Capítulo 1

Introdução à Verificação Automática de Protocolos de Segurança com Scyther

Diego Kreutz (UNIPAMPA), Rodrigo Mansilha (UNIPAMPA), Silvio E. Quincozes (UFF), Tadeu Jenuário (UNIPAMPA), João Otávio Chervinski (Monash University)

Resumo. *Os protocolos de segurança representam o alicerce das comunicações realizadas na Internet. Um dos principais desafios no projeto desses protocolos é garantir a sua própria segurança. Para superar esses desafios, foram desenvolvidas ferramentas de verificação formal e automática de protocolos de segurança, como a Scyther, CryptoVerif, ProVerif, AVISPA e Tamarim Prover. Entretanto, essas ferramentas são ainda pouco conhecidas e utilizadas na prática por projetistas de protocolos e estudantes de computação. Este minicurso visa diminuir essa lacuna através da apresentação da ferramenta Scyther e sua aplicação na verificação de quatro protocolos de segurança.*

1.1. Introdução

Com o surgimento de novos instrumentos legais, como a Lei Geral de Proteção de Dados (LGPD)[1], a segurança na Internet, mais especificamente a proteção dos dados em trânsito, é um dos assuntos que entrou na pauta de diversas nações. As entidades comunicantes e os dados em trânsito são autenticados e protegidos através de protocolos de segurança como o *Needham-Schroeder* (NS) [Needham and Schroeder 1978], *Diffie-Hellman* (DH) [Steiner et al. 1996], *Internet Key Exchange* (IKE) [Carrel and Harkins 1998], *Internet Protocol Security* (IPsec) [Atkinson 1995], *Secure Shell* (SSH) [Lonvick and Ylonen 2006], *Transport Layer Security* (TLS) [Rescorla 2018] e *Signal Protocol*[2].

Um dos principais desafios de projeto e implementação de sistemas e protocolos de segurança da informação é garantir a corretude e a resistência a ataques com o estabelecimento de propriedades como confidencialidade, integridade, disponibilidade, autenticidade e legalidade [Rainer et al. 2020]. A verificação automática e formal de sistemas e protocolos, utilizando métodos formais e matemáticos, tem sido

[1] http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/L13709.htm
[2] https://signal.org/

cada vez mais utilizada para analisar de forma sistemática e automática a robustez de protocolos de segurança [Chen et al. 2016, Baelde et al. 2017, Chudnov et al. 2018, Li et al. 2018, Bai et al. 2018, Liu and Liu 2019, Kreutz et al. 2019, Jenuario et al. 2020, Cohn-Gordon et al. 2020]. Recorrentemente, pesquisas relatam, por exemplo, que o processo de verificação formal já contribuiu de maneira significativa para a correção de protocolos e serviços que, embora vulneráveis, estavam em utilização na Internet [Dalal et al. 2010, Arapinis et al. 2012, Affeldt and Marti 2013, Cremers et al. 2016, Delia Jurcut et al. 2018, Cook 2018].

Existem diversas ferramentas desenvolvidas especificamente para auxiliar o projeto e a verificação automática de protocolos, como Scyther[3] [Cremers 2006, Cremers 2008], CryptoVerif[4] [Blanchet 2006], AVISPA [Armando, A., et al. 2005], ProVerif [Blanchet et al. 2018] e Tamarim Prover[5] [Meier et al. 2013]. No entanto, essas ferramentas são pouco conhecidas e utilizadas pela comunidade. Por exemplo, encontramos pouquíssimos trabalhos (*e.g.*, alguns trabalhos dos autores deste minicurso) sobre verificação formal de protocolos de segurança utilizando essas ferramentas nos principais cursos de computação do Rio Grande do Sul e nas últimas três edições do Workshop Regional de Segurança da Informação e de Sistemas Computacionais (WRSeg) e do Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg), dois dos principais eventos de segurança do Brasil.

Ao final deste minicurso, o estudante (ou profissional) reconhecerá o potencial do emprego de ferramentas de verificação automática e formal de protocolos de segurança. Especificamente, o estudante (ou profissional) será capaz de:

- (o_1) conhecer as semânticas operacionais da ferramenta Scyther (Seção 1.2);
- (o_2) aplicar a Scyther na análise do protocolo Atualização de Chave Simétrica (ACS) (Seção 1.3);
- (o_3) aplicar a Scyther na verificação automática e correção do protocolo Wide Mouth Frog (WMF) [Kelsey et al. 1997, Velibor et al. 2016] (Seção 1.4);
- (o_4) aplicar a Scyther na verificação automática e correção do protocolo Needham-Schroeder (NS) [Needham and Schroeder 1978] (Seção 1.5);
- (o_5) aplicar a Scyther na análise do protocolo Four-party Generalized Needham-Schroeder-Lowe (β) [Cremers and Mauw 2006] (Seção 1.6).

Considerações finais sobre verificação automática utilizando a ferramenta Scyther são apresentadas na Seção 1.7.

1.2. A Ferramenta Scyther

Nesta seção, iremos introduzir as semânticas operacionais e práticas com a ferramenta Scyther [Cremers 2008]. As semânticas operacionais (Seção 1.2.1) são necessárias para traduzir um protocolo de segurança convencional, apresentado numa notação específica, para a semântica da Scyther. Na Seção 1.2.2, apresentamos um primeiro exemplo prático de utilização da ferramenta em ambiente Linux, bem como os pré-requisitos de sistema.

[3] https://people.cispa.io/cas.cremers/scyther/
[4] https://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/
[5] https://tamarin-prover.github.io/

1.2.1. Semânticas Operacionais

Cada ferramenta de verificação automática e formal de protocolos de segurança utiliza linguagens e semânticas próprias. As principais semânticas operacionais da ferramenta Scyther [Cremers 2006, Cremers 2008] são apresentadas a seguir.

Termos atômicos: a ferramenta Scyther manipula termos de modo que o protocolo apresente o comportamento desejado. Os termos atômicos, ou tipos específicos de dados, servem para definir os dados que serão utilizados no protocolo. Entre eles estão: o **var**, que define variáveis para armazenar os dados recebidos de uma comunicação; o **const**, que define constantes que são geradas para cada instanciação de uma função e possuem visibilidade local; e o **fresh**, que define elementos inicializados com valores pseudo-aleatórios.

Chaves simétricas: qualquer termo atômico pode atuar como chave para encriptação simétrica através da notação `{dado}termo`, que corresponde a encriptação de `dado` usando `termo` como chave simétrica (na Seção 1.3 é apresentado um exemplo de protocolo baseado em chaves simétricas). Por padrão, `termo` é interpretado como parte de um infraestrutura de chave simétrica, a menos que seja explicitamente definido como elemento de um par de chaves assimétricas, como segue.

Chaves assimétricas: uma infraestrutura de chave públicas ou assimétricas é predefinida pela ferramenta, sendo `sk(X)` correspondente a uma chave privada de `X` e `pk(X)` a chave pública correspondente. Um dado cifrado utilizando uma chave pública (*e.g.*, `{dado}pk(X)`) só poderá ser decifrado com a chave privada correspondente (`sk(X)`). É importante observar que a Scyther permite modelar infraestruturas de chaves assimétricas adicionais (*e.g.,* usando `termo` como uma das chaves). Para fins didáticos, esse processo é omitido deste minicurso. No entanto, recomendamos que o leitor interessado consulte os tópicos avançados do manual da Scyther[6].

Função Hash: permite embaralhamento através de uma função de resumo criptográfico cujo a função inversa é desconhecida por todos — assim, torando-se impraticável a obtenção dos dados originais a partir e tal resumo. A função Hash pode ser definida através de uma declaração como **hashfunction** `H`, que corresponde a definição de uma função Hash denominada `H`. Essa função pode ser usada da seguinte forma: `H(dado)`, que corresponde a geração de um resumo criptográfico de um `dado` através da função hash `H`. É importante notar que as funções hash criptográficas normalmente são definidas com visibilidade global, pois devem ser conhecidas por todos participantes.

Tipos predefinidos: determinam o comportamento de uma função ou termo. Por exemplo, o tipo **Agent** é utilizado para criar um agente que irá interagir nas comunicações. **Function** é um tipo especial que define um termo como sendo uma função que pode receber uma lista de termos como parâmetro. O tipo **Nonce** é considerado padrão para termos e é destinado ao armazenamento de valores utilizados durante a troca de mensagens.

Tipos básicos de eventos: incluem **send** (enviar) e **recv** (receber). Eles são utilizados para a comunicação entre os agentes de um protocolo (*e.g.*, Alice e Bob). É comum que cada evento de envio (*e.g.*, **send_1**) possua um evento de recebimento correspondente (*e.g.*, **recv_1**). Assim, para representar o envio de uma mensagem de Alice para Bob, contendo o dado cifrado com a chave pública de Bob, pode-se utilizar a sintaxe

[6] https://github.com/cascremers/scyther/blob/master/gui/scyther-manual.pdf

send_1(`Alice`, `Bob`,{`dado`}`pk`(`Bob`)), onde o `dado` é declarado por Alice como, por exemplo, uma variável **fresh** do tipo **Nonce**. Similarmente, para representar o recebimento da mensagem enviada de Alice para Bob, basta utilizar a sintaxe **recv_1**(`Alice`, `Bob`,{`dado`}`pk`(`Bob`)). Com isso, Bob irá receber a mensagem de Alice e utilizar a sua chave privada correspondente para decifrar o dado. O dado será armazenado em uma variável da semântica operacional da Scyther, como uma variável **var** do tipo **Nonce**.

Eventos de afirmação (**claim**): são utilizados em especificações de funções para modelar propriedades de segurança. Na prática, após afirmar que uma variável é secreta, a ferramenta irá verificar se a afirmação procede durante a execução do protocolo (*i.e.,* verificar se somente as partes comunicantes possuem acesso ao dado secreto). Por exemplo, para afirmar que uma variável `nonceB` é secreta, utiliza-se o termo **Secret** dentro de um evento de afirmação (*i.e.,* **claim**(`Bob`,`Secret`,`nonceB`)). Também pode-se utilizar o termo **Nisynch** dentro de um evento de afirmação (*i.e.,* **claim**(`Bob`,`Nisynch`)) para verificar se todas as mensagens recebidas pelo receptor foram de fato enviadas pelo parceiro de comunicação e não por um agente desconhecido.

1.2.2. Práticas com Scyther

Durante o minicurso, nós utilizaremos a versão v1.1.3 (via linha de comando) da ferramenta Scyther, para os sistemas operacionais Linux[7], Windows[8] e Mac OS X[9]. Nesta e nas próximas seções, iremos demonstrar a utilização prática com a versão para Linux, testes realizados na distribuição Debian 10, utilizando o Python na versão 2.7.16.

Ao descompactar o arquivo `scyther-linux-v1.1.3.tgz`, é criado o diretório `scyther-linux-v1.1.3`. Dentro dele há o código Python da Scyther, incluindo o arquivo `scyther.py`, que permite a execução da ferramenta via linha de comando. Ao ser executada com a opção `--help`, a ferramenta exibe suas opções e parâmetros, entre as quais destacamos as seguintes.

(p_1) `--filter=<protocol>[,<label>]` para checar apenas algumas claims específicas do protocolo;
(p_2) `-a, --auto-claims` para ignorar toda e qualquer *claim* existente (definida pelo projetista do protocolo) e gerar automaticamente as *claims* para o protocolo;
(p_3) `-r, --max-runs=<int>` para definir o número máximo `<int>` de execuções (o valor padrão da ferramenta é 5);
(p_4) `-A, --all-attacks` para a ferramenta gerar todos os ataques dentro do espaço de estado com protocolo (sem a definição desta opção, a ferramenta gera apenas 1 ataque).

Vamos utilizar como exemplo o Protocolo 1.1, ilustrado também no diagrama da Figura 1.1, que ilustra uma comunicação entre Alice e Bob utilizando criptografia assimétrica (chaves públicas). No protocolo hipotético, Alice envia (linha 1) para Bob um `nonceA` e Bob responde com um `nonceB` (linha 2). Como pode ser observado, é assumido que tanto Alice quanto Bob possuem um par de chaves pública e privada.

[7] https://people.cispa.io/cas.cremers/downloads/scyther/scyther-linux-v1.1.3.tgz
[8] https://people.cispa.io/cas.cremers/downloads/scyther/scyther-w32-v1.1.3.zip
[9] https://people.cispa.io/cas.cremers/downloads/scyther/scyther-mac-v1.1.3.tgz

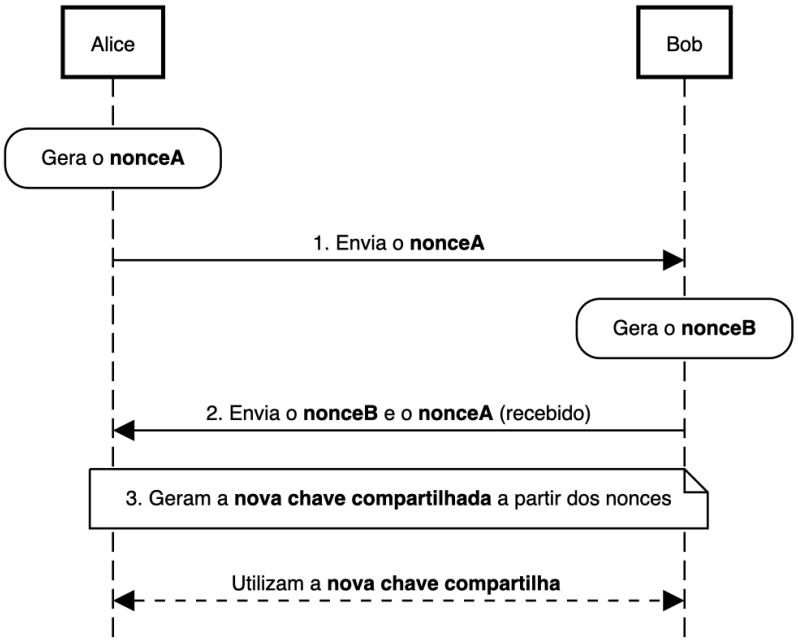

Figura 1.1. Ilustração do exemplo de comunicação com chaves públicas

Protocolo 1.1. Exemplo de comunicação com chaves públicas

1. Alice $\rightarrow$ Bob	[Alice, $E_{pk_{Bob}}$(nonceA)]
2. Bob $\rightarrow$ Alice	[Bob, $E_{pk_{Alice}}$(nonceB,nonceA)]
3. Bob, Alice	K $\leftarrow$ H(nonceA $\|\|$ nonceB)

Vale ressaltar que a semântica de especificação do protocolo é tipicamente diferente da semântica operacional da ferramenta de verificação automática. A especificação dos protocolos segue uma semântica própria, mais geral e em nível mais alto de abstração, como é o caso do Protocolo 1.1. Posteriormente, a semântica do protocolo deve ser traduzida para a semântica específica de cada uma das ferramentas de verificação automática que serão utilizadas para validar o protocolo. No caso deste minicurso, iremos converter a semântica geral para a semântica operacional da ferramenta Scyther (vide Protocolo 1.1 traduzido para a semântica da Scyther no Algoritmo 1.1).

Protocolo 1.1, Alice gera e envia para Bob o `nonceA` (linha 1). A mensagem é cifrada (função E, do inglês E*ncrypt*) com a chave pública do Bob ($E_{pk_{Bob}}$). Ao receber a mensagem, Bob decodifica-a e recupera o `nonceA`, gera e envia um novo `nonceB`, junto ao `nonceA`, para Alice (linha 2) – ambos cifrados com a chave pública da receptora ($E_{pk_{Alice}}$). Alice verifica que o *nonce* `nonceA` recebido é o mesmo que foi enviado para Bob anteriormente. Finalmente, Alice e Bob geram a chave simétrica compartilhada `K` (linha 3) a partir dos *nonces* concatenados ($||$), utilizados como entrada para uma função hash criptográfica `H` (*e.g.*, SHA256). Com isso, nas comunicações subsequentes, Alice e Bob passam a utilizar criptografia simétrica ao invés de assimétrica.

O Algoritmo 1.1 corresponde a uma representação do Protocolo 1.1 na semântica

operacional da Scyther. Vale ressaltar que a linha 3 do protocolo não é representada no algoritmo da Scyther por questão de simplicidade de ilustração. Além disso, se não ocorrer nenhum ataque e comprometimento às linhas 1 e 2 do protocolo, a geração e a utilização da chave secreta compartilhada (linha 3) pode ser considerada segura.

Algoritmo 1.1: Protocolo 1.1 na semântica da Scyther

```
1  const pk: Function;
2  secret sk: Function;
3  inversekeys (pk,sk);
4  const Eve: Agent;
5  untrusted Eve;
6  protocol exemplo(Alice,Bob,Eve){
7     role Alice{
8        fresh nonceA: Nonce;
9        var nonceB: Nonce;
10       send_1(Alice,Bob,{nonceA}pk(Bob));
11       recv_2(Bob,Alice,{nonceA,nonceB}pk(Alice));
12       claim(Alice,Secret,nonceA);
13       claim(Alice,Secret,nonceB);
14       claim(Alice,Nisynch);
15    }
16    role Bob{
17       var nonceA: Nonce;
18       fresh nonceB: Nonce;
19       recv_1(Alice,Bob,{nonceA}pk(Bob));
20       send_2(Bob,Alice,{nonceA,nonceB}pk(Alice));
21       claim(Bob,Secret,nonceA);
22       claim(Bob,Secret,nonceB);
23       claim(Bob,Nisynch);
24    }
25 }
```

Nas linhas 1, 2 e 3, são definidas e declaradas como inversas (uma cifra utilizando a chave pública pode ser decifrada com a respectiva chave privada) as chaves pública `pk` e privada `sk`. Na linha 4, é declarado um agente malicioso Eve, que não é confiável (**untrusted**, na linha 5).

A declaração do protocolo é realizada na linha 6, definindo os três agentes, Alice, Bob e Eve. Para Alice e Bob há um papel (**role**) explicito no protocolo, nas linhas 7 e 16, respectivamente. Os `nonceA` e `nonceB` são declarados como **fresh** (novo valor gerado) e **var** (valor recebido e armazenado) para a Alice e o inverso para Bob. Há dois pares de **send** e **recv** entre Alice e Bob. No **send_1** (linha 10) e **recv_1** (linha 19) é enviado o *nonce* `nonceA` da Alice para o Bob. Já no **send_2** (linha 20) e **recv_2** (linha 11) são enviados os *nonces* `nonceB` e `nonceA` do Bob para a Alice. Finalmente, os papéis são encerrados com três *claims* (linhas 12 a 14 e 21 a 23). Duas *claims* são utilizadas para afirmar que os *nonces* `nonceA` e `nonceB` são secretos. A última é utilizada para verificar se as mensagens recebidas foram, de fato, enviadas pelo par legítimo e não por um impostor intermediário, como o agente malicioso Eve.

Para realizar a verificação automática do Algoritmo 1.1, basta copiar o código para um arquivo de texto (*e.g.*, `protocolo_exemplo.spdl`) e passá-lo como parâmetro para a ferramenta Scyther, como ilustrado na Verificação 1.1. Como pode ser observado, há pelo menos um ataque que pode comprometer o protocolo. Bob (linha 19 do Algo-

ritmo 1.1 e linha 1 do Protocolo 1.1) não dispõem de nenhuma forma de confirmação acerca da procedência de `nonceA` (*i.e.,* ele não é capaz de verificar se veio da Alice ou não).

```
./scyther.py --all-attacks --max-runs=5 protocolo_exemplo.spdl

Verification results:
claim id [exemplo,Alice1], Secret(nA) : No attacks.
claim id [exemplo,Alice2], Secret(nB) : No attacks.
claim id [exemplo,Alice3], Nisynch    : No attacks.
claim id [exemplo,Bob1], Secret(nA)   : Exactly 1 attack.
claim id [exemplo,Bob2], Secret(nB)   : No attacks.
claim id [exemplo,Bob3], Nisynch      : Exactly 1 attack.
```

Verificação 1.1. Execução e saída da Scyther para o Algoritmo 1.1

Eve, o agente malicioso, pode interceptar o envio de Alice para Bob, mudar o `nonceA` e usar a chave pública do Bob para entregar a mensagem a ele. Bob irá reponder para Eve, que irá responder para Alice, que também não consegue verificar se a mensagem veio do Bob ou não. É importante observar que, apesar de ser um protocolo muito simples, há uma falha grave (um ataque), o que reforça a importância de utilizarmos ferramentas de verificação automática de protocolos. Para resolver o problema, é necessário incluir a identificação do remetente e do destinatário nas mensagens cifradas. Na Seção 1.5 discutimos um ataque similar, incluindo o fluxo do ataque e os detalhes da solução do problema, para o protocolo Needham-Schroeder.

1.3. O protocolo de Atualização de Chave Simétrica (ACS)

1.3.1. Protocolo didático de utilização de chaves simétricas

O protocolo didático de Atualização de Chave Simétrica (ACS), criado pelos autores, realiza a atualização da chave simétrica, compartilhada entre Alice e Bob, através de uma chave privada previamente compartilhada entre tais entidades, como ilustrado no diagrama da Figura 1.2. A atualização da chave de sessão, utilizada por ambas as partes, é realizada pelo agente que inicia a comunicação (*e.g.*, Alice, no cenário ilustrado a seguir).

O Protocolo 1.2 detalha o funcionamento do ACS. Para atualizar a chave de sessão, Alice envia uma mensagem ao Bob contendo um `nonce` (linha 1). A chave `K` e o `nonce` são concatenados como parâmetro de uma função hash criptográfica `H` e o resumo criptográfico de saída é atribuído para a chave `K`, atualizando a chave secreta de sessão. Tal prática também é comumente utilizada na geração de códigos de autenticação de mensagem HMACs (*Hash Message Authentication Codes*) [Krawczyk et al. 1997].

Protocolo 1.2. Especificação do ACS entre Alice e Bob

1. Alice $\rightarrow$ Bob	$[E_K(\text{nonce})]$
3. Bob, Alice	$K \leftarrow H(K \,\|\| \text{nonce})$

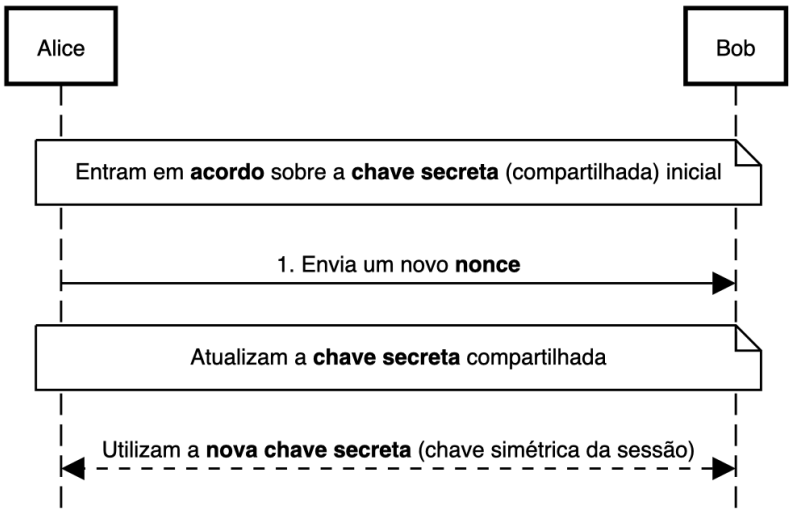

Figura 1.2. Ilustração de Alice e Bob utilizando o ACS

1.3.2. Verificação automática do protocolo

O Algoritmo 1.2 descreve o Protocolo 1.2 (ACS) na semântica da Scyther. Na linha 1, é criada a chave de sessão (secreta) `K`, compartilhada entre Alice e Bob. É importante observar que a própria ferramenta Scyther oferece nativamente a definição de chave compartilhada `k` (minúsculo), que possui a mesma finalidade. Entretanto, neste exemplo de verificação automática, utilizaremos a nossa declaração `K` para fins didáticos (*e.g.*, ficar mais próximo da sintaxe do Protocolo 1.2). Alice envia um novo `nonce`, cifrado utilizando a chave secreta `K`, para Bob (linha 7). Bob recebe (linha 13) e decifra o `nonce`, que será utilizado para atualizar a chave `K`, conforme descrito no Protocolo 1.2. Tanto Alice quanto Bob fazem dois *claims* idênticos – ambos afirmam que o `nonce` e a chave de sessão `K` são secretos.

Algoritmo 1.2: ACS na semântica da Scyther

```
1  secret K: SessionKey;
2  const Eve: Agent;
3  untrusted Eve;
4  protocol ACS(Alice,Bob,Eve){
5     role Alice{
6        fresh nonce: Nonce;
7        send_1(Alice,Bob,{nonce}K(Alice,Bob));
8        claim_ Alice1(Alice,Secret,nonce);
9        claim_ Alice2(Alice,Secret,K);
10    }
11    role Bob{
12       var nonce: Nonce;
13       recv_1(Alice,Bob,{nonce}K(Alice,Bob));
14       claim_Bob1(Bob,Secret,nonce);
15       claim_Bob2(Bob,Secret,K);
16    }
17 }
```

Como pode ser visto na saída da Verificação 1.2, o ACS não possui nenhuma falha. O ACS é apenas um protocolo didático, muito simples, para ilustrar a utilização de chaves

simétricas para realizar uma operação (*i.e.,* atualização da própria chave de sessão, nesta demonstração) entre Alice e Bob.

```
./scyther.py  --all-attacks  --max-runs=5 protocolo_acs.spdl

Verification results:
claim id [ACS,Alice1], Secret(nonce) : No attacks.
claim id [ACS,Alice2], Secret(K)     : No attacks.
claim id [ACS,Bob1], Secret(nonce)   : No attacks.
claim id [ACS,Bob2], Secret(K)       : No attacks.
```

Verificação 1.2. Execução e saída da Scyther para o Algoritmo 1.2

1.4. O protocolo Wide Mouth Frog (WMF)

1.4.1. Protocolo para Troca de Chave Simétrica

O protocolo WMF realiza a troca da chave simétrica, compartilhada entre Alice e Bob, através de um agente confiável (Charles) utilizando uma chave privada (*private key*) compartilhada [Burrows 1989, Burrows et al. 1990, Kelsey et al. 1997, Velibor et al. 2016]. Essa chave é utilizada exclusivamente para distribuição das chaves. Resumidamente, no WMF, quem gera a chave de sessão, utilizada por Alice e Bob, é o agente que inicia a comunicação (*e.g.*, Alice).

O diagrama da Figura 1.3 e o Protocolo 1.3 ilustram e detalham o funcionamento do WMF, respectivamente. Para trocar a chave de sessão, Alice envia uma mensagem ao agente confiável Charles contendo sua identificação, um *timestamp* $\mathtt{T}_a$, a identificação do destinatário (Bob) e a nova chave de sessão $\mathtt{K}_{ab}$, que será utilizada por ambos para assegurar propriedades de segurança (*e.g.*, confidencialidade, integridade e autenticidade) das mensagens trocadas. A mensagem de Alice (linha 1 do Protocolo 1.3) é cifrada (E) utilizando a chave $\mathtt{K}_{ac}$, compartilhada entre ela e Charles.

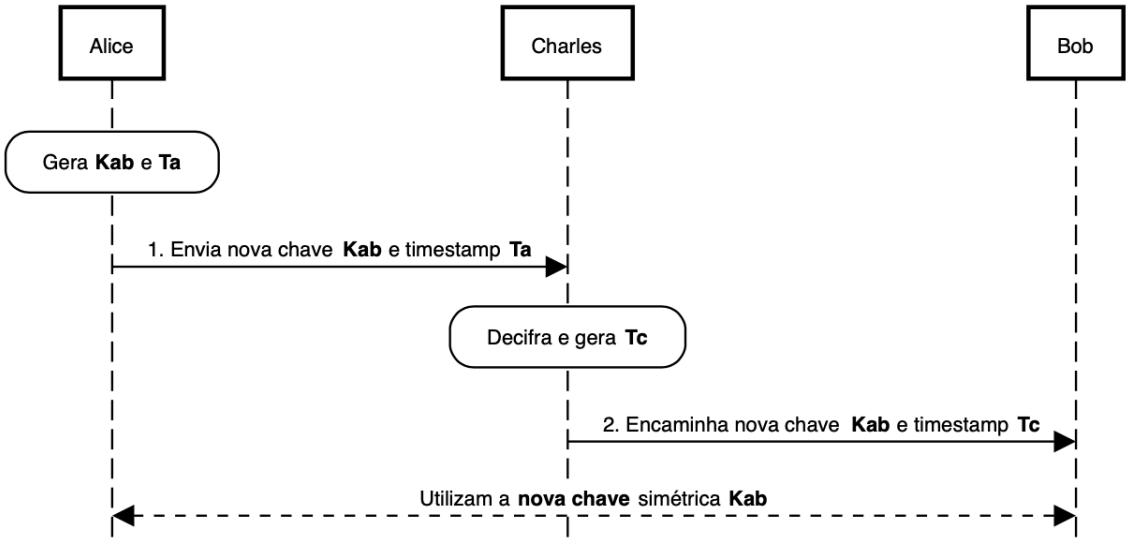

Figura 1.3. Ilustração de Alice e Bob utilizando o WMF

Charles, ao receber a mensagem de Alice, decifra ela utilizando a chave K_{ac} e cria uma nova mensagem contendo sua identificação (`Charles`), um *timestamp* T_c, o identificador do remetente (`Alice`) e a chave de sessão compartilhada K_{ab}, criada e enviada por Alice (linha 2). A mensagem é cifrada (E) utilizando a chave K_{bc}, compartilhada entre Charles e Bob.

Protocolo 1.3. Especificação do WMF entre Alice e Bob

1. Alice → Charles	[Alice, $E_{K_{ac}}(T_a,\text{Bob},K_{ab})$]
2. Charles → Bob	[$E_{K_{bc}}(T_c,\text{Alice},K_{ab})$]

Vale ressaltar que o protocolo original possui diversos problemas, como:

(p_1) requer um relógio global, pois utiliza *timestamps* (carimbos temporais);
(p_2) Charles (ou servidor intermediário) possui acesso a todas as chaves compartilhadas entre Alice e Bob;
(p_3) o valor da chave de sessão K_{ab} é inteiramente definido pela Alice, que precisa ser competente o suficiente para gerar chaves de alta qualidade (*e.g.*, evitar problemas de entropia ou baixa qualidade na geração das chaves através de construções e implementações robustas como a solução proposta em [Kreutz et al. 2019]);
(p_4) um agente malicioso pode re-encaminhar as mensagens que estiverem dentro do período de tempo de validade do *timestamp*, resultando em ataques de *replay*;
(p_5) Alice não assume que Bob existe (e não há mensagens de verificação entre ambos);
(p_6) o protocolo é com estado (*stateful*), o que é tipicamente indesejado por que requer mais funcionalidades e capacidades do servidor (ou Charles, no caso). Por exemplo, análogo a um servidor de e-mails, Charles precisa lidar com situações de indisponibilidade de Bob.

1.4.2. Verificação automática do protocolo

O Algoritmo 1.3 descreve o Protocolo 1.3 (WMF) na semântica da Scyther. Na linha 1 é declarado um novo tipo de termo, denominado `TimeStamp`. O protocolo WMF é iniciado na linha 4, tendo como agentes Alice, Bob e Charles.

No papel (**role**) Alice (linha 5) são criadas e inicializadas com valores pseudo-aleatórios as variáveis `Kab`, que representa a nova chave de sessão que será compartilhada entre Alice e Bob (linha 6), e `Ta`, que representa o *timestamp* (do inglês, carimbo de tempo) (linha 7). Então, Alice envia para Charles (linha 8) sua identificação (`Alice`), o *timestamp* `Ta`, a identificação do recebedor (`Bob`) e `Kab`. É importante observar que a identificação da Alice é enviada fora e dentro (*i.e.*, ...,`Alice`,{`Alice`,...) do bloco cifrado ({...}`k(Alice,Bob)`) da mensagem. A verificação de `Kab` é realizada nas afirmações **claim** das linhas 9 e 10. Nas linhas 10 e 18, é verificado se a nova chave de sessão é "fresca" (`Fresh`). O termo `Empty` indica vazio, uma vez que é a entidade (*i.e.*, Alice ou Bob) verificando a cópia local da chave K_{ab}.

No papel (**role**) Bob (linha 12) são criadas as variáveis `Tc` (linha 13), que armazenará o *timetamp* de Charles, e `Kab` (linha 14), que armazenará a chave de sessão recebida de Charles. Bob recebe a mensagem de Charles (linha 15) contendo a sua

Algoritmo 1.3: WMF na semântica da Scyther

```
1  usertype SessionKey;
2  usertype TimeStamp;
3  const Fresh: Function;
4  protocol WMF(Alice,Bob,Charles){
5     role Alice{
6        fresh Kab: SessionKey;
7        fresh Ta: TimeStamp;
8        send_1(Alice,Charles,Alice,{Alice,Ta,Bob,Kab}k(Alice,Charles));
9        claim_ Alice1(Alice,Secret,Kab);
10       claim_ Alice2(Alice,Empty,(Fresh,Kab));
11    }
12    role Bob{
13       var Tc: TimeStamp;
14       var Kab: SessionKey;
15       recv_2(Charles,Bob,{Charles,Tc,Alice,Kab}k(Bob,Charles));
16       claim_Bob1(Bob,Secret,Kab);
17       claim_Bob2(Bob,Nisynch);
18       claim_Bob3(Bob,Empty,(Fresh,Kab));
19    }
20    role Charles{
21       var Kab: SessionKey;
22       var Ta: TimeStamp;
23       fresh Tc:TimeStamp;
24       recv_1( Alice,Charles,Alice,{Alice,Ta,Bob,Kab}k(Alice,Charles));
25       send_2(Charles,Bob,{Charles,Tc,Alice,Kab}k(Bob,Charles));
26    }
27 }
```

identificação (`Charles`), o *timestamp* `Tc`, a identificação `Alice` e a chave de sessão `Kab`. A verificação da chave de sessão `Kab` e da segurança da comunicação são realizadas nas afirmações **claim** das linhas 16, 17 e 18.

No papel do agente confiável Charles (**role** definido na linha 20) são criadas as variáveis `Kab` (linha 21), que irá receber a chave de sessão de Alice, o *timestamp* `Ta` (linha 22), que irá armazenar o *timestamp* recebido de Alice, e um *timestamp* `Tc` (linha 23), que irá armazenar o *timestamp* gerado por Charles. Charles recebe a mensagem de Alice (linha 24) contendo o identificador dela (`Alice`), o identificador do destinatário (`Bob`) e a chave de sessão `Kab`. A chave de sessão recebida é encaminhada por Charles para Bob, incluindo um novo *timestamp* `Tc` (linha 25).

Ao verificar automaticamente o WMF com a ferramenta Scyther, podemos visualizar algumas falhas na protocolo, como pode ser observado na Verificação 1.3. Um dos principais problema do protocolo é o fato de Alice não saber se Bob recebeu a nova chave secreta K_{ab}. Uma solução para o problema é apresentada no Protocolo 1.4.

A principal correção do Protocolo 1.3, denominada WMF-Lowe [Lowe 1997a, Lowe 1997b], pode ser vista no Protocolo 1.4. Resumidamente, são acrescentadas duas novas mensagens entre Alice e Bob. Bob envia um *nonce* `nonceB` para Alice (linha 3) e Alice responde (linha 4) para Bob com o sucessor do *nonce* (*e.g.*, `nonceB + 1`). Esse mecanismo evita que Bob pense que Alice tenha estabelecido duas sessões de comunicação – um ataque possível na versão original do WMF. Esta solução evita ataques como os registrados pelos eventos `[WMF,Alice5]`, `[WMF,Alice6]`, `[WMF,Bob6]`,

[WMF,Bob7], [WMF,Bob8], [WMF,Bob9], entre Alice e Bob, na Verificação 1.3.

```
./scyther.py --auto-claims --all-attacks protocolo_wmf.spdl

Verification results:
claim id [WMF,Alice3], Secret(Ta)     : No attacks.
claim id [WMF,Alice4], Secret(Kab)    : No attacks within bounds.
claim id [WMF,Alice5], Alive          : Exactly 1 attack.
claim id [WMF,Alice6], Weakagree      : Exactly 1 attack.
claim id [WMF,Alice7], Niagree        : No attacks.
claim id [WMF,Alice8], Nisynch        : No attacks.
claim id [WMF,Bob4], Secret(Kab)      : No attacks within bounds.
claim id [WMF,Bob5], Secret(Tc)       : No attacks within bounds.
claim id [WMF,Bob6], Alive            : Exactly 1 attack.
claim id [WMF,Bob7], Weakagree        : Exactly 1 attack.
claim id [WMF,Bob8], Niagree          : At least 3 attacks.
claim id [WMF,Bob9], Nisynch          : At least 3 attacks.
claim id [WMF,Charles1], Secret(Tc)   : No attacks within bounds.
claim id [WMF,Charles2], Secret(Ta)   : No attacks within bounds.
claim id [WMF,Charles3], Secret(Kab)  : No attacks within bounds.
claim id [WMF,Charles4], Alive        : Exactly 1 attack.
claim id [WMF,Charles5], Weakagree    : Exactly 1 attack.
claim id [WMF,Charles6], Niagree      : At least 3 attacks.
claim id [WMF,Charles7], Nisynch      : At least 3 attacks.
```

Verificação 1.3. Execução e saída da Scyther para o Algoritmo 1.3

Protocolo 1.4. Especificação do WMF-Lowe entre Alice e Bob

1. Alice $\rightarrow$ Charles	[Alice, $E_{K_{ac}}$(T$_a$,Bob,K$_{ab}$)]
2. Charles $\rightarrow$ Bob	[$E_{K_{bc}}$(T$_c$,Alice,K$_{ab}$)]
3. Bob $\rightarrow$ Alice	[$E_{K_{ab}}$(nonceB)]
4. Alice $\rightarrow$ Bob	[$E_{K_{ab}}$(nonceB+1)]

1.5. O protocolo Needham-Schroeder (NS)

1.5.1. Método de Autenticação para dois Participantes

O protocolo NS fornece um método de autenticação para dois participantes em uma rede insegura utilizando criptografia assimétrica [Needham and Schroeder 1978]. Entretanto, o protocolo original apresenta falhas de segurança, as quais são exploradas e corrigidas em [Lowe 1995a], culminando no protocolo *Needham-Schroeder-Lowe* (NSL). Nesta seção, demonstraremos o processo de detecção e avaliação da correção dessas falhas por meio da ferramenta Scyther.

A Figura 1.4 e o Protocolo 1.5 apresentam, respectivamente, o diagrama e a especificação da comunicação entre Alice e Bob utilizando o NS. É assumido que cada usuário possui um par de chaves privada e pública. Alice deseja comunicar-se com Bob e, para tanto, requisita ao servidor de chaves a chave pública do Bob. Na sequência, Alice gera e envia para Bob um `nonceA` juntamente com sua identificação. A mensagem é cifrada com a chave pública do Bob. Ao receber a mensagem, Bob decodifica-a e recupera

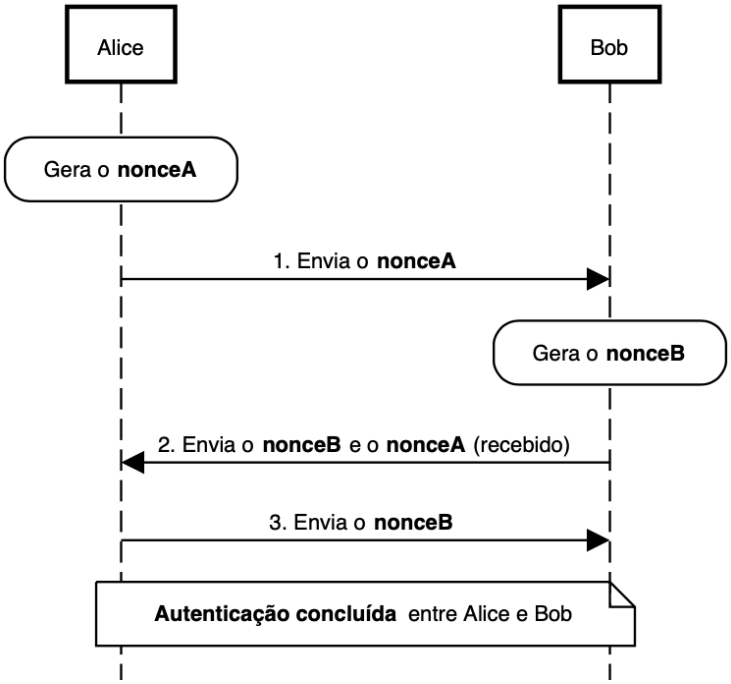

Figura 1.4. Alice e Bob utilizando o protocolo NS

o `nonceA` e a identificação da remetente. Bob lê a identificação de Alice e requisita a respectiva chave pública para o servidor de chaves. No próximo passo, Bob gera e envia para Alice um novo `nonceB`, cifrado com a chave pública da receptora. Ao receber a mensagem, Alice verifica que o `nonceA` recebido é o mesmo que foi enviado para Bob anteriormente. Finalmente, Alice envia para Bob o `nonceB`. Ao receber e verificar os respectivos *nonces*, utilizando as respectivas chaves privadas, Alice e Bob são capazes de confirmar a autenticidade das mensagens.

Protocolo 1.5. Especificação do NS entre Alice e Bob

1. Alice $\rightarrow$ Bob	$[E_{pk_{Bob}}(\text{Alice, nonceA})]$
2. Bob $\rightarrow$ Alice	$[E_{pk_{Alice}}(\text{nonceA, nonceB})]$
3. Alice $\rightarrow$ Bob	$[E_{pk_{Bob}}(\text{nonceB})]$

1.5.2. Verificação automática do protocolo

A Figura 1.5 e o Algoritmo 1.4 apresentam o protocolo NS na semântica da Scyther. As chaves pública `pk` e privada `sk` são declaradas globalmente (linhas 1 e 2), embora sejam atribuídas autonomamente a Alice e Bob pela ferramenta. O comando `inversekeys` (linha 3) determina que as chaves `pk` e `sk` são inversas, ou seja, ao cifrar com a chave pública só é possível decifrar com a chave secreta e vice-versa. Para a execução de ataques ao protocolo, é definido um agente não-confiável Eve (linhas 4 e 5).

A chamada à função **protocol** (linha 6) marca o início da especificação do protocolo NS, com três agentes (Alice, Bob e Eve), sendo que Alice e Bob possuem papéis

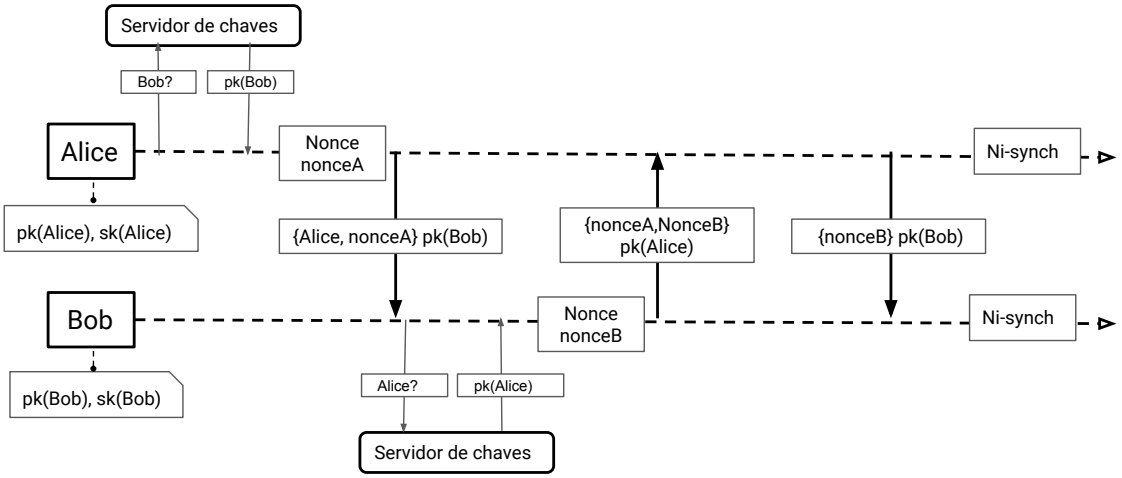

Figura 1.5. Alice e Bob utilizando o protocolo Needham-Schroeder na semântica Scyther

(**role**) explícitos. A variável `nonceA` (linha 8), do tipo **Nonce** e antecedida por **fresh**, irá conter um valor pseudo-aleatório. Já a variável `nonceB` (linha 9), do mesmo tipo, mas **var** ao invés de **fresh**, é utilizada para armazenar valores recebidos (*e.g.*, linha 11).

Cada evento de envio (*e.g.*, **send_1**, linha 10) possui um evento de recebimento (*e.g.*, **recv_1**, linha 20) correspondente. A sintaxe do evento **send_1** indica que a transmissão é de Alice para Bob, e que os dados enviados, cifrados com a chave pública de Bob `pk(Bob)`, são `Alice` e `nonceA`. No agente Bob (linha 20), há um evento com sintaxe quase idêntica, cuja única diferença é o tipo (*i.e.*, **recv** ao invés de **send**). Na sequência, Bob envia os valores `nonceA` e `nonceB` para Alice (linha 21). Alice recebe a resposta (linha 11), verifica o valor do `nonceA` e envia `nonceB` para Bob (linha 12), que recebe e verifica o valor (linha 22).

Finalmente, as afirmações **claim** (linhas 13, 14, 15 23, 24, 25) definem os requisitos de segurança do protocolo. No caso, as afirmações criadas verificam se os *nonces* gerados por Alice (`nonceA`) e Bob (`nonceB`) permanecem secretos durante as comunicações (**claim Secret**) e se as mensagens são de fato trocadas entre eles apenas (**claim Nisynch**).

A ferramenta Scyther aponta a existência de falhas no Algoritmo 1.4, como pode ser visto na Verificação 1.4. O relatório da ferramenta indica pelo menos um ataque que afeta o protocolo NS, relacionado aos eventos `[NS,Bob1]`, `[NS,Bob2]` e `[NS,Bob3]`. O ataque está relacionado aos *nonces* `nonceA` e `nonceB` e ao **Nisynch**. Em casos como esses, a ferramenta gera também um grafo com os passos de execução do(s) ataque(s), como ilustrado na Figura 1.5.

Para comprometer a comunicação entre Alice e Bob, Eve (agente malicioso) precisa apenas convencer Alice de que é Bob. Alice envia a Eve uma mensagem `{Alice,nonceA}` cifrada com a chave pública `pk{Eve}`. Como Eve possui a chave privada correspondente, ele consegue ler o conteúdo da mensagem. Eve então cifra e envia a mensagem para Bob. Bob, sem desconfiar, pois recebeu uma mensagem cifrada com sua chave pública, responde ao atacante com a mensagem `{nonceA,nonceB}pk{Alice}`. O agente malicioso simplesmente encaminha a mensagem para Alice. Para confirmar o

Algoritmo 1.4: NS na semântica da Scyther

```
1  const pk: Function;
2  secret sk: Function;
3  inversekeys (pk,sk);
4  const Eve: Agent;
5  untrusted Eve;
6  protocol NS(Alice,Bob,Eve){
7     role Alice{
8        fresh nonceA: Nonce;
9        var nonceB: Nonce;
10       send_1(Alice,Bob,{Alice,nonceA}pk(Bob));
11       recv_2(Bob,Alice,{nonceA,nonceB}pk(Alice));
12       send_3(Alice,Bob,{nonceB}pk(Bob));
13       claim(Alice,Secret,nonceA);
14       claim(Alice,Secret,nonceB);
15       claim(Alice,Nisynch);
16    }
17    role Bob{
18       var nonceA: Nonce;
19       fresh nonceB: Nonce;
20       recv_1(Alice,Bob,{Alice,nonceA}pk(Bob));
21       send_2(Bob,Alice,{nonceA,nonceB}pk(Alice));
22       recv_3(Alice,Bob,{nonceB}pk(Bob));
23       claim(Bob,Secret,nonceA);
24       claim(Bob,Secret,nonceB);
25       claim(Bob,Nisynch);
26    }
27 }
```

recebimento do *nonce* `nonceB`, Alice responde ao atacante com {`nonceB`}`pk`{`Eve`}. O atacante decifra a mensagem, descobre o valor de `nonceB`, cifra e envia a mensagem para Bob. Bob confirma o valor de `nonceB` (autenticação) e acredita que está comunicando-se com Alice. A partir deste ponto, todas as mensagens trocadas entre Alice e Bob passam primeiro pelo atacante. Como pode ser observado, a execução do ataque é simples, porém, eficaz. Basta que Eve conheça a chave pública de Bob e intermedie a comunicação entre Alice e Bob.

```
./scyther.py --all-attacks --max-runs=5 protocolo_ns.spdl

Verification results:
claim id [NS,Alice1], Secret(nonceA)   : No attacks.
claim id [NS,Alice2], Secret(nonceB)   : No attacks.
claim id [NS,Alice3], Nisynch          : No attacks.
claim id [NS,Bob1],   Secret(nonceA)   : Exactly 1 attack.
claim id [NS,Bob2],   Secret(nonceB)   : Exactly 1 attack.
claim id [NS,Bob3],   Nisynch          : Exactly 1 attack.
```

Verificação 1.4. Execução e saída da Scyther para o Algoritmo 1.4

Vale observar que a falha de segurança do NS foi descoberta cerca de 17 anos após sua concepção com ajuda de ferramentas computacionais de análise de protocolo [Lowe 1996]. O protocolo NSL (ou NS-Lowe) corrige a falha como segue [Lowe 1995b, Lowe 1989, Cremers 2006]. Bob deve adicionar a sua identidade na resposta à primeira mensagem de Alice (*i.e.*, $nonceA, nonceB, Bob$), na linha 21 do Algo-

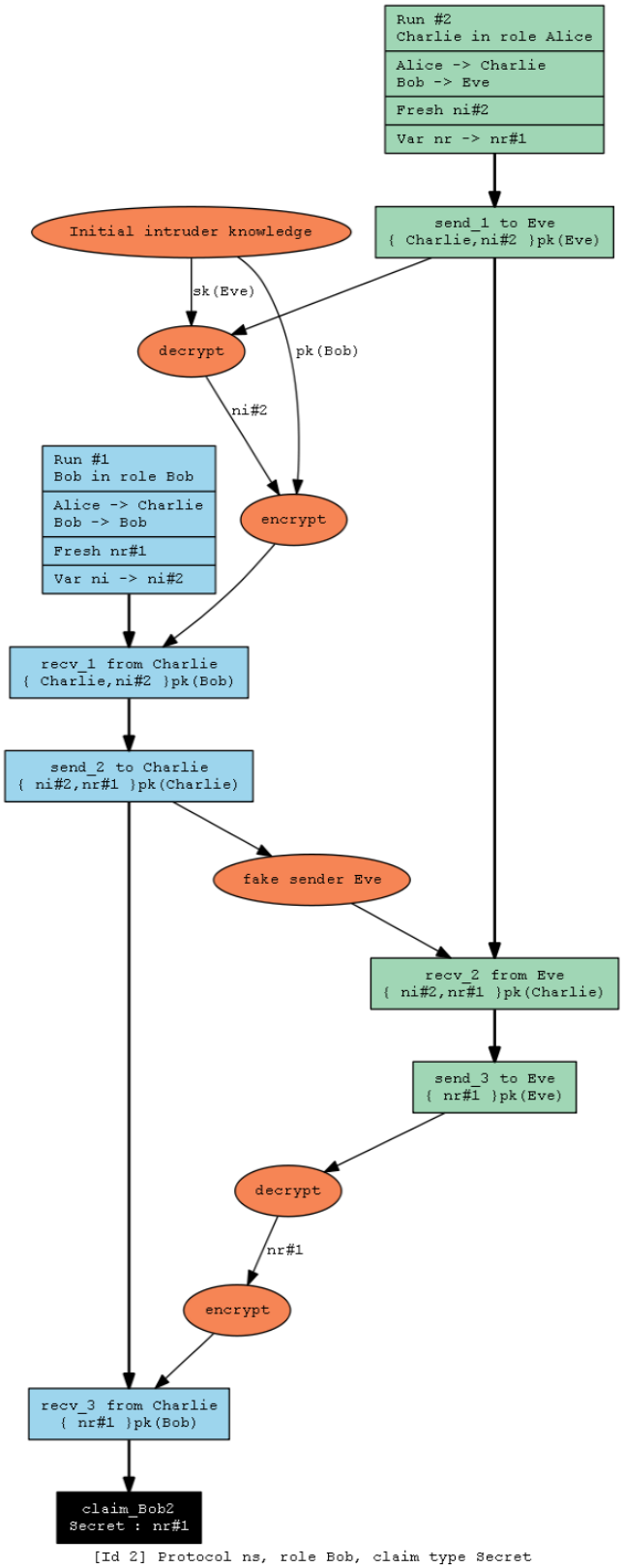

Figura 1.5. Diagrama do ataque ao protocolo NS

ritmo 1.4, como ilustrado no fluxograma da Figura 1.6. Com isso, Alice consegue descobrir que a identidade contida na mensagem é diferente da identidade fornecida por Eve, para quem está enviando as suas mensagens. Nesse caso, ao detectar tal inconsistência, Alice simplesmente encerra a troca de mensagens.

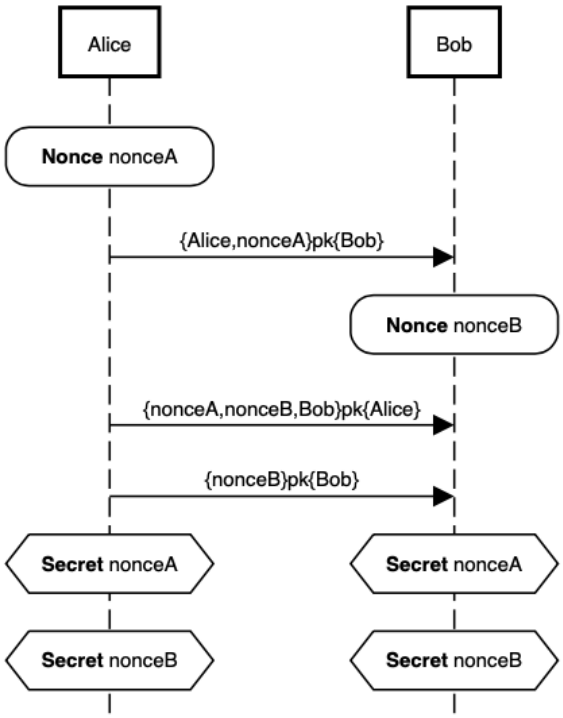

Figura 1.6. Diagrama do protocolo Needham-Schroeder-Lowe (NSL) [Cremers 2006]

Com o algoritmo corrigido (linha 21), o resultado da análise da Scyther pode ser visto na Verificação 1.5. Como pode ser observado, um simples detalhe de especificação pode comprometer toda a segurança do protocolo. Isso demonstra o quão importante é a utilização de ferramentas de verificação automática de protocolos.

```
./scyther.py --all-attacks --max-runs=5 protocolo_ns_corrigido.spdl

Verification results:
claim id [NS,Alice1], Secret(nonceA)  : No attacks.
claim id [NS,Alice2], Secret(nonceB)  : No attacks.
claim id [NS,Alice3], Nisynch         : No attacks.
claim id [NS,Bob1],   Secret(nonceA)  : No attacks.
claim id [NS,Bob2],   Secret(nonceB)  : No attacks.
claim id [NS,Bob3],   Nisynch         : No attacks.
```

Verificação 1.5. Execução e saída da Scyther para o Algoritmo 1.4 (corridigo)

1.6. O protocolo Generalized NSL public-key protocol (β)

1.6.1. Família de Protocolos para Autenticação entre Múltiplas Entidades

Em sistemas modernos é comum a necessidade de autenticação entre múltiplas (p) entidades, como no processo de atendimento de um técnico a um cliente, ambos coordenados por um gestor do Provedor de Acesso à Internet [Quincozes et al. 2020] (onde

$p = 3$). Uma maneira simples de obter autenticação mútua entre múltiplas entidades (*i.e.*, $p > 2$) é realizar autenticação aos pares usando protocolos de autenticação mútua projetados para duas entidades ($p = 2$). Seguindo esse processo, para realizar autenticação mútua entre p entidades são necessárias $(p * (p-1))/2$ instâncias de cada protocolo de autenticação mútua entre duas entidades, e pelo menos três mensagens para cada instância [Cremers and Mauw 2006].

Para minimizar o número de mensagens trocadas durante o processo de autenticação entre múltiplas entidades, foi proposto um arcabouço de protocolos [Cremers and Mauw 2006]. Esse arcabouço permite generalizar protocolos de autenticação mútua entre duas entidades ($p = 2$) para múltiplas entidades ($p > 2$) que desejam autenticar entre si. Um protocolo generalizado através do arcabouço permite autenticação mútua entre p entidades trocando $2p-1$ mensagens. Assim, por exemplo, para $p = 4$, a quantidade de mensagens a serem trocadas usando um protocolo NSL $(4*(4-1))/2*3 = 18$ é reduzida para $(2*4)-1 = 7$ usando protocolo NSL generalizado para quatro entidades.

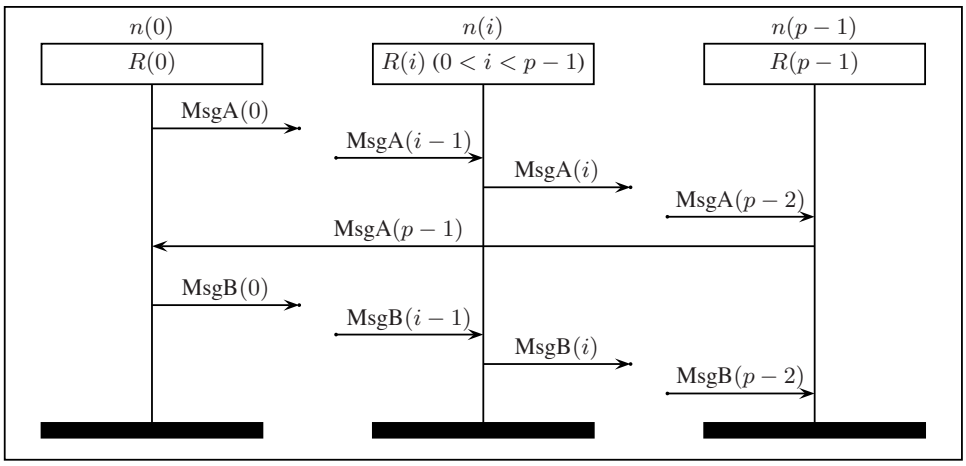

Figura 1.6. Estrutura de comunicação de múltiplas entidades [Cremers and Mauw 2006]

A Figura 1.6 apresenta uma visão geral do arcabouço de generalização. O arcabouço define uma coleção de p entidades $R_0, \cdots, R_{p-1}$ [Cremers and Mauw 2006]. Cada entidade controla uma mensagem **nounce** $n_0, \cdots, n_{p-1}$. A primeira entidade (R_0) encaminha o desafio para a segunda entidade (R_1), que, por sua vez, encaminha para a terceira entidade e assim sucessivamente até a última entidade (R_{p-1}) do sistema.

A primeira entidade (R_0) resolve seu desafio quando recebe a resposta da última entidade (R_{p-1}), confirmando que todas as outras entidades responderam ao seu desafio. A primeira entidade então encaminha as respostas restantes para a segunda entidade (R_1) encerrar o ciclo dela. A segunda entidade, então, encerra seu ciclo e encaminha as respostas restantes para a entidade seguinte. Esse processo se repete até a última entidade (R_{p-1}) encerrar seu próprio ciclo. Observe-se que as primeiras p mensagens (denotadas como MsgA) contém tanto desafios como respostas, e que as últimas $p-1$ mensagens (do tipo MsgB) contém apenas respostas dos desafios.

Há uma família de protocolos para demonstrar a aplicabilidade do arcabouço proposto [Cremers and Mauw 2006]. Cada membro da família corresponde a uma

generalização para múltiplas entidades de um protocolo de autenticação mútua entre duas entidades, incluindo Needham-Schroeder (NS), Needham-Schroeder-Lowe (NSL) e Bilateral Key Exchange (BKE).

1.6.2. Protocolo NSL Generalizado para Quatro Entidades

Por razões didáticas, vamos focar em apenas uma instância de um protocolo generalizado da referida família de protocolos. Especificamente, nesta seção vamos apresentar a generalização do NSL (*Generalized Needham-Schroeder-Lowe*) com chave pública, denotado como protocolo β da família, e considerando quatro entidades (*i.e.*, $p = 4$).

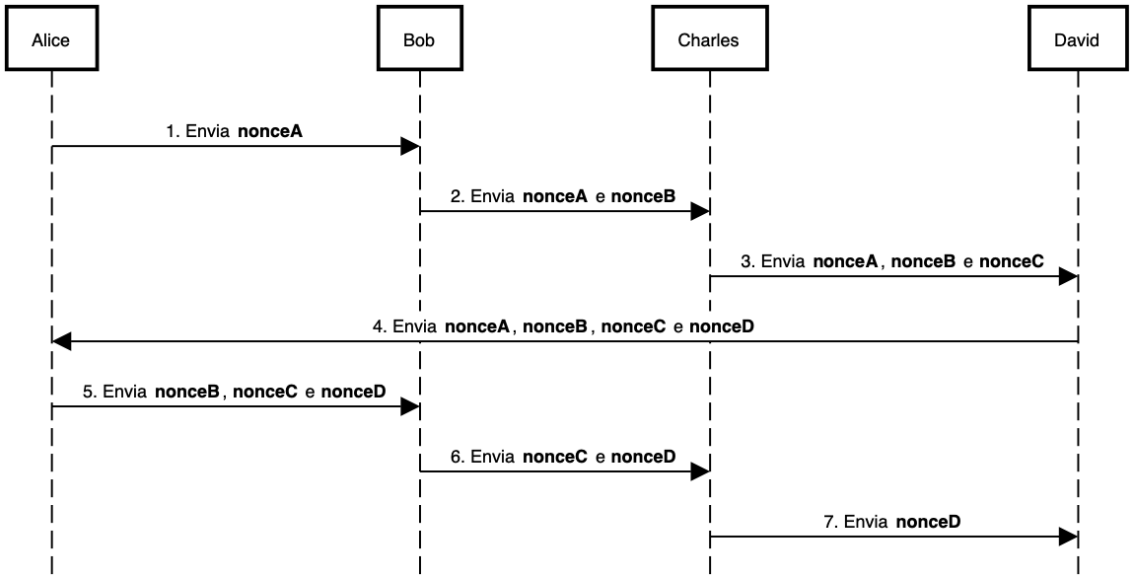

Figura 1.7. Diagrama de Alice, Bob, Charles e David utilizando o protocolo NSL generalizado

A Figura 1.7 e o Protocolo 1.6 ilustram e detalham o funcionamento do protocolo *Four-party Generalized NSL*, respectivamente. No primeiro ciclo, Alice envia uma mensagem contendo um desafio e a identificação das outras entidades para Bob cifrada com a chave pública do receptor (linha 1). Por sua vez, Bob decifra a mensagem com a sua chave privada e envia uma nova mensagem para Charles cifrada com a chave pública do receptor (linha 2). Essa mensagem contém, além do desafio de Alice, o desafio do Bob, e a identificação das outras entidades. Charles repete o processo parecido em uma comunicação com David (linha 3). O primeiro ciclo se encerra quando David envia uma mensagem para Alice contendo os desafios de todas entidades cifrada com a chave privada da receptora (linha 4).

É oportuno lembrar que o segundo ciclo permite encerrar o processo de desafio. Nesse sentido, Alice confere o seu desafio e envia para Bob uma mensagem contendo os desafios restantes (linha 5). Processos similares se repetem entre Bob e Charles (linha 6) e, por fim, entre Charles e David (linha 7).

1.6.3. Verificação automática do protocolo

O Algoritmo 1.5 apresenta uma instância do protocolo Generalized Four-Party Needham-Schroeder-Lowe na semântica da Scyther. Observe-se que há quatro entidades no pro-

Protocolo 1.6. Especificação do NS entre Alice e Bob

1. Alice → Bob	[E$_{pk_{Bob}}$(Alice,Charles,David,nonceA)]
2. Bob → Charles	[E$_{pk_{Charles}}$(Alice,Bob,David,nonceA,nonceB)]
3. Charles → David	[E$_{pk_{David}}$(Alice,Bob,Charles,nonceA,nonceB,nonceC)]
4. David → Alice	[E$_{pk_{Alice}}$(Bob,Charles,David,nonceA,nonceB,nonceC,nonceD)]
5. Alice → Bob	[E$_{pk_{Bob}}$(nonceB,nonceC,nonceD)]
6. Bob → Charles	[E$_{pk_{Charles}}$(nonceC,nonceD)]
7. Charles → David	[E$_{pk_{David}}$(nonceD)]

tocolo. O primeiro ciclo de mensagem foi denotado com sufixo _11, _12, _13, _14 e o segundo ciclo com sufixo _21, _22 e _23.

Algoritmo 1.5: *Four-party Generalized NSL* na semântica da Scyther

```
1  protocol 4GNSL(Alice, Bob, Charles, David){
2    role Alice {
3      fresh nounceA: Nonce;
4      var nounceB, nounceC, nounceD: Nonce;
5      send_11(Alice,Bob, {nounceA, Alice, Charles, David}pk(Bob));
6      recv_14(David,Alice, {nounceA, nounceB, nounceC, nounceD, Bob,Charles,David}pk(Alice));
7      send_21(Alice, Bob, {nounceB, nounceC, nounceD}pk(Bob));
8    }
9    role Bob {
10     fresh nounceB: Nonce;
11     var nounceA, nounceC, nounceD: Nonce;
12     recv_11(Alice,Bob, {nounceA, Alice, Charles, David}pk(Bob));
13     send_12(Bob,Charles, {nounceA, nounceB, Alice, Bob, David}pk(Charles));
14     recv_21(Alice, Bob, {nounceB, nounceC, nounceD}pk(Bob));
15     send_22(Bob, Charles, {nounceC, nounceD}pk(Charles));
16   }
17   role Charles {
18     fresh nounceC: Nonce;
19     var nounceA, nounceB, nounceD: Nonce;
20     recv_12(Bob,Charles, {nounceA, nounceB, Alice, Bob, David}pk(Charles));
21     send_13(Charles, David, {nounceA, nounceB, nounceC, Alice, Bob, Charles}pk(David));
22     recv_22(Bob, Charles, {nounceC, nounceD}pk(Charles));
23     send_23(Charles, David, {nounceD}pk(David));
24   }
25   role David {
26     fresh nounceD: Nonce;
27     var nounceA, nounceB, nounceC: Nonce;
28     recv_13(Charles, David, {nounceA, nounceB, nounceC, Alice, Bob, Charles}pk(David));
29     send_14(David, Alice, {nounceA, nounceB, nounceC, nounceD, Bob, Charles, David}pk(Alice));
30     recv_23(Charles, David, {nounceD}pk(David));
31   }
32 }
```

A Verificação 1.6 apresenta o resultado da avaliação do protocolo acima usando a ferramenta Scyther com parâmetros `--auto-claims` e `--max-runs=7`. Observe-se que é possível obter prova de corretude para alguns claims gerado pelo Scyther enquanto

para outros claims não foram detectados ataques dentro do número de rodadas empregrado.

```
./scyther.py --auto-claims --all-attacks --max-runs=7 protocolo_4gnsl.spdl

Verification results:
claim id [4GNSL,Alice1], Secret(nonceA)   : Proof of correctness
claim id [4GNSL,Alice2], Secret(nonceD)   : No attacks within bounds.
claim id [4GNSL,Alice3], Secret(nonceC)   : No attacks within bounds.
claim id [4GNSL,Alice4], Secret(nonceB)   : Proof of correctness.
claim id [4GNSL,Alice5], Alive            : Proof of correctness.
claim id [4GNSL,Alice6], Weakagree        : Proof of correctness.
claim id [4GNSL,Alice7], Niagree          : Proof of correctness.
claim id [4GNSL,Alice8], Nisynch          : Proof of correctness.
claim id [4GNSL,Bob1], Secret(nonceB)     : No attacks within bounds.
claim id [4GNSL,Bob2], Secret(nonceD)     : No attacks within bounds.
claim id [4GNSL,Bob3], Secret(nonceC)     : No attacks within bounds.
claim id [4GNSL,Bob4], Secret(nonceA)     : No attacks within bounds.
claim id [4GNSL,Bob5], Alive              : No attacks within bounds.
claim id [4GNSL,Bob6], Weakagree          : No attacks within bounds.
claim id [4GNSL,Bob7], Niagree            : No attacks within bounds.
claim id [4GNSL,Bob8], Nisynch            : No attacks within bounds.
claim id [4GNSL,Charles1], Secret(nonceC) : No attacks within bounds.
claim id [4GNSL,Charles2], Secret(nonceD) : No attacks within bounds.
claim id [4GNSL,Charles3], Secret(nonceB) : No attacks within bounds.
claim id [4GNSL,Charles4], Secret(nonceA) : No attacks within bounds.
claim id [4GNSL,Charles5], Alive          : No attacks within bounds.
claim id [4GNSL,Charles6], Weakagree      : No attacks within bounds.
claim id [4GNSL,Charles7], Niagree        : No attacks within bounds.
claim id [4GNSL,Charles8], Nisynch        : No attacks within bounds.
claim id [4GNSL,David1], Secret(nonceD)   : No attacks within bounds.
claim id [4GNSL,David2], Secret(nonceC)   : No attacks within bounds.
claim id [4GNSL,David3], Secret(nonceB)   : No attacks within bounds.
claim id [4GNSL,David4], Secret(nonceA)   : No attacks within bounds.
claim id [4GNSL,David5], Alive            : No attacks within bounds.
claim id [4GNSL,David6], Weakagree        : No attacks within bounds.
claim id [4GNSL,David7], Niagree          : No attacks within bounds.
claim id [4GNSL,David8], Nisynch          : No attacks within bounds.
```

Verificação 1.6. Execução e saída da Scyther para o Algoritmo 1.5

1.7. Considerações Finais

Neste minicurso, nós apresentamos uma introdução às semânticas operacionais e a utilização prática da ferramenta de verificação automática de protocolos de segurança Scyther (Seção 1.2). Para o desenvolvimento do minicurso, foram selecionados seguintes protocolos: o Atualização de Chave Simétrica (ACS), na Seção 1.3; o Wide Mouth Frog (WMF), na Seção 1.4; o Needham-Schroeder (NS), na Seção 1.5; e o Generalized NSL public-key protocol (β), na Seção 1.6. Os protocolos foram apresentados de maneira didática, incluindo representação através de fluxogramas simplificados e uma semântica genérica de especificação de protocolos de segurança, que é independente das semânticas operacionais e linguagens das ferramentas de verificação.

Para cada um dos protocolos utilizados como exemplo, foi detalhado o algoritmo de verificação automática na semântica operacional da ferramenta Scyther. As verificações automáticas (ver Verificações 1.1, 1.2, 1.3, 1.4, 1.5, 1.6) demonstraram

falhas de protocolos como o WMF e o NS, que foram discutidas e corrigidas através da modificação de mensagens ou da introdução de versões atualizadas do protocolo, como o WMF-Lowe (ver Protocolo 1.4 na Seção 1.4).

Acreditamos que o minicurso tenha cumprido o seu objetivo principal, isto é, fazer com que estudantes e profissionais percebam a importância de ferramentas de verificação automática e formal de protocolos de segurança. Por exemplo, como comentado anteriormente na Seção 1.5, a falha do NS foi descoberta apenas 17 anos após sua concepção do protocolo. Outros exemplos práticos similares, recentes, podem ser encontrados na literatura.

Agradecimentos

Agradecemos ao editor Gustavo Griebler, da Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação (ReABTIC)[10], pela autorização expressa de re-utilização integral do conteúdo do artigo "Verificação Automática dos Protocolos de Segurança Needham-Schroeder, WMF e CSA com a ferramenta Scyther" [Jenuario et al. 2020], de autoria de co-autores deste minicurso. As Seções 1.4 e 1.5 deste minicurso, apesar de apresentarem novos dados e conteúdos, foram baseadas no conteúdo original do artigo.

Referências

Affeldt, R. and Marti, N. (2013). Towards Formal Verification of TLS Network Packet Processing Written in C. In *7th PLPV*, pages 35–46. ACM.

Arapinis, M., Mancini, L., Ritter, E., Ryan, M., Golde, N., Redon, K., and Borgaonkar, R. (2012). New privacy issues in mobile telephony: fix and verification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 205–216.

Armando, A., et al. (2005). The AVISPA tool for the automated validation of internet security protocols and applications. In *International conference on computer aided verification*, pages 281–285. Springer.

Atkinson, R. (1995). Security Architecture for the Internet Protocol. RFC 1825.

Baelde, D., Delaune, S., Gazeau, I., and Kremer, S. (2017). Symbolic verification of privacy-type properties for security protocols with xor. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 234–248. IEEE.

Bai, X., Cheng, Z., Duan, Z., and Hu, K. (2018). Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ICSCA 2018, page 322–326, New York, NY, USA. Association for Computing Machinery.

Blanchet, B. (2006). A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, Oakland, California.

Blanchet, B., Smyth, B., Cheval, V., and Sylvestre, M. (2018). ProVerif 2.00: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial. https://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf.

Burrows, M. (1989). Wide mouthed frog. http://www.lsv.fr/Software/spore/wideMouthedFrog.html.

[10] https://revistas.setrem.com.br/index.php/reabtic

Burrows, M., Abadi, M., and Needham, R. (1990). A logic of authentication. *ACM TRANSACTIONS ON COMPUTER SYSTEMS*, 8:18–36.

Carrel, D. and Harkins, D. (1998). The Internet Key Exchange (IKE). RFC 2409.

Chen, S., Fu, H., and Miao, H. (2016). Formal verification of security protocols using spin. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6. IEEE.

Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., and Westbrook, E. (2018). Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification*, pages 430–446.

Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., and Stebila, D. (2020). A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983.

Cook, B. (2018). Formal reasoning about the security of amazon web services. In *International Conference on Computer Aided Verification*, pages 38–47. Springer.

Cremers, C., Horvat, M., Scott, S., and v. d. Merwe, T. (2016). Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *IEEE SP*.

Cremers, C. and Mauw, S. (2006). A family of multi-party authentication protocols. In *First Benelux Workshop on Information and System Security (WISSec)*.

Cremers, C. J. (2008). The scyther tool: Verification, falsification, and analysis of security protocols. In *International conference on computer aided verification*, pages 414–418. Springer.

Cremers, C. J. F. (2006). *Scyther: Semantics and verification of security protocols*. Eindhoven University of Technology Eindhoven. https://doi.org/10.6100/IR614943.

Dalal, N., Shah, J., Hisaria, K., and Jinwala, D. (2010). A comparative analysis of tools for verification of security protocols. *Int. J. of Comm., Network and System Sciences*, 3(10):779.

Delia Jurcut, A., Liyanage, M., Chen, J., Gyorodi, C., and He, J. (2018). On the security verification of a short message service protocol. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6.

Jenuario, T., Chervinski, J. O., Paz, G., Fernandes, R., Beltran, R., and Kreutz, D. (2020). Verificação Automática dos Protocolos de Segurança Needham-Schroeder, WMF e CSA com a ferramenta Scyther. *Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação (ReABTIC)*, pages 1–11. http://dx.doi.org/10.5281/zenodo.3833260.

Kelsey, J., Schneier, B., and Wagner, D. (1997). Protocol interactions and the chosen protocol attack. In *International Workshop on Security Protocols*, pages 91–104. Springer.

Krawczyk, D. H., Bellare, M., and Canetti, R. (1997). HMAC: Keyed-Hashing for Message Authentication. RFC 2104.

Kreutz, D., Yu, J., Ramos, F. M. V., and Esteves-Verissimo, P. (2019). ANCHOR: Logically centralized security for software-defined networks. *ACM Trans. Priv. Secur.*, 22(2):8:1–8:36.

Li, L., Sun, J., Liu, Y., Sun, M., and Dong, J. (2018). A Formal Specification and Verification Framework for Timed Security Protocols. *IEEE Trans. on Soft. Engineering*, 44(8):725–746.

Liu, J. and Liu, Z. (2019). A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894–77904.

Lonvick, C. M. and Ylonen, T. (2006). The Secure Shell (SSH) Transport Layer Protocol. RFC 4253.

Lowe, G. (1989). Lowe's fixed version of needham-schroder public key. http://www.lsv.fr/Software/spore/nspkLowe.html.

Lowe, G. (1995a). An Attack on the Needham- Schroeder Public- Key Authentication Protocol. *Information processing letters*, 56(3).

Lowe, G. (1995b). An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133.

Lowe, G. (1996). Breaking and fixing the needham-schroeder public-key protocol using fdr. In Margaria, T. and Steffen, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lowe, G. (1997a). A family of attacks upon authentication protocols. Technical Report 1997/5, Department of Mathematics and Computer Science, University of Leicester. http://www.cs.ox.ac.uk/gavin.lowe/Security/Papers/multiplicityTR.ps.

Lowe, G. (1997b). Lowe modified wide mouthed frog. http://www.lsv.fr/Software/spore/wideMouthedFrogLowe.html.

Meier, S., Schmidt, B., Cremers, C., and Basin, D. (2013). The tamarin prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 696–701. Springer.

Needham, R. M. and Schroeder, M. D. (1978). Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999.

Quincozes, V. E., Temp, D., Quincozes, S. E., Kreutz, D., and Mansilha, R. B. (2020). Sistema para Autenticação entre Clientes, Técnicos e ISPs. In *5o Workshop Regional de Segurança da Informação e de Sistemas Computacionais*, Joinville-SC, Brasil.

Rainer, D., Matthias, P., and Helmut, K. (2020). A comprehensive model of information security factors for decision-makers. *Computers & Security*, 92:101747.

Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446.

Steiner, M., Tsudik, G., and Waidner, M. (1996). Diffie-hellman key distribution extended to group communication. In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 31–37.

Velibor, S., Ivana, O., and Ramo, S. (2016). Comparative analysis of some cryptographic systems. Technical Report 15, Business Information Security Conference.