [shostak](0.05 s)
   identseq2_TCC1......................proved - complete   [shostak](0.01 s)
   identseq2...........................proved - complete   [shostak](0.27 s)
   eqseq_comm..........................proved - complete   [shostak](0.02 s)
   seqelem_comm........................proved - complete   [shostak](0.06 s)
   eqseq_tran..........................proved - complete   [shostak](0.01 s)
   empty_0.............................proved - complete   [shostak](0.09 s)
   seqcompositionempty1................proved - complete   [shostak](0.29 s)
   seqcompositionempty2................proved - complete   [shostak](0.11 s)
   zerolengthempty.....................proved - complete   [shostak](0.00 s)
   seq_first_rest_TCC1.................proved - complete   [shostak](0.01 s)
   seq_first_rest......................proved - complete   [shostak](1.11 s)
   inv_first_rest_TCC1.................proved - complete   [shostak](0.08 s)
   inv_first_rest......................proved - complete   [shostak](0.37 s)
   first_rest..........................proved - complete   [shostak](0.53 s)
   eqseq_cons_TCC1.....................proved - complete   [shostak](0.03 s)
   eqseq_cons_TCC2.....................proved - complete   [shostak](0.01 s)
   eqseq_cons..........................proved - complete   [shostak](1.40 s)
   eqseq_split.........................proved - complete   [shostak](0.58 s)
   eqseq_reduce........................proved - complete   [shostak](0.72 s)
   eqseq_reduce2_TCC1..................proved - complete   [shostak](0.11 s)
   eqseq_reduce2_TCC2..................proved - complete   [shostak](0.05 s)
   eqseq_reduce2.......................proved - complete   [shostak](0.59 s)
   eqseq_reduce3_TCC1..................proved - complete   [shostak](0.16 s)
   eqseq_reduce3_TCC2..................proved - complete   [shostak](0.01 s)
   eqseq_reduce3.......................proved - complete   [shostak](0.31 s)
   eqseq_reduce4_TCC1..................proved - complete   [shostak]( n/a s)
   eqseq_reduce4_TCC2..................proved - complete   [shostak]( n/a s)
   eqseq_reduce4.......................proved - complete   [shostak](1.10 s)
   eqseq_equivalence_TCC1..............proved - complete   [shostak](0.15 s)
   eqseq_equivalence...................proved - complete   [shostak](0.80 s)
   eqop_comp_TCC1......................proved - complete   [shostak](0.11 s)
   eqop_comp...........................proved - complete   [shostak](0.09 s)
   eqop_comp2..........................proved - complete   [shostak](0.00 s)
   eqop_comp3_TCC1.....................proved - complete   [shostak](0.13 s)
   eqop_comp3..........................proved - complete   [shostak](0.12 s)
   eqop3_comp_TCC1.....................proved - complete   [shostak](0.12 s)
   eqop3_comp_TCC2.....................proved - complete   [shostak](0.15 s)
   eqop3_comp_TCC3.....................proved - complete   [shostak](0.18 s)
   eqop3_comp..........................proved - complete   [shostak](0.46 s)
   o_assoc2A...........................proved - complete   [shostak](0.02 s)
   o_assoc2B...........................proved - complete   [shostak](0.00 s)
   o_assoc3............................proved - complete   [shostak](0.01 s)
   seq_elem_preserve_TCC1..............proved - complete   [shostak](0.13 s)
   seq_elem_preserve_TCC2..............proved - complete   [shostak](0.01 s)
   seq_elem_preserve...................proved - complete   [shostak](0.14 s)
   seq_preserve_TCC1...................proved - complete   [shostak](0.12 s)
   seq_preserve........................proved - complete   [shostak](0.91 s)
   seq_preserve2.......................proved - complete   [shostak](0.76 s)
   seq_preserve3_TCC1..................proved - complete   [shostak]( n/a s)
   seq_preserve3_TCC2..................proved - complete   [shostak]( n/a s)
   seq_preserve3.......................proved - complete   [shostak](0.50 s)
   seq_commute_TCC1....................proved - complete   [shostak](0.02 s)
   seq_commute.........................proved - complete   [shostak](0.00 s)
   seq_commute2_TCC1...................proved - complete   [shostak](0.14 s)
```

```
   seq_commute2.........................proved - complete   [shostak](1.83 s)
   pos_in_seq_preserve_TCC1.............proved - complete   [shostak](0.22 s)
   pos_in_seq_preserve..................proved - complete   [shostak](0.56 s)
   Theory totals: 62 formulas, 62 attempted, 62 succeeded (15.96 s)

 Proof summary for theory CascadeProtocolsSecurity
   theorem1.............................proved - complete   [shostak](0.03 s)
   Theory totals: 1 formulas, 1 attempted, 1 succeeded (0.03 s)

Grand Totals: 214 proofs, 214 attempted, 214 succeeded (105.64 s)
```

Das 214 provas verificadas acima, temos 106 TCCs e dois axiomas, em cujas provas estamos atualmente trabalhando. Estes axiomas são `linkage_property_composition` e `linkage_normalized`, e não foram computados pelo PVS na estatística de provas mostrada. O desenvolvimento completo roda em PVS 4.1 e a especificação foi realizada em 1211 linhas (55 KB) e 22876 linhas (1.4 MB) de provas.

Anexo 2 – Quebra do Protocolo de Needham-Schroeder

Para descrever o processo no qual o protocolo de Needham-Schroeder é *quebrado*, utilizaremos a seguinte notação:

- A, B e Z denotam os participantes de uma rede de comunicação, sendo Z o adversário, usualmente limitado pelas primitivas criptográficas.
- A → B : Indica que A envia uma mensagem para B.
- Z(A): Significa que Z está personificando ou "*se passando por*" um participante A.
- n_X denota um número aleatório (*nonce*) criado pelo participante X.
- pkX e pkX^{-1} denotam as chaves de um participante X. pkX é chamada chave pública e pkX^{-1} chave privada.
- $\{m_1, m_2\}_{pkX}$: Mensagem encriptada com chave pkX, contendo a informação m_1 concatenada à informação m_2. m_1 e m_2 podem ser um nonce ou a identificação de um participante do protocolo.

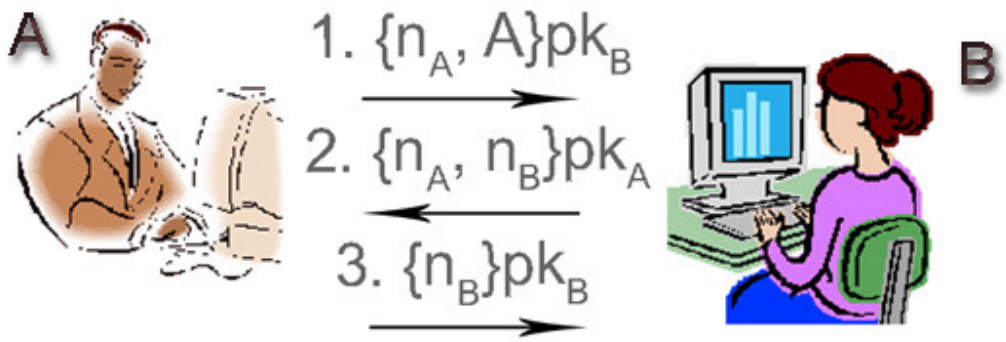

Figura 5: Funcionamento do protocolo de Needham-Schroeder.

Admite-se que as chaves privadas sejam conhecidas somente pelos seus proprietários, ou seja, somente um participante X conhece a chave privada pkX^{-1}. No passo 1 do protocolo, o usuário A envia para B uma mensagem com um *nonce* n_A e seu identificador. Esta mensagem está encriptada com a chave pública de B. No passo 2, B envia para A um *nonce* n_B e também n_A, provando a A que B aprendeu n_A. Neste momento,

B prova a A que sua identidade é a mesma que a esperada por A. No passo 3, A prova a B sua identidade, através do envio de n_B recebido no passo 2.

O ataque *man-in-the-middle* ao protocolo de Needham-Schroeder, descoberto por Lowe, em [5], é mostrado abaixo.

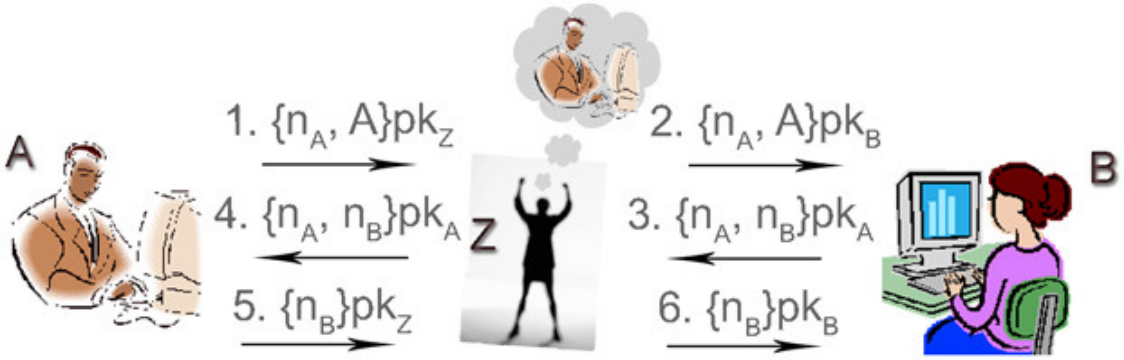

Figura 6: Quebrando o protocolo de Needham-Schroeder.

Z personifica A para se autenticar com B, utilizando informações de sua sessão com A. No ataque, a mensagem do passo 1 do protocolo é enviada de A para Z, e Z envia os mesmos dados da mensagem recebida para B, como se fosse A. B, ao responder para o falso A, espera ter de volta o n_B enviado, para comprovação de que está, efetivamente, se comunicando com A. O problema de Z, personificando A, se autenticar com B sem possuir pk_A^{-1}, está em Z responder corretamente ao desafio proposto por B, que consiste em esperar que A decripte n_B e envie-o de volta a B. E isso pode ser feito, pois Z, durante a autenticação com A, envia a mensagem que contém o desafio de B. Implicitamente no protocolo, o participante A acredita que n_B tenha sido gerado por Z e "revela" a Z este n_B no passo 5. Assim, Z, que se autenticou com A, pode se autenticar também com B. Do ponto de vista de B, a seqüência de mensagens é exatamente a esperada de um participante A que tenha iniciado o protocolo.

A falha deste protocolo poderia resultar na seguinte situação:

> Se A é um indivíduo honesto, Z é um comerciante on-line embusteiro e B é o banco de A, então, quando A executar o protocolo para se autenticar com Z para iniciar uma compra, Z terá se autenticado com B, o banco de A, podendo Z, denotado por Z(A), prosseguir com operações bancárias, *se passando por* A.