Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Verificación automática de protocolos criptográficos de seguridad

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Nataniel Renzo Rondo Van Ysseldyk

Tutor: Dr. Santiago Escobar Román

2015/2016

Resumen

En el presente proyecto se estudiarán, explicarán y aplicarán las nuevas posibilidades que nos proporciona la introducción de mejoras en la herramienta de verificación de protocolos Maude-NPA. Estas mejoras nos permitirán analizar y verificar la seguridad de protocolos de comunicación, utilizando ataques que exploten estas nuevas mejoras. En concreto nos centraremos en la asociatividad, siendo ésta una nueva característica, y en cómo añadiendo esta característica existen protocolos que siendo seguros pierden esta propiedad.

Para ello utilizaremos Maude-NPA, una herramienta de verificación de protocolos de comunicación con propiedades criptográficas, desarrollada por el profesor Santiago Escobar (Universitat Politècnica de València), en colaboración con el profesor José Meseguer (University of Illinois at Urbana-Champaign, USA) y la profesora Catherine Meadows (Naval Research Laboratory, USA).

Esta herramienta gracias a su capacidad para verificar protocolos complejos de forma automática, nos permitirá analizar cómo se desarrollan estos ataques y en qué puntos se producen las vulnerabilidades, y así también poder comprobar la efectividad de la herramienta en cuestión al ser los ataques conocidos.

Palabras clave: Maude-NPA, Protocolos de comunicación, verificación, modelado, asociatividad, ataques, criptografía.

Abstract

In this project will be studied , explain and apply the new possibilitiesIt provides improvements in protocol verification toolMaude -NPA .These improvements will allow us to analyze and verify security protocols communication , using attacks that exploit these new enhancements . Specifically we focus on associativity , and this is a new feature and how adding thisthere are protocols that feature remain safe lose this property.

We will use Maude -NPA , a verification tool protocols communication with cryptographic properties, developed by Professor Santiago Escobar (Polytechnic University of Valencia) , in collaboration with Professor José Meseguer(University of Illinois at Urbana- Champaign , USA) and Professor Catherine Meadows (Naval Research Laboratory, USA) .

This tool thanks to its ability to verify complex protocols so automatic , allow us to analyze how these attacks take place and at what points produce vulnerabilities , and thus also to check the effectiveness of the tool in question to be known attacks .

Keywords: Maude-NPA, protocols communication, verification, cryptography, attacks, associativity

Índice general

Índice de ilustraciones

1.Introducción.

En el primer apartado, explicaremos qué ha motivado el desarrollo de este proyecto, asignaturas que han sido un apoyo para la realización del proyecto, los objetivos que se pretenden alcanzar y, por último, explicaremos cómo se organizará la memoria en los siguientes apartados.

1.1. Motivación.

Hoy en día la verificación de protocolos de comunicación es imprescindible para poder asegurar la robustez de los mismos. Para ello se deben cumplir dos condiciones: que se cumplan las postcondiciones al ejecutar cada una de las acciones de los participantes especificadas en el protocolo, o bien que se pueda garantizar la autenticidad de cada participante. Estos protocolos en ocasiones son propensos a errores en su creación, la mayoría de errores son provocados por situaciones no esperadas. En la práctica se pueden encontrar errores al suponer situaciones en las que se den unas condiciones que permitan el ataque.

Muchas herramientas se han fijado el objetivo de conseguir no solo verificar protocolos sino poder probar distintas características criptográficas que pueden poseer estos protocolos. Actualmente Maude-NPA [1] es la única que nos garantiza resultados deterministas en los que podemos asegurar que las soluciones son finitas, aunque no en todos los protocolos de comunicación esto es posible.

La principal motivación del proyecto ha sido estudiar cómo con la herramienta Maude-NPA, podemos verificar distintos protocolos y probar gracias a la nueva versión de la herramienta una propiedad en concreto, la asociatividad, y cómo esta propiedad influye en la robustez de los mismos. Para ello utilizaremos protocolos que sabemos que tienen ataques dependientes de esta propiedad.

Gracias a la información disponible podremos comprobar si la herramienta se comporta de la forma esperada al utilizar esta nueva propiedad ya que ésta puede simularse mediante otros métodos menos potentes (reglas ecuacionales). También podremos asegurar que las soluciones sean las únicas posibles, esto tiene una gran importancia al tratarse de protocolos tan complejos como los de clave simétrica o con interacción de un servidor.

Por otra parte, podremos ver cómo se expande el árbol de búsqueda en el que se generan diferentes estados hasta encontrar un estado que coincida con las condiciones para ser estado final, esto nos permite observar la evolución dependiendo de las acciones que tomen los participantes, también nos permitirá ver los costes tanto espaciales como temporales al aumentar la profundidad de búsqueda.

Por último, gracias al estudio del comportamiento de la herramienta y de los resultados obtenidos podremos especular con nuevos experimentos en los que podamos explotar aún más esta propiedad lo cual no es posible con ninguna otra herramienta, y en cómo la posibilidad de añadir nuevas propiedades nos aportara aún más información de los protocolos a estudiar.

1.2. Objetivos.

Actualmente ninguna herramienta, a excepción de Maude-NPA, nos permite verificar diversos protocolos de seguridad, expandiendo todo el árbol de búsqueda en profundidad hasta en encontrar un estado final que cumpla las condiciones especificadas, y a partir de este ejecutar un recorrido hacia atrás hasta el inicio. También nos permite ver cuántos estados hay en cada momento, gracias a esto podemos ver cómo aumenta tanto el coste temporal como el espacial, y por último ver si se ha llegado a un punto en el que todos los estados sean solución.

Otra característica única de Maude-NPA, y por tanto objetivo de estudio, es la capacidad de estudiar cómo agregar ciertas propiedades a protocolos aparentemente seguros, y cómo afecta a su robustez, provocando ataques. Nosotros nos centraremos en la propiedad de asociatividad, sin ninguna otra propiedad más (antes se permitía asociatividad y conmutatividad juntas).

Para analizar la influencia de esta propiedad en un protocolo de clave simétrica usaremos un ataque conocido al protocolo Needham-Schroeder-Lowe mediante asociatividad, y estudiaremos cómo la herramienta puede revelar el secreto utilizando acciones que involucren la asociatividad. Por otra parte, para estudiar la vulnerabilidad de un protocolo en el que se utiliza un servidor, usaremos el protocolo Secret2016, el cual sabemos que tiene un ataque mediante asociatividad. Utilizando estos dos protocolos podremos confirmar que la herramienta se comporta de la forma esperada al utilizar el axioma ecuacional en lugar de las reglas ecuacionales.

1.3. Asignaturas relacionadas.

Para este proyecto hemos utilizado como apoyo conocimientos adquiridos en otras asignaturas a lo largo del grado, primeramente la asignatura de Redes de Computadores nos proporcionó el conocimiento sobre la especificación clásica en protocolos de comunicación (Alice y Bob) y también proporcionaron un conocimiento inicial sobre protocolos de clave simétrica. Por otra parte la asignatura Tecnologías y Paradigmas de la Programación, nos ha permitido entender cómo funciona el lenguaje Maude, ya que en ella se estudiaron lenguajes interpretados y los paradigmas de los lenguajes funcionales y declarativos como Haskell (funcional) y Prolog (lógico).

Otros conocimientos importantes son los adquiridos en la asignatura Ingeniería del Software (ISW), como los conceptos de modelado de protocolos mediante diagramas, así como la importancia de la verificación de software, protocolos, etc. Para entender los conceptos de determinismo e indeterminismo necesario a la hora de estudiar la expansión del árbol de búsqueda, para poder determinar cuándo las soluciones son únicas, y finitas, concepto de suma importancia.

Por último, la asignatura optativa de criptografía, ha sido de mucha utilidad al explicar en ella algunos de los protocolos, entre ellos uno de los que trataremos en este proyecto el Needham-Schroeder-Lowe, así como conceptos relacionados con la seguridad como los nonces o los números pseudoaleatorios que se utilizan para construir estos nonce.

1.4. Estructura de la Memoria.

A continuación, pasaremos a explicar cómo hemos decidido organizar la estructura de la memoria, en qué apartados hemos distribuido el contenido y qué información se encuentra en cada apartado, para hacer la búsqueda de la misma más rápida, la distribución se hará en 7 apartados:

-Introducción: En esta introducción hemos explicado la motivación, los protocolos a estudiar, las asignaturas que nos han servido como base para poder entender conceptos implicados en el proyecto y la estructura que se utilizara en cada apartado de la memoria.

-Maude-NPA: En este apartado explicaremos la herramienta Maude-NPA, las diferentes posibilidades que nos ofrece la última versión, cómo se especifican los protocolos en Maude-NPA, en cada uno de los diferentes apartados del fichero y cómo se especifican cada una de las acciones, participantes y propiedades que poseerá el protocolo. Por último explicaremos cómo interpretar distintos apartados de la salida y cómo realizar la ejecución para obtener los objetivos deseados.

-Protocolo Needham-Schroeder-Lowe (NSL): En este apartado primero presentaremos el protocolo NSL original sin ataques por asociatividad, después modelaremos el protocolo utilizando la asociatividad mediante una regla ecuacional, continuaremos con el modelo de NSL con asociatividad utilizando axiomas ecuacionales pero sin límites en los tipos y por último realizaremos una versión del modelo anterior en la que hemos utilizado tipos que limitan el espacio de búsqueda.

-Protocolo Secret2016: En este apartado, primero explicaremos cómo se desarrolla el protocolo Secret2016, identificando los participantes legítimos, el intruso,

así como las acciones que es capaz de desarrollar cada uno de los participantes y por último en qué consiste un ataque a este protocolo. Continuaremos realizando una versión del protocolo en la que la asociatividad se implemente con regla ecuacional y finalmente realizaremos una versión en la que la asociatividad se implemente mediante un axioma ecuacional, compararemos las respuestas de los árboles de búsqueda y estados de ataque dados por la herramienta para poder así comparar las dos versiones del protocolo.

-Conclusiones y trabajo futuro: En este apartado para finalizar presentaremos las conclusiones a las que hemos llegado utilizando los resultados de la investigación realizada, esquematizaremos los resultados obtenidos para clarificar el fruto del proyecto y por último presentaremos objetivos futuros a realizar gracias al conocimiento adquirido en este proyecto.

2. Maude-NPA.

2.1. Especificación de protocolos en Maude-NPA.

En esta sección describiremos cómo se especifica un protocolo, explicando cada uno de los módulos Maude utilizados, participantes en el protocolo, ataques, etc. Utilizaremos un ejemplo como plantilla, de esta forma el lector podrá ver cada uno de los apartados de una forma más clara.

2.1.1. Organización del protocolo en el fichero.

El protocolo se modela mediante un único fichero que estará dividido en tres módulos Maude [2], con el nombre y el formato ya fijado. El primero contendrá la sintaxis del protocolo: tipos, subtipos y operadores, el segundo módulo especificará las propiedades algebraicas de los operadores y por último el tercer módulo especificará el comportamiento de cada uno de los participantes legítimos [3] y del intruso (Strands Dolev-Yao). Por último se describirán los estados de ataque que modelan una situación en la que la seguridad del protocolo se ve comprometida.

El fichero utilizará la sintaxis del lenguaje Maude [2], que tiene como característica que es casi auto-explicativa. Otros puntos a remarcar son los comentarios precedidos por tres guiones y los finales de declaración que son expresados como un espacio y un punto. La plantilla base para la especificación será la siguiente:

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
protecting DEFINITION-PROTOCOL-RULES .
------------------------------------------------------------
---Overwrite this module with the syntax of your protocol.
---Notes
--* Sorts Msg and Fresh are special and imported
---* Every sort must be a subsort of Msg
---* No sort can be a supersort of Msg
---* Variables of sort Fresh are really fresh 14
---and no substitution is allowed on them
---* Sorts Msg and Public cannot be empty
------------------------------------------------------------
endfm
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
protecting PROTOCOL-EXAMPLE-SYMBOLS
------------------------------------------------------------------
---Overwrite this module with the algebraic properties
---of your protocol.
---* Use only equations of the form (eq Lhs = Rhs [nonexec] .)
---* Maude attribute owise cannot be used
---* There is no order of application between equations
```

Ilustración 1: Plantilla de protocolo (Parte 1).

```
---------------------------------------------------------------
endfm
fmod PROTOCOL-SPECIFICATION is
protecting PROTOCOL-EXAMPLE-SYMBOLS .
protecting DEFINITION-PROTOCOL-RULES .
protecting DEFINITION-CONSTRAINTS-INPUT .
-----------------------------------------------------------
---Overwrite this module with the Strands
---of your protocol and the attack states
-----------------------------------------------------------
eq STRANDS-DOLEVYAO =
---Add Dolev-Yao intruder strands here. Strands are
---properly renamed.
[nonexec] .
eq STRANDS-PROTOCOL =
---Add protocol strands here.
 Strands are properly renamed.
[nonexec] .
eq ATTACK-STATE(0) =
---Add attack state here
---More than one attack state can be specified, but each
---must be identified by a number (e.g. ATTACK-STATE(1) = ...
---ATTACK-STATE(2) = ... etc.)
[nonexec] .
endfm
---THE FOLLOWING COMMAND HAS TO BE THE LAST ACTION !!!!
select MAUDE-NPA .
```

Ilustración 2: Plantilla de protocolo (Parte 2).

2.1.2 Sintaxis del protocolo.

Como hemos explicado en la sección anterior este es el primer módulo Maude dentro del fichero. En el explicaremos los tipos, subtipos y operadores, para especificar los tipos de datos usaremos sorts para los mensajes usaremos el tipo “Msg”, para identidades el tipo “Name”, y para las claves el tipo “Key”.

Para especificar cuando un tipo es subtipo de otro utilizaremos subsorts. De esta forma podremos decir que por ejemplo un tipo Key puede tener dos subtipos PubliKey y PrivateKey.

```
subsort PublicKey PrivateKey < Key .
```

Ilustración 3: 1º Ejemplo de Tipos y Subtipos.

Este módulo tiene un nombre fijo “PROTOCOL-EXAMPLE-SYMBOLS” y tendrá restricciones para ciertos tipos propios:

-Msg: Sera el tipo raíz de todo tipo definido por el usuario en el protocolo. No puede haber ningún supertipo de Msg, a su vez el tipo Msg no podrá ser vacío.

-Public: Este tipo indicará términos conocidos por el intruso. Ningún tipo podrá ser supertipo de Public y al igual que Msg no puede ser vacío.

-Fresh: Este tipo indicara variables que corresponden a un número único, es de vital importancia en elementos de seguridad como Nonces o claves ya que son únicos de cada participante.

Así pues, usando de ejemplo un protocolo básico como el Needham-Schroeder mostraremos como se modelan los tipos y subtipos.

```
--- Sort Information
sorts Name Nonce Key .
subsort Name Nonce Key < Msg .
subsort Name < Key .
subsort Name < Public .
```

Ilustración 4: 2º Ejemplo de Tipos y Subtipos.

A continuación, pasamos a describir los operadores. Los primeros serán los que nos permitan encriptar el mensaje:

```
--- Encoding operators for public/private encryption
op pk : Key Msg -> Msg [frozen] .
op sk : Key Msg -> Msg [frozen] .
```

Ilustración 5: Operador de cifrado.

Después necesitaremos un operador que nos permita generar Nonces utilizando para ello la identidad del participante (Name) y un tipo de dato “Fresh”.

```
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
```

Ilustración 6: Operador generador de Nonces.

Utilizaremos operadores para especificar los nombres de los participantes. En la mayoría se utilizarán tres, pero esto no quiere decir que solo puedan haber tres pues como veremos analizando el protocolo Needham-Schroeder-Lowe se pueden iniciar sesiones ilimitadas mediante la herramienta utilizando variables en lugar de constantes, aunque el uso de constantes sigue siendo necesario en situaciones en las que querremos asignar roles específicos. Podremos ver como se aplica esto cuando analicemos los Strands.

```
--- Principals
op a : -> Name . --- Alice
op b : -> Name . --- Bob
op i : -> Name . --- Intruder
```

Ilustración 7: Operadores para generar participantes.

Por último especificaremos un operador que nos permita concatenar elementos.

```
--- Concatenation operator
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
```

Ilustración 8: Operador concatenación.

De esta forma podremos concatenar elementos usando el operador ";" que como hemos especificado será infijo.

2.1.3. Propiedades algebraicas del protocolo.

Existen dos tipos de propiedades algebraicas: axiomas ecuacionales (conmutatividad, asociatividad) que se especifican en los propios operadores y reglas ecuacionales las cuales se especifican en el apartado "PROTOCOL-EXAMPLE-ALGEBRAIC".

Nosotros utilizaremos ambas ya que el poder utilizar la asociatividad como axioma ecuacional es una característica añadida en la versión 2.7.1 de Maude, por ello utilizaremos las reglas ecuacionales para poder utilizar la asociatividad y así poder comparar esta nueva capacidad de poder declarar la asociatividad como axioma ecuacional con los métodos anteriores.

Las ecuaciones definen un comportamiento en el que la parte izquierda de la ecuación se reduce en la parte derecha, para ello se reescribe. Un ejemplo lo encontramos en las ecuaciones que definen la relación entre la clave privada y la pública en el protocolo NSPK:

```
var Z : Msg .
var Ke : Key .

*** Encryption/Decryption Cancellation
eq pk(Ke,sk(Ke,Z)) = Z [variant] .
eq sk(Ke,pk(Ke,Z)) = Z [variant] .
```

Ilustración 9: Ejemplo del uso de reglas ecuacionales

En el caso de los axiomas ecuacionales Maude-NPA utiliza algoritmos de unificación empotrados [4], siendo el último en ser añadido la asociatividad, aunque en versiones anteriores podía utilizarse en conjunto con la conmutatividad, ésta es la primera vez que podemos utilizarla por si sola.

Para ello deberemos especificarla en la declaración de los operadores en el módulo "PROTOCOL-EXAMPLE-SYMBOLS" a continuación podemos ver cómo se implementaría esto en el protocolo Needham-Schroeder-Lowe

```
--- Associativity operator
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen assoc] .
op _;_ : Nonce Name  -> Data [gather (e E) frozen assoc] .
```

Ilustración 10: Operador concatenación, con asociatividad

2.1.4. Strands legítimos.

En este apartado analizaremos los comportamientos de los participantes, a partir de ahora Strands, éstos se especificarán en el tercer y último modulo. Aquí se definirán los mensajes que envían y reciben cada uno de los participantes, debemos recordar que cada Strand puede ser instanciado un número infinito de veces. Para ilustrar mejor esta sección utilizaremos de ejemplo los Strands utilizados en el protocolo Needham-Schroeder Public Key (NSPK) indicando cada uno de los participantes y explicando su comportamiento.

```
:: r ::
[ nil |  +(pk(B,A ; n(A,r))),
         -(pk(A,n(A,r) ; N)),
         +(pk(B, N)),
         nil ]
```

Ilustración 11: Strand de Alice en NSPK

Comenzaremos analizando el Strand de Alice que en este caso es el iniciador, podemos ver que los mensajes salientes se indican mediante un símbolo "+" y los mensajes entrantes con un "-". El protocolo empieza cuando Alice envía, de forma codificado con la clave pública de Bob, su identidad y su nonce, el cual se genera con su identidad y una variable de tipo Fresh utilizando el operador "n" descrito en secciones anteriores. Después esperará recibir un mensaje codificado con su clave pública en el que se encuentra su propia identidad el Nonce que envió en un principio y un elemento de tipo de Nonce. Por último envía un mensaje cifrado con la clave pública de Bob en el que se encuentra el Nonce recibido en el anterior mensaje.

```
:: r ::
[ nil |  -(pk(B,A ; N)),
         +(pk(A,N ; n(B,r))),
         -(pk(B, n(B,r))),
         nil ]
```

Ilustración 12: Strand de Bob en NSPK

Esta vez podemos observar que los mensajes siguen un orden inverso, al Strand de Alice, primero recibe la identidad de Alice y un Nonce cifrado con su propia clave pública, luego envía un mensaje Alice con su propio Nonce y el que recibió en el paso anterior, ambos cifrados con la clave pública de Alice, para por último recibir su propio Nonce.

Los tipos de las variables utilizadas por tanto serán los siguientes:

```
var Ke : Key .
vars X Y Z : Msg .
vars r r' : Fresh .
vars A B : Name .
vars N N1 N2 : Nonce .
```

Ilustración 13: Variables utilizadas en los Strands.

Por último tenemos que recalcar que en el mensaje se utilizan variables, no constantes, esto permite que se puedan generar múltiples instancias de cada miembro y es de utilidad para ataques como el que veremos más adelante.

2.1.5. Reglas Dolev-Yao.

En esta última parte explicaremos cómo se comporta el intruso, es decir las reglas Dolev-Yao. Éstas indican las acciones que el intruso puede ejecutar y la sintaxis es la siguiente. Cada paso de una acción estará separado por una coma, el orden en que se ejecutan es el mismo en el que aparecen es decir de izquierda a derecha, se distinguirán los mensajes, al igual que en los Strands legítimos, los entrantes precedidos por un símbolo “-”, y los salientes precedidos por un símbolo”+”.

Para entender mejor cómo funcionan las reglas Dolev-Yao utilizaremos de ejemplo el intruso del protocolo NSPK, analizando una a una cada acción que puede realizar.

```
:: nil :: [ nil | -(X), +(pk(Ke,X)), nil ]
```

Ilustración 14: Dolev-Yao, cifrado.

Esta acción es la que permite al intruso recibir un mensaje “X” y cifrarlo utilizando la clave pública de cualquier participante.

```
:: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ]

:: nil :: [ nil | -(X ; Y), +(X), nil ]

:: nil :: [ nil | -(X ; Y), +(Y), nil ]
```

Ilustración 15: Dolev-Yao, Concatenacion.

Estas acciones serán las que permiten al intruso recibir dos mensajes y concatenarlos, o por el contrario recibir un mensaje y separarlo, en muchos ataques esta habilidad es esencial.

```
:: nil :: [ nil | -(X), +(sk(i,X)), nil ]
```

Ilustración 16: Dolev-Yao descifrado

En esta acción se describe la capacidad que posee el intruso para mediante su clave secreta poder descifrar un mensaje recibido en el paso anterior, es importante remarcar que en este caso se utiliza la constante "i", no en forma de variable.

De esta forma el conjunto total de acciones quedará de la siguiente manera:

```
eq STRANDS-DOLEVYAO
= :: nil ::      [ nil | -(X), -(Y), +(X ; Y), nil ] &
:: nil ::        [nil | -(X ; Y), +(X), nil ] &
:: nil ::        [ nil | -(X ; Y), +(Y), nil ] &
:: nil ::        [ nil | -(X), +(sk(i,X)), nil ] &
:: nil ::        [ nil | -(X), +(pk(Ke,X)), nil ]
[nonexec] .
```

Ilustración 17: Dolev-Yao en NSPK

De esta forma las acciones deben estar precedidas del símbolo constante "STRANDS-DOLEVYAO" , las acciones estarán unidas mediante el operador &, que permitirá generar el conjunto de Strands del intruso.

2.2. Análisis de protocolos.

En este apartado estudiaremos tanto los estados normales como los estados de ataque y cómo se especifican estos últimos.

Los estados en Maude-NPA siguen una forma fija única para todos en la que cada estado posee 5 componentes cada uno separado por el símbolo "||", a continuación, describiremos cada uno de ellos.

1º) Conjunto de Strands actuales: Indica el avance de cada Strand en la ejecución actual.

2º) Conocimiento del intruso en el estado actual: Indica en el estado actual el conocimiento que posee el intruso marcado con *inI* y el que no posee, pero aspira a poseer marcado con *!inI*.

3º) Secuencia de mensajes al ejecutar la búsqueda hacia atrás: Comienza con un valor nulo, al estar en un estado de ataque, ya que la búsqueda es hacia atrás a medida que avance hacia el estado inicial.

4º) Información adicional: Este componente tiene una función de depuración al aportar información acerca del espacio de búsqueda en nuestro trabajo no nos aporta ninguna información de interés.

5º) Never Pattern: Este componente nos permite excluir patrones para evitar alcanzar un estado de ataque; en nuestro trabajo no tiene utilidad ya que no pretendemos evitar ningún ataque, para más información consultar [1].

Mediante estos componentes podremos definir un estado de ataque, en este utilizaremos los componentes para poder modelar el ataque que se espera encontrar en el protocolo para cada ataque deberemos especificar un estado de ataque nuevo.

Para poder entender mejor en qué consiste un estado de ataque y cómo se usan los componentes de cada estado para poder modelar el ataque deseado, utilizaremos el estado de ataque que se espera encontrar en el protocolo NSPK.

```
eq ATTACK-STATE(0)

= :: r ::

[ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]

|| n(b,r) inI, empty

|| nil

|| nil

|| nil

[nonexec] .
```

Ilustración 18: Estado de ataque en NSPK.

Podemos observar cómo en el primer componente se especifica el conjunto de Strands que deben darse, y el conocimiento que el intruso debe poseer, en este caso el objetivo es averiguar *n(b,r)* para poder así suplantar la identidad de Alice. También es importante el hecho de que se usan constantes y no variables, esto es así para fijar los participantes en los que se ejecuta la cadena de Strands.

A raíz de este estado se iniciará una búsqueda hacia atrás hasta llegar al ataque inicial, en caso de que fuera un estado inalcanzable no se podría considerar este estado un estado de ataque valido.

2.3. Comandos de Interés

La herramienta nos proporciona una gran cantidad de comandos que nos permiten obtener mucha información a la hora de entender y analizar protocolos, pero en nuestro proyecto nos centraremos en tres que serán los que usaremos a la hora de buscar ataques: “run”, “summary” e “initials”.

Estos comandos siempre estarán precedidos por la palabra red que le indicará a la herramienta que aplica la evaluación en Maude del comando.

El primero “run” le indicará a la herramienta hasta qué profundidad debe expandir el árbol de búsqueda, además se le puede añadir un segundo argumento para encontrar un ataque específico. Por ejemplo, si ejecutásemos “run(7)” nos generaría el árbol buscando ataques hasta una profundidad de 7 niveles, pero si añadimos “run(0,7)” buscaría el ataque 0 hasta una profundidad de 7 niveles.

Por ejemplo, al ejecutar “run(0)” en el protocolo NSPK obtenemos el siguiente resultado, en el que podemos observar los 5 componentes de cada estado.

```
rewrites: 3520 in 47ms cpu (51ms real) (74155 rewrites/second)
result ShortIdSystem: < 1 >
:: r:Fresh ::
[ nil,
-(pk(b, a ; N:Nonce)),
+(pk(a, N:Nonce ; n(b, r:Fresh))) |
-(pk(b, n(b, r:Fresh))), nil]
|
pk(b, n(b, r:Fresh)) inI,
n(b, r:Fresh) inI
|
-(pk(b, n(b, r:Fresh)))
|
nil
```

Ilustración 19: run(0) en NSPK

El segundo “summary” nos permite omitir los estados para obtener únicamente el número actual de estado activos y el número de ataque encontrados.

Podemos observar su funcionamiento ejecutando “summary(6)” y “summary(7)” en el protocolo NSPK y obtenemos la siguiente respuesta:

```
rewrites: 365528 in 457ms cpu (467ms real) (798157 rewrites/second)
result Summary: States>> 2 Solutions>> 0

rewrites: 851688 in 1031ms cpu (1042ms real) (825855 rewrites/second)
result Summary: States>> 4 Solutions>> 1
```

Ilustración 20: summary(6) y summary(7) en NSPK

Por último, el comando “initials”, a diferencia de “run”, nos permite ver únicamente los estados de ataque encontrados en la profundidad especificada; por ejemplo, si usáramos el comando “initials(7)” en el protocolo NSPK obtendríamos la siguiente salida:

```
result IdSystem: < 1 . 5 . 2 . 7 . 1 . 4 . 3 . 1 > (
:: nil :: [nil | -(pk(i, n(b, #1:Fresh))), +(n(b, #1:Fresh)), nil] &
:: nil :: [nil | -(pk(i, a ; n(a, #0:Fresh))), +(a ; n(a, #0:Fresh)), nil] &
:: nil :: [nil | -(n(b, #1:Fresh)), +(pk(b, n(b, #1:Fresh))), nil] &
:: nil :: [nil | -(a ; n(a, #0:Fresh)), +(pk(b, a ; n(a, #0:Fresh))), nil] &
:: #1:Fresh ::
[nil |  -(pk(b, a ; n(a, #0:Fresh))),
        +(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
        -(pk(b, n(b, #1:Fresh))), nil] &
:: #0:Fresh ::
[nil |  +(pk(i, a ; n(a, #0:Fresh))),
        -(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
        +(pk(i, n(b, #1:Fresh))), nil])
        ||
        pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh)) !inI,
        pk(b, n(b, #1:Fresh)) !inI,
        pk(b, a ; n(a, #0:Fresh)) !inI,
        pk(i, n(b, #1:Fresh)) !inI,
        pk(i, a ; n(a, #0:Fresh)) !inI,
        n(b, #1:Fresh) !inI,
        (a ; n(a, #0:Fresh)) !inI
        ||
        +(pk(i, a ; n(a, #0:Fresh))),
        -(pk(i, a ; n(a, #0:Fresh))),
        +(a ; n(a, #0:Fresh)),
        -(a ; n(a, #0:Fresh)),
```

Ilustración 21: initials(7) en NSPK (Parte 1)

```
+(pk(b, a ;n(a, #0:Fresh))),
-(pk(b, a ; n(a, #0:Fresh))),
+(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
-(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
+(pk(i, n(b, #1:Fresh))),
-(pk(i, n(b, #1:Fresh))),
+(n(b, #1:Fresh)),
-(n(b, #1:Fresh)),
+(pk(b, n(b, #1:Fresh))),
-(pk(b, n(b, #1:Fresh)))
||
nil
```

Ilustración 22: initials(7) en NSPK (Parte 2)

Podemos comprobar que este intercambio de mensajes se corresponde con el ataque que buscábamos en el protocolo.

3. Protocolo Needham-Schroeder-Lowe.

En este apartado analizaremos el protocolo Needham-Schroeder-Lowe (NSL), explicaremos el modelo inicial sin propiedades añadidas en el que no hay ningún ataque, a partir de este modelo implementaremos una versión con la propiedad de asociatividad simulada mediante reglas, luego analizaremos una versión inicial usando la asociatividad implementada en la herramienta, sin la necesidad de la regla antes utilizada, y por último explicaremos una versión depurada de la anterior.

3.1. Needham-Schroeder-Lowe inicial.

Este protocolo surge como una mejora del protocolo Needham-Schroeder [5] [6], éste es un protocolo de clave pública en el que intervienen dos participantes con capacidad para una mútua autentificación, el esquema es el siguiente:

A -> B: $\{N_{A,}\ A\}\ P_{kB}$

B -> A: $\{N_a,\ N_b\}\ P_{kA}$

A -> B: $\{N_b\}\ P_{kB}$

Ilustración 23: Esquema del NSPK

La descripción es la siguiente, Alice envía a Bob un Nonce y su identidad encriptadas con la clave pública de Bob, Bob recibe esto y con su clave privada obtiene el Nonce de Alice y su identidad, a continuación, envía a Alice su propio Nonce y el que recibió de ella encriptados con la clave pública de Alice, Alice mediante su clave privada puede obtener el Nonce de Bob y renviárselo a Bob encriptado con su clave pública.

De esta forma Alice y Bob saben que nadie está suplantando la identidad del otro, pero esta primera versión es vulnerable a un ataque *man-in-the-middle*, éste consiste en un intruso que primero se hace pasar por Bob para obtener el mensaje con la identidad y el Nonce de Alice que por error lo encripta con la clave pública del intruso, el intruso utiliza ese mensaje para hacerse pasar por Alice ante Bob, Bob por error enviará al intruso su Nonce junto al de Alice encriptados por la clave pública de A, el intruso envía este mensaje a Alice haciéndose pasar otra vez por Bob, Alice descifra el mensaje y se lo reenvía al intruso encriptado con su clave pública, ahora el intruso conoce el Nonce de Bob y con él completa la suplantación y Bob creerá que el intruso es Alice. El esquema es el siguiente:

A -> I: $\{N_A, A\}\ P_{kI}$

I -> B: $\{N_A, A\}\ P_{kB}$

B -> I: $\{N_a, N_b\}\ P_{kA}$

I -> A: $\{N_a, N_b\}\ P_{kA}$

A -> I: $\{N_b\}\ P_{kI}$

I -> B: $\{N_b\}\ P_{kB}$

Ilustración 24: Ataque al NSPK

Para evitar este ataque Gavin Lowe propone en 1995 una solución modificando el esquema inicial de la siguiente forma [7]:

A -> B: $\{N_A, A\}\ P_{kB}$

B -> A: $\{N_a, N_b, B\}\ P_{kA}$

A -> B: $\{N_b\}\ P_{kB}$

Ilustración 25: Esquema del NSL

Este nuevo esquema introduce la identidad de Bob dentro del mensaje de respuesta del mismo así solo Alice puede descifrarlo y asegurarse de que el emisor es Bob.

Este protocolo se modela en Maude-NPA de la siguiente forma

-Tipos, Subtipos y Operadores:

```
--- Sort Information
sorts Name Nonce Key .
subsort Name Nonce Key < Msg .
subsort Name < Key .
subsort Name < Public .

 --- Encoding operators for public/private encryption
op pk : Key Msg -> Msg [frozen] .
op sk : Key Msg -> Msg [frozen] .

--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .

--- Principals
op a : -> Name . --- Alice
op b : -> Name . --- Bob
op i : -> Name . --- Intruder

--- Concatenation
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
```

Ilustración 26: Tipos, subtipos y operadores NSL

-Las propiedades algebraicas:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .


------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
--- of your protocol
------------------------------------------------------------
var Z : Msg .
var Ke : Key .


*** Encryption/Decryption Cancellation
eq pk(Ke,sk(Ke,Z)) = Z [variant] .
eq sk(Ke,pk(Ke,Z)) = Z [variant] .


endfm
```

Ilustración 27: Cifrado y descifrado en NSL

- Roles participantes:

```
--- Intruder Strand
:: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
:: nil :: [ nil | -(X ; Y), +(X), nil ] &
:: nil :: [ nil | -(X ; Y), +(Y), nil ] &
:: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
:: nil :: [ nil | -(X), +(pk(Ke,X)), nil ] &
:: nil :: [ nil | +(A), nil ]
```

Ilustración 29: Strands Dolev-Yao en NSL

```
--- Alice's Strand
:: r ::
[ nil |   +(pk(B,A ; n(A,r))),
          -(pk(A,n(A,r) ; N ; B)),
          +(pk(B, N)), nil ]
```

Ilustración 30: Strands de Alice en NSL

```
--- Bob's Strand
:: r ::
[ nil |    -(pk(B,A ; N)),
          +(pk(A, N ; n(B,r) ; B)),
          -(pk(B,n(B,r))), nil ]
```

Ilustración 28: Strands de Bob en NSL

-Estado de ataque:

```
--- Attack-State
:: r ::
        [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]
        || n(b,r) inI, empty
        || nil
        || nil
        || nil
```

Ilustración 31: Estado de ataque en NSL

Podemos observar que el protocolo se considerará vulnerable en el momento que el intruso tiene acceso al nonce generado por Bob.

Una vez visto el archivo procederíamos a ejecutar la herramienta con los siguientes comandos:

```
red genGrammars .
red run(0) .
red summary(1) . ---Estados=4 , Ataques=0
red summary(2) . ---Estados=7 , Ataques=0
red summary(3) . ---Estados=6 , Ataques=0
red summary(4) . ---Estados=2 , Ataques=0
red summary(5) . ---Estados=0 , Ataques=0
```

Ilustración 32: Comandos en NSL

Como podemos observar en el árbol de búsqueda llegamos a una profundidad en el árbol en la que ya no quedan más estados que puedan ser expandidos, y tampoco hay ataques descubiertos por lo que podemos llegar a la conclusión de que no hay ataques posibles a este protocolo solo con estas propiedades.

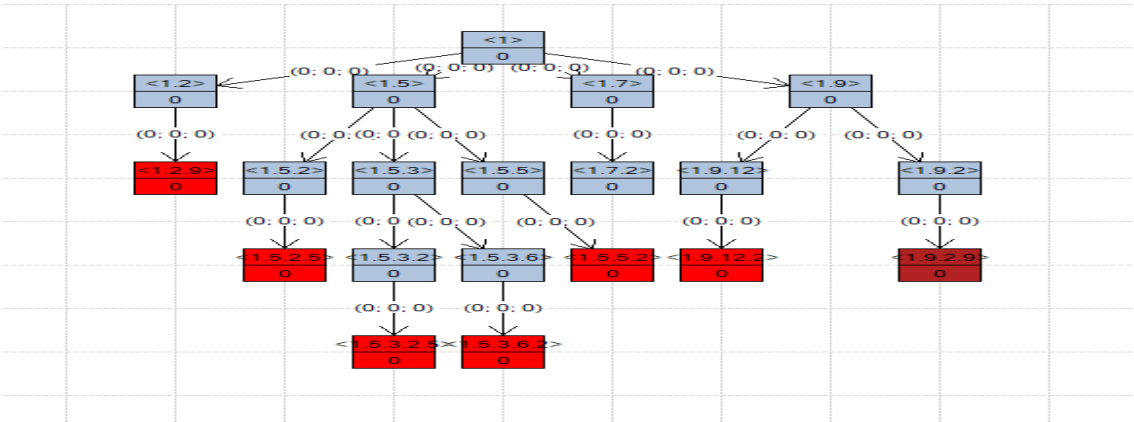

Ilustración 33: Árbol de búsqueda en NSL

3.2. Needham-Schroeder-Lowe con asociatividad mediante regla ecuacional.

En este apartado analizaremos una primera versión del ataque al protocolo Needham-Schroeder-Lowe utilizando para el mismo una regla que permita utilizar la asociatividad el esquema del ataque [8] es el siguiente.

I -> B: { A,I} K_{pkB}
B -> I: {(I, N_B),B} K_{pA}
I -> A: {I,(N_B,B)} K_{pA}
A -> I: {((N_B,B),N_A), K_{pI}
I -> B: {N_B} K_{pB}

Ilustración 34: Esquema de ataque en NSL

A continuación, utilizaremos un diagrama para comprender mejor el intercambio de mensajes en el ataque al protocolo.

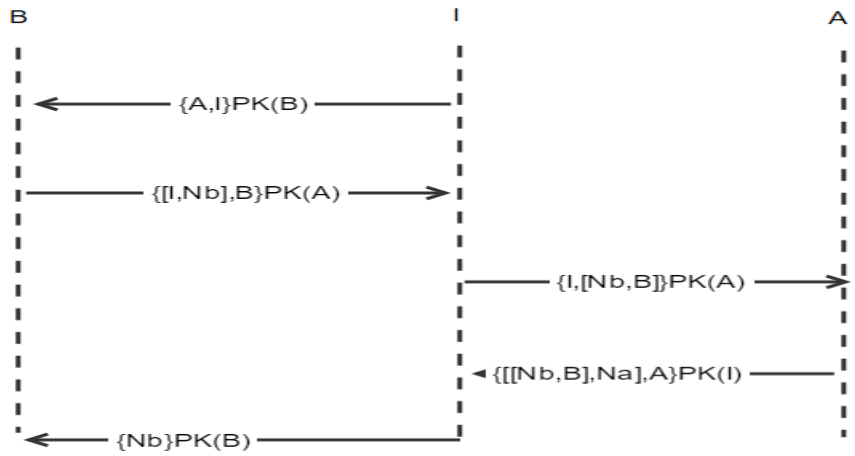

Ilustración 35: Diagrama de ataque en NSL

Al inicio del protocolo el intruso haciéndose pasar por Alice envía a Bob, la identidad de Alice y en lugar del correspondiente Nonce introduce su identidad, todo cifrado con la clave pública de Bob, Bob recibe este mensaje y como no tiene capacidad para poder saber si ha recibido o no un Nonce se limita a enviar el falso Nonce del intruso (su identidad), su propio Nonce y su identidad, los dos Nonce se encuentran en el mismo bloque no unido a la identidad todo cifrado con la clave pública de Alice.

El intruso haciendo uso de la asociatividad separa el bloque de los Nonce y une el Nonce de Bob y la identidad de Bob y envía este mensaje a Alice, ella lo recibe e interpreta que la identidad del emisor es I, por lo que reenvía el mensaje al intruso, pero esta vez cifrado con la clave pública del intruso.

El intruso recibe el mensaje y lo único que debe hacer es usar su clave privada para descifrar el mensaje que contendrá el Nonce de Bob, una vez con él ya es capaz de enviar el mensaje a Bob con su Nonce y cifrado con su clave pública, ahora Bob cree que el intruso tiene la capacidad de descifrar el mensaje.

De esta forma el Intruso ha suplantado a Alice y ha hecho creer a Bob que la comunicación es segura ya que supuestamente ambos participantes se han autentificado mutuamente.

Para modelar este ataque sin utilizar propiedad sin la actualización de la herramienta debemos usar una regla ecuacional que simulará la asociatividad.

```
*** Bounded Associativity (for 3-depth)
eq (Xe ; Ye) ; Ze = Xe ; (Ye ; Ze) [variant] .
```

Ilustración 36: Regla ecuacional de asociatividad en NSL

Esta regla nos permite utilizar la asociatividad, pero con un inconveniente, solo es asociatividad en una dirección, en este caso a izquierda, este inconveniente hace que en el caso de que el orden de los mensajes en el ataque fuera el inverso no sería capaz de encontrar el ataque.

También deberemos modificar el protocolo inicial para que los participantes no tengan la capacidad de poder distinguir si lo que reciben es un Nonce o no.

```
---Modificación en los comportamientos
---Alice
:: r ::
        [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; V ; B)), +(pk(B, V)), nil ]
---Bob
:: r ::
        [ nil | -(pk(B,A ; V)), +(pk(A, V ; n(B,r) ; B)), -(pk(B,n(B,r))), nil ]
---Intruso
:: r ::
         [ nil, -(pk(b,a ; V)), +(pk(a, V ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]

---Modificación en los tipos
sorts Name Nonce Key Data Elm List .
subsort Name Nonce Key Data Elm List < Msg .
subsort Name < Key .
subsort Name < Public .
subsort Name < Data .
subsort Data < List .
```

Ilustración 37: Tipos, subtipos y operadores en NSL con asociatividad

Por último también deberemos modificar el orden de los componentes en el intercambio de los mensajes iniciales en lugar de enviar (Nonce; Identidad), envía (Identidad; Nonce).

Una vez realizados los cambios podemos proceder a la ejecución siguiendo el mismo procedimiento que en el apartado anterior, aumentando la profundidad hasta que no hallan más estados que puedan ser expandidos.

```
red genGrammars .
red run(0) .
red summary(1) . ---Estados=1 , Ataques=0
red summary(2) . ---Estados=2 , Ataques=0
red summary(3) . ---Estados=4, Ataques=0
red summary(4) . ---Estados=3 , Ataques=0
red summary(5) . ---Estados=2 , Ataques=0
red summary(6) . ---Estados=1 , Ataques=1
```

Ilustración 38: Comandos de ataque a NSL

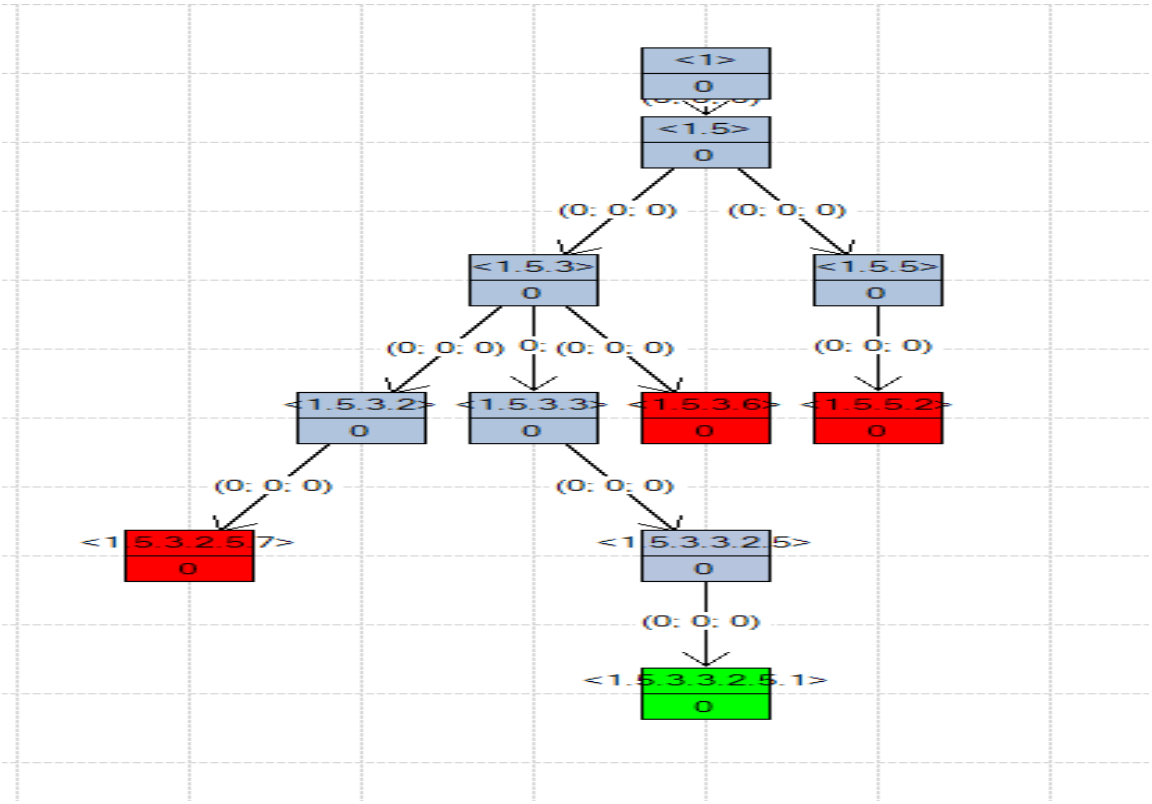

Ilustración 39: Árbol de búsqueda de ataque en NSL

Podemos observar que efectivamente encontramos un ataque en la profundidad 6, podemos también asegurar que es el único ya que el resto de estado son finales. Una vez visto esto procederemos a estudiar el ataque mediante la salida de la herramienta.

```
Maude> red initials(6) .
reduce in MAUDE-NPA : initials(6) .
rewrites: 387 in 0ms cpu (0ms real) (~ rewrites/second)
result ShortIdSystem: < 1 . 5 . 3 . 3 . 2 . 5 . 1 > (
:: nil ::
[ nil |
   -(pk(i, (n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a)),
   +((n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh)),
   +(pk(b, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh) ; b),
   +(n(b, #0:Fresh)), nil]  &
:: nil ::
[ nil |
   -((n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a),
   +(n(b, #0:Fresh) ; b), nil]  &
:: #0:Fresh ::
[ nil |
   -(pk(b, a ; i)),
   +(pk(a, i ; n(b, #0:Fresh) ; b)),
   -(pk(b, n(b, #0:Fresh))), nil]  &
:: #1:Fresh ::
[ nil |
   -(pk(a, i ; n(b, #0:Fresh) ; b)),
   +(pk(i, (n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a)), nil] )
|
pk(a, i ; n(b, #0:Fresh) ; b) !inI,
pk(b, n(b, #0:Fresh)) !inI,
pk(b, a ; i) !inI,
pk(i, (n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a) !inI,
n(b, #0:Fresh) !inI,
(n(b, #0:Fresh) ; b) !inI,
((n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a) !inI
|
generatedByIntruder(pk(b, a ; i)),
-(pk(b, a ; i)),
+(pk(a, i ; n(b, #0:Fresh) ; b)),
-(pk(a, i ; n(b, #0:Fresh) ; b)),
+(pk(i, (n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a)),
-(pk(i, (n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a)),
+((n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a),
-((n(b, #0:Fresh) ; b) ; n(a, #1:Fresh) ; a),
+(n(b, #0:Fresh) ; b),
-(n(b, #0:Fresh) ; b),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
|
nil
```

Ilustración 40: Salida de la herramienta en el ataque a NSL

Podemos observar que efectivamente el ataque que ha encontrado la herramienta es el descrito en [8], el que queríamos encontrar por lo tanto podemos concluir que mediante la regla anteriormente descrita la herramienta tiene la capacidad de encontrar el ataque y de generar el árbol de búsqueda correspondiente.

Asimismo podemos concluir que el árbol de búsqueda es finito por lo que no son posibles otros ataques utilizando esta propiedad. En las siguientes secciones intentaremos recrear este resultado utilizando únicamente la capacidad de la herramienta para utilizar la asociatividad.

3.3. Needham-Schroeder-Lowe con asociatividad mediante axioma ecuacional.

En este apartado utilizaremos como punto de partida el archivo visto en el primer apartado, y añadiremos las modificaciones necesarias para utilizar la asociatividad y así poder detectar el ataque.

Empezaremos modificando el operador de concatenación, para poder utilizar la asociatividad como se muestra a continuación.

```
--- Concatenation

op _;_ : Msg  Msg  -> Msg [gather (e E) frozen assoc] .
```

Ilustración 41: Operador concatenación con asociatividad

El marcador "assoc" indicará a la herramienta que el operador concatenación puede utilizar la asociatividad, de esta forma no será necesario crear reglas específicas para ello.

También se modificarán las variables y los comportamientos de los participantes de forma parecida a como hemos hecho en la sección anterior

```
---Modificación en los comportamientos
---Alice
:: r ::
        [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; M ; B)), +(pk(B, M)), nil ]
---Bob
:: r ::
        [ nil | -(pk(B,A ; M)), +(pk(A, M ; n(B,r) ; B)), -(pk(B,n(B,r))), nil ]
---Intruso
:: r ::
         [ nil, -(pk(b,a ; M)), +(pk(a, M ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]

---Modificación en las variables
var Ke : Key .
vars X Y Z M : Msg .
vars r r' : Fresh .
vars A B : Name .
vars N N1 N2 : Nonce .
```

Ilustración 42 : Modificaciones en NSL para el axioma ecuacional

De esta forma los participantes no podrán saber si reciben un Nonce o un elemento Msg, gracias a esto el intruso podrá introducir su identidad en el mensaje sin que ni Alice ni Bob lo detecten.

Ejecutando este protocolo obtenemos los siguientes resultados:

```
red genGrammars .
red run(0) .
red summary(1) .     ---Estados=4      Ataques=0
red summary(2) .     ---Estados=7      Ataques=0
red summary(3) .     ---Estados=9      Ataques=0
red summary(4) .      ---Estados=11    Ataques=0
red summary(5) .     ---Estados=19     Ataques=1
red summary(6) .     ---Estados=34     Ataques=3
red summary(7) .     ---Estados=60     Ataques=5
red summary(8) .     ---Estados=107    Ataques=7
red summary(9) .     ---Estados=202    Ataques=9
red summary(10) .    ---Estados=399    Ataques=11
red summary(11) .    ---Estados=796    Ataques=13
```

Ilustración 43: Comando de ataque a NSL usando Axioma ecuacional

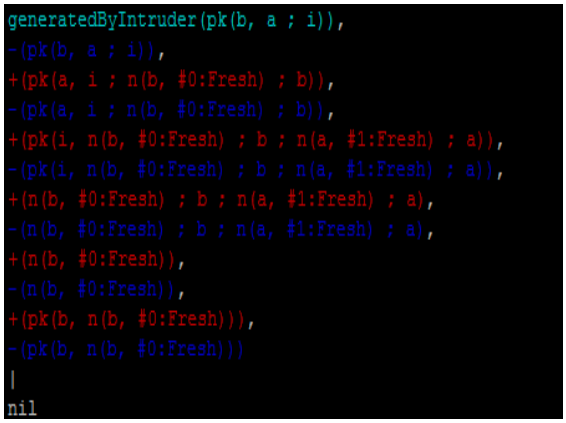

Ilustración 44: Salida de la herramienta 1º ataque

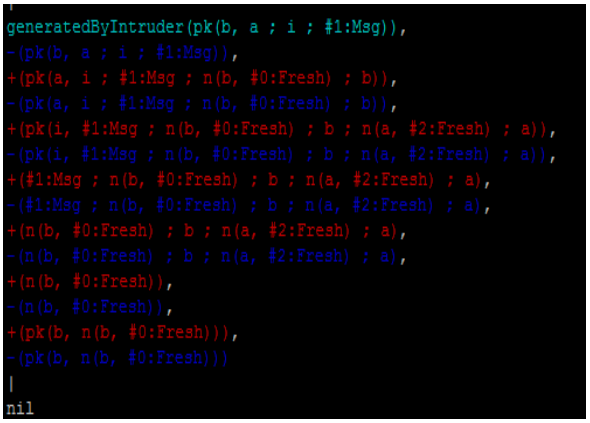

Ilustración 45: Salida de la herramienta 2º ataque

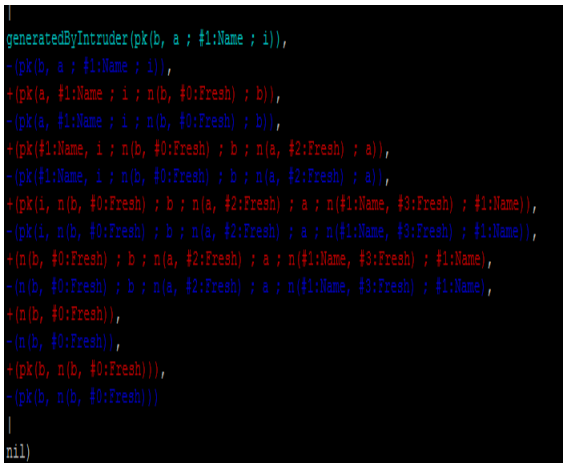

Ilustración 46: Salida de la herramienta 3º ataque

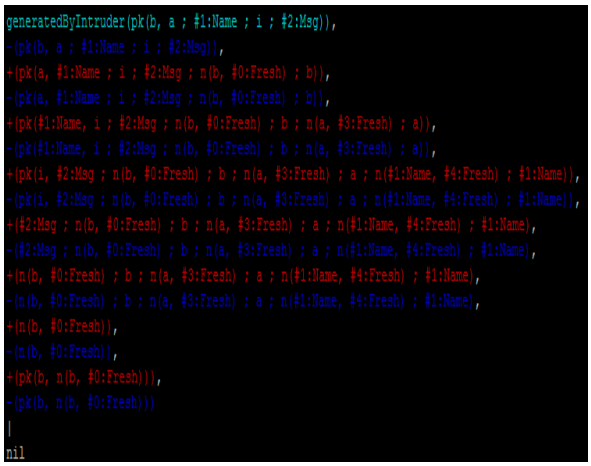

Ilustración 47: Salida de la herramienta 4º ataque

Podemos observar que se generan tanto estados como ataques sin fin. Esto es debido a que al no haber limitado la capacidad del intruso para extender el mensaje introduciendo elementos. Esto puede tener una utilidad, a la hora de esconder información en los mensajes pero en nuestro caso no nos interesa para ello en la sección siguiente modificaremos el protocolo para únicamente aceptar un tipo de dato finito, así de esta forma limitar la capacidad para generar estados de forma infinita.

3.4. Needham-Schroeder-Lowe con límites y asociatividad mediante axioma ecuacional.

En este apartado añadiremos las modificaciones necesarias para poder limitar la expansión de estados en el árbol de búsqueda y así poder asegurar que solo hay un ataque, al conseguir un árbol de búsqueda finito para ello tendremos que crear unos nuevos tipos de datos para el intercambio de mensajes manteniendo la incapacidad para saber si se ha recibido un Nonce, pero permitiendo únicamente enviar como alternativa una identidad así de esta forma el protocolo se modificara de la siguiente forma:

```
---Modificación en los comportamientos
---Alice
:: r ::
        [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; V ; B)), +(pk(B, V)), nil ]
---Bob
:: r ::
        [ nil | -(pk(B,A ; V)), +(pk(A, V ; n(B,r) ; B)), -(pk(B,n(B,r))), nil ]
---Intruso
:: r ::
         [ nil, -(pk(b,a ; V)), +(pk(a, V ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]

---Modificación en los tipos
sorts Name Nonce Key Data .
subsort Name Nonce Key Data < Msg .
subsort Name < Key .
subsort Name < Public .
subsort Name < Data .

---Modificación en los operadores
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen assoc] .

op _;_ : Nonce Name  -> Data [gather (e E) frozen assoc] .

---Modificación en las variables
var V : Data .
```

Ilustración 48: Modificaciones en NSL con limites

Con el nuevo tipo de dato "Data" se limita la búsqueda ya que al sustituir el tipo "Msg" por éste en el intercambio de mensajes se obliga a que se envíe un Nonce o una Identidad obligatoriamente.

Una vez hecha la modificación ejecutamos la herramienta de igual forma similar que en apartados anteriores. Obteniendo los siguientes resultados:

```
red genGrammars .
red run(0) .
red summary(1) . ---Estados=1 , Ataques=0
red summary(2) . ---Estados=2 , Ataques=0
red summary(3) . ---Estados=3, Ataques=0
red summary(4) . ---Estados=3 , Ataques=0
red summary(5) . ---Estados=1 , Ataques=1
```

Ilustración 49: Comandos de ataque en NSL con limites

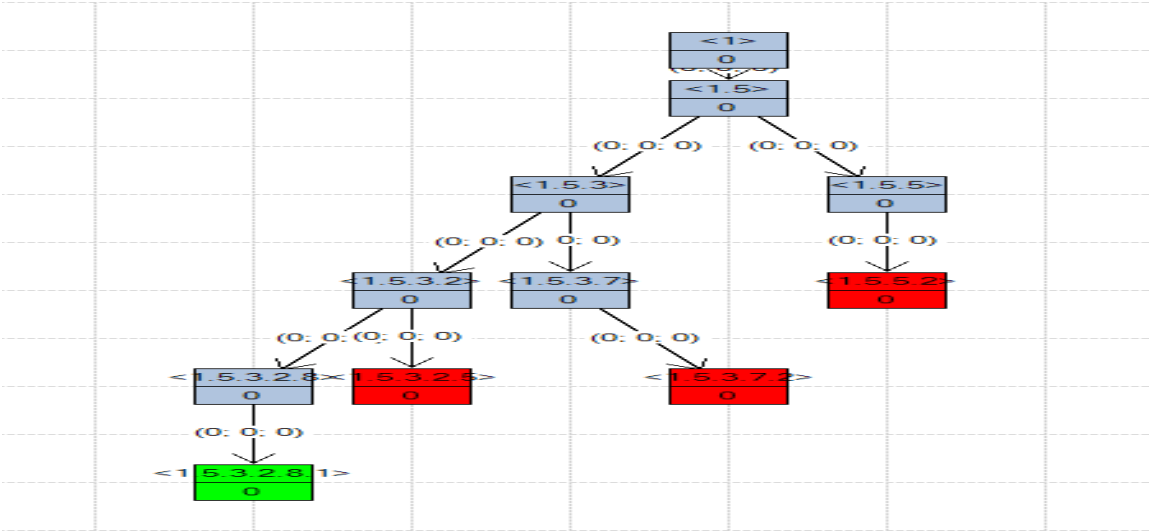

Ilustración 50: Árbol de búsqueda en NSL con limites

Podemos observar que se encuentra una solución en profundidad 5, un nivel menos que en la versión con asociatividad simulada, además en este caso se han explorado 11 estados y en la versión anterior no limitada 19 por lo que hemos conseguido reducir el número y por último se ha conseguido que todos los estados sean finales por lo que tenemos una solución y única.

La siguiente tabla muestra las comparativas entre versiones:

Versión	Estados	Profundidad	Tiempo	Rescrituras /seg
Regla ecuacional	11	6	5482	5625299
Axioma ecuacional	11	5	5595	5735618

Por ultimo observando la solución que ha encontrado la herramienta:

```
Maude> red initials(5) .
reduce in MAUDE-NPA : initials(5) .
rewrites: 5735612 in 5500ms cpu (5499ms real) (1042773 rewrites/second)
result ShortIdSystem: < 1 . 5 . 3 . 2 . 8 . 1 > (
:: nil ::
[ nil |
    -(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)),
    +(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a), nil]  &
:: nil ::
[ nil |
    -(n(b, #0:Fresh)),
    +(pk(b, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
    -(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a),
    +(n(b, #0:Fresh)), nil]  &
:: #0:Fresh ::
[ nil |
    -(pk(b, a ; i)),
    +(pk(a, i ; n(b, #0:Fresh) ; b)),
    -(pk(b, n(b, #0:Fresh))), nil]  &
:: #1:Fresh ::
[ nil |
    -(pk(a, i ; n(b, #0:Fresh) ; b)),
    +(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)), nil] )
|
pk(a, i ; n(b, #0:Fresh) ; b) !inI,
pk(b, n(b, #0:Fresh)) !inI,
pk(b, a ; i) !inI,
pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a) !inI,
n(b, #0:Fresh) !inI,
(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a) !inI
|
generatedByIntruder(pk(b, a ; i)),
-(pk(b, a ; i)),
+(pk(a, i ; n(b, #0:Fresh) ; b)),
-(pk(a, i ; n(b, #0:Fresh) ; b)),
+(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)),
-(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)),
+(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a),
-(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
|
nil
```

Ilustración 51: Salida de la herramienta en NSL con limites

Podemos ver que es exactamente la misma solución que encontramos con la solución utilizando la asociatividad simulada mediante reglas ecuacionales por lo tanto queda comprobado que la asociatividad incorporada en la nueva versión de la herramienta que implementa la asociatividad como axioma ecuacional funciona correctamente y es capaz de ser aplicada para verificar protocolos que tengan ataques que exploten dicha propiedad.

4. Protocolo Secret2016

En este apartado estudiaremos el protocolo Secret2016 [9] en su versión con asociatividad simulada mediante las reglas ecuacionales y con la asociatividad utilizando el axioma ecuacional para ello primero explicaremos el protocolo de forma esquemática, así como el ataque para luego en las secciones siguientes modelar este ataque mediante la herramienta Maude-NPA.

Este es un protocolo que explota un ataque por confusión [10] es decir, su objetivo es hacer que un participante de la comunicación piense que ha recibido un tipo de dato cuando en verdad está recibiendo otro, en este caso

El esquema será el siguiente:

S -> A: N_S

A -> B: $\{N_S, S\}\ S_{kA}$

A -> B: $\{B,(N_A,S)\}\ S_{kA}$

Ilustración 52: Esquema de Secret2016

En un principio el servidor, envía a Alice su Nonce, Alice con el Nonce enviado por el servidor, envía a Bob el Nonce del servidor y la identidad del servidor cifrado con su clave secreta, luego Alice vuelve a enviar a Bob otro mensaje este conteniendo la identidad de B, su propio Nonce y la identidad del servidor cifrado con su clave privada.

De esta forma Bob puede comprobar que Alice es quien dice ser, al utilizar la clave pública de Alice para descifrar los mensajes recibidos, puede comprobar que la identidad del servidor es la misma en ambos, así como obtener el Nonce de Alice y el del servidor.

En este protocolo podemos encontrar dos estados de ataque, es decir dos situaciones en que la seguridad se ve comprometida, el principal factor que influye en el ataque es la incapacidad de reconocer los tipos del mensaje, una vez visto esto pasaremos a describir los dos ataques.

1º Ataque: El intruso generara un Nonce y se lo enviara a Alice, Alice no tiene capacidad para poder saber quién envía el Nonce ya que cada Nonce solo es conocido por su creador, Alice enviará dos mensajes a Bob, ambos cifrados con su clave privada, el problema surge porque Bob no es capaz de distinguir el tipo de dato del mensaje, en el segundo mensaje el Nonce de Alice y la identidad del servidor deben ir juntos y precedidos por la identidad de Bob {b,(Ns,S)}, pero como asumimos la asociatividad Bob puede malinterpretar el segundo mensaje entendiendo (b,Ns) como si fuera Na por lo que puede interpretarlo como el primer mensaje, esto generara una confusión. El ataque pues tendrá la siguiente forma:

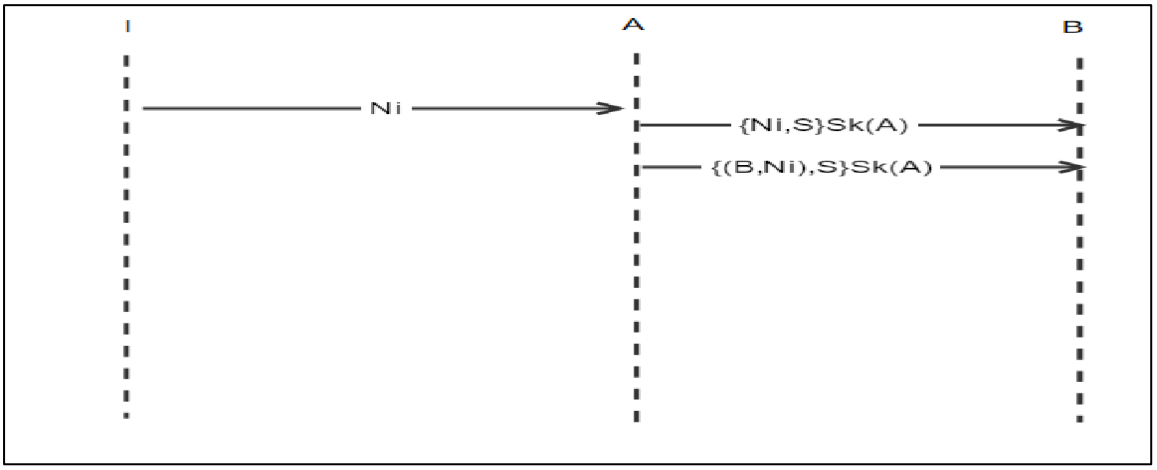

Ilustración 53: 1º Ataque al Protocolo Secret2016 (Diagrama)

I -> A: N_I

A -> B: {N_I, S} S_{kA}

A -> B: {(B, N_A),S} S_{kA}

Ilustración 54: 1º Ataque al Protocolo Secret2016

2º Ataque: Este ataque se produce cuando se generan dos instancias a la vez, primero el servidor generara un Nonce (Ni1) y Alice una vez recibido el Nonce envía el siguiente mensaje a Bob {Ni1,S}, al mismo tiempo el intruso generara otro Nonce (Ni2) y se lo enviara a Alice, Alice repetirá el procedimiento, como Bob únicamente espera un mensaje de la forma {Msg,S} no es capaz de distinguir cual es el mensaje que corresponde a la pareja del segundo mensaje, por lo que entra un estado de confusión, este ataque tendrá la siguiente forma:

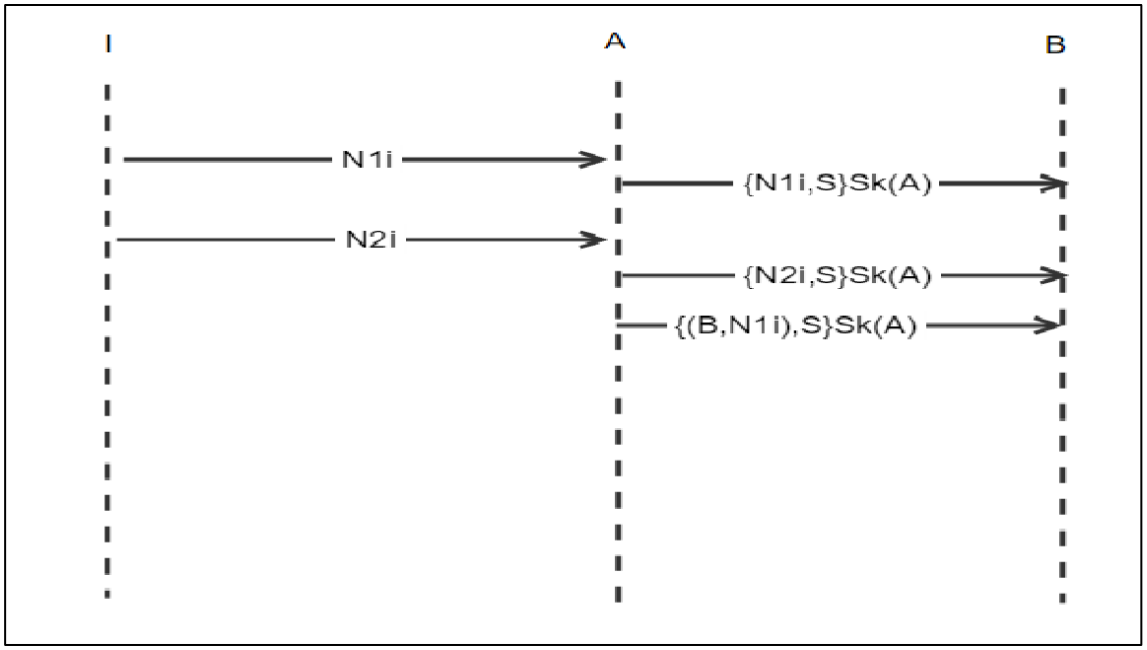

Ilustración 55: 2º Ataque al Protocolo Secret2016 (Diagrama).

I -> A: $N1_I$

I -> A: $N2_I$

A -> B: $\{N1_I, S\}\ S_{kA}$

A -> B: $\{N2_I, S\}\ S_{kA}$

A -> B: $\{(B, N_A),S\}\ S_{kA}$

Ilustración 56: 2º Ataque al Protocolo Secret2016

Una vez vistos los ataques podemos pasar a estudiar los dos modelos del protocolo el primero con la asociatividad usando reglar ecuacionales y el segundo aprovechando la nueva capacidad de Maude 2.7.1 usando axiomas ecuacionales.

Así podremos comparar los resultados obtenidos comparando el tiempo de búsqueda, la cantidad de rescrituras y por último sus árboles de búsqueda.

4.1. Secret2016 con asociatividad mediante regla ecuacional

Primero estudiaremos cómo se comportan los participantes legítimos de la comunicación, éstos son Alice, Bob y el servidor.

```
--- servidor
:: r :: [nil |      +(n(s,r)), nil ]

--- Alice
:: r :: [nil |      -(N),
                    +(sk(a,N ; s)),
                    +(sk(a,b ; (n(a,r) ; s))), nil]

--- Bob
:: nil :: [nil |    -(sk(a,X ; s)),
                    -(sk(a,(b ; Z) ; s)), nil]
```

Ilustración 57: Strands legítimos en Secret2016

- El servidor enviará un mensaje con su Nonce que es el resultado de aplicar el operador "n" a un elemento de tipo "fresh" y su identidad.

- Alice recibirá un elemento de tipo Nonce, y enviará dos mensajes, ambos cifrados con su clave privada, primero el elemento Nonce que recibió junto con la identidad de servidor y en el segundo mensaje la identidad de Bob junto con su propio Nonce y la identidad del servidor, estos dos últimos unidos.

- Bob recibirá dos mensajes el primero consistirá en un elemento de tipo "Msg" concatenado con la identidad del servidor, el segundo mensaje será su propia identidad un elemento de tipo "Msg" y la identidad del servidor.

A continuación, veremos los Strands Dolev-Yao, es decir las acciones que podrá realizar el intruso.

```
---Operadores de concatenacion
        :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ]
        :: nil :: [ nil | -(X ; Y), +(X), nil ]
        :: nil :: [ nil | -(X ; Y), +(Y), nil ]

--- Cifrado con su clave privada
        :: nil :: [ nil | -(X), +(sk(i,X)), nil ]

--- Cifrado con cualquier clave publica
        :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ]
```

Ilustración 58: Strands Dolev-Yao en Secret2016

La asociatividad se implementará en este modelo mediante una regla ecuacional que será la siguiente.

```
---Regla ecuacional.

eq Xe ; (Ye ; Ze) = (Xe ; Ye) ; Ze [variant] .
```

Ilustración 59: Regla ecuacional de asociatividad en Secret2016

Esta regla define una asociatividad con dirección a izquierda y con una profundidad de tres elementos.

Por último, el estado de ataque será el siguiente.

```
:: nil ::
        [nil, -(sk(a,X ; s)), -(sk(a,(b ; N) ; s)) | nil]
        || empty
        || nil
        || nil
        || nil
```

Ilustración 60: Estado de ataque en Secret2016

Podemos ver como se considerará un estado en el que la seguridad se ve comprometida, aquellos en los que se reciben dos mensajes, que no pueden diferenciarse, es decir un estado de confusión, podemos ver como "X" al ser de tipo "Msg" no puede diferenciarse de "(b;N)" al finalizar ambos mensajes con "s", respecto al resto de componentes del estado podemos ver que no es necesario que el intruso tenga conocimiento.

Una vez analizados los componentes del protocolo modelado pasaremos a su ejecución en la cual usaremos los siguientes comandos.

```
red genGrammars .
red run(0) .
red summary(1) . ---  Estados=2     Ataques=0
red summary(2) . ---  Estados=3     Ataques=1
red summary(3) . ---  Estados=2     Ataques=2
```

Ilustración 61: Comandos de ataque en Secret2016

Podemos observar como la herramienta efectivamente encuentra los ataques esperados, que habíamos desarrollado de forma teórica, así podemos afirmar que la herramienta es capaz mediante reglas ecuacionales de encontrar los ataques objetivo.

El árbol de búsqueda resultado de la exploración del protocolo será el siguiente.

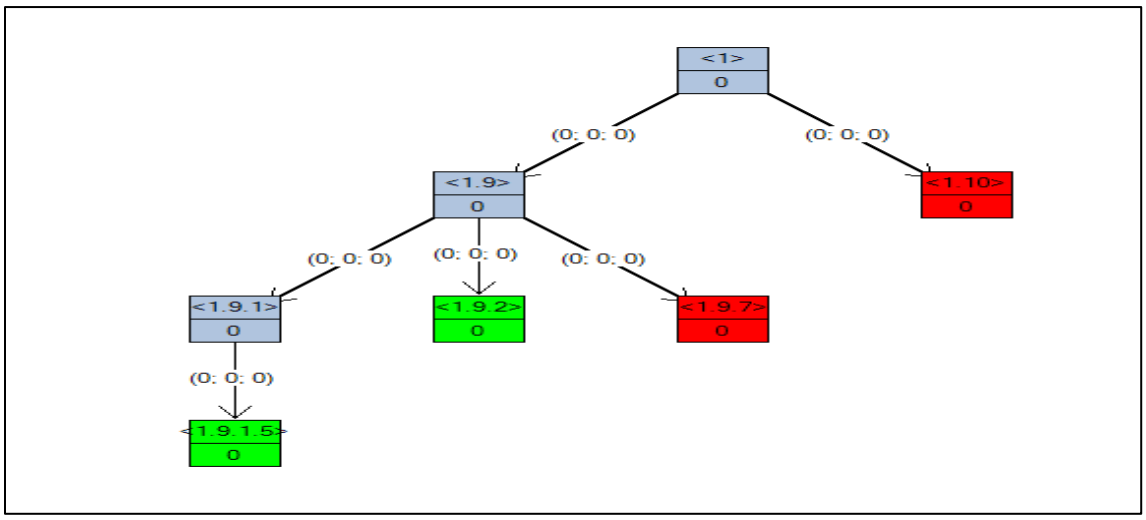

Ilustración 62: Arbol de búsqueda en Secret2016

Podemos observar los dos estados en los que se a un ataque, así como los estados que no continúan.

Por último, la salida que nos ofrece la herramienta de cada uno de los ataques es la siguiente.

```
< 1 . 9 . 1 . 5 > (
:: nil ::
[ nil |
   -(sk(a, #1:Nonce ; s)),
   -(sk(a, (b ; n(a, #2:Fresh)) ; s)), nil]  &
:: #0:Fresh ::
[ nil |
   -(#1:Nonce),
   +(sk(a, #1:Nonce ; s)), nil]  &
:: #2:Fresh ::
[ nil |
   -(#3:Nonce),
   +(sk(a, #3:Nonce ; s)),
   +(sk(a, (b ; n(a, #2:Fresh)) ; s)), nil] )
|
#3:Nonce !inI,
#1:Nonce !inI,
sk(a, #1:Nonce ; s) !inI,
sk(a, (b ; n(a, #2:Fresh)) ; s) !inI,
inst(#3:Nonce)
|
generatedByIntruder(#1:Nonce),
-(#1:Nonce),
+(sk(a, #1:Nonce ; s)),
generatedByIntruder(#3:Nonce),
-(#3:Nonce),
+(sk(a, #3:Nonce ; s)),
+(sk(a, (b ; n(a, #2:Fresh)) ; s)),
-(sk(a, #1:Nonce ; s)),
-(sk(a, (b ; n(a, #2:Fresh)) ; s))
|
nil
```

Ilustración 63: Respuesta de la herramienta al 1º ataque Secret2016

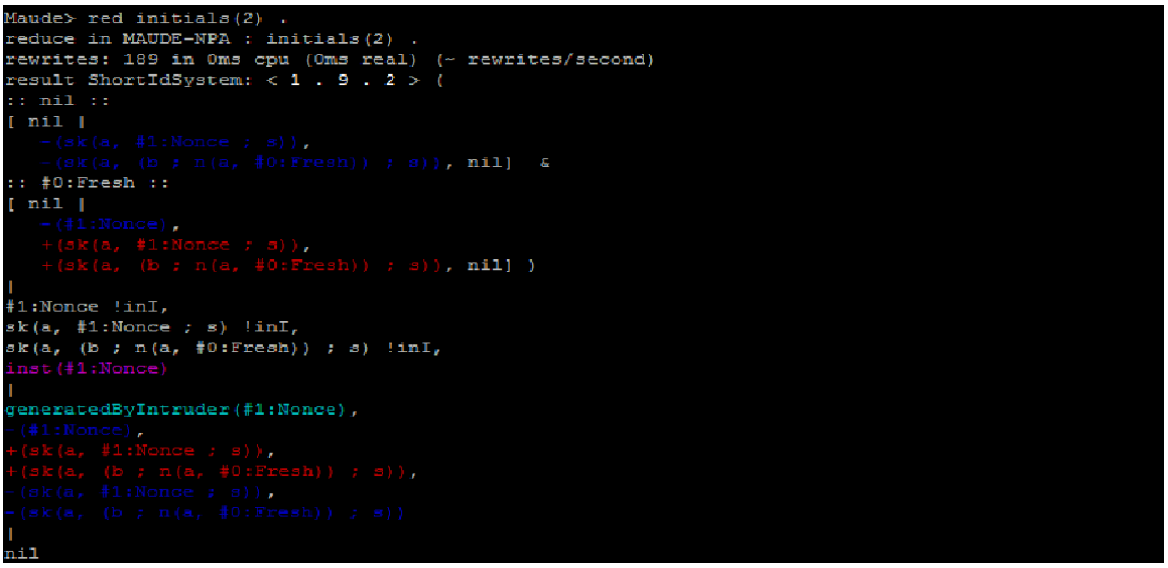

Ilustración 64: Respuesta de la herramienta al 2º ataque Secret2016

4.2. Secret2016 con asociatividad mediante axioma ecuacional

Por último, en esta sección realizaremos el modelado del protocolo utilizando la asociatividad como axioma ecuacional. Para ello deberemos realizar modificaciones en el código que pasaremos a redactar a continuación.

Primero deberemos eliminar ciertos tipos de datos no necesarios, al no necesitarse el uso de listas podremos prescindir de los tipos "Elm" y "List", quedando de la siguiente forma.

```
--- Tipos:
        sorts Name Nonce Key .
        subsort Name < Key .
        subsort Name Nonce Key < Msg .
        subsort Name < Public .
```

Ilustración 65: Tipos y subtipos modificados en SeCret06

La siguiente modificación será añadir al operador de concatenación la capacidad de utilizar la asociatividad, característica de la versión 2.7.1.

```
--- Operador concatenacion
        op _;_ : Msg  Msg  -> Msg [assoc frozen] .
```

Ilustración 66: Operados concatenación con asociatividad (Secret2016)

Por ultimo deberemos retirar la regla que permitía al modelo anterior utilizar la asociatividad.

```
*** Bounded Associativity (for 3-depth)
        eq Xe ; (Ye ; Ze) = (Xe ; Ye) ; Ze [variant] .
```

Ilustración 67: Regla ecuacional de asociatividad en Secret2016

Una vez hechas las modificaciones podemos realizar la ejecución del protocolo para poder evaluar los resultados, para ello utilizaremos los mismos comandos que en la versión anterior.

```
red genGrammars .
red run(0) .
red summary(1) . ---   Estados=2      Ataques=0
red summary(2) . ---   Estados=3      Ataques=1
red summary(3) . ---   Estados=2      Ataques=2
```

Ilustración 68: Comandos de ataque en Secret2016 con axioma ecuacional

El árbol de búsqueda será el siguiente.

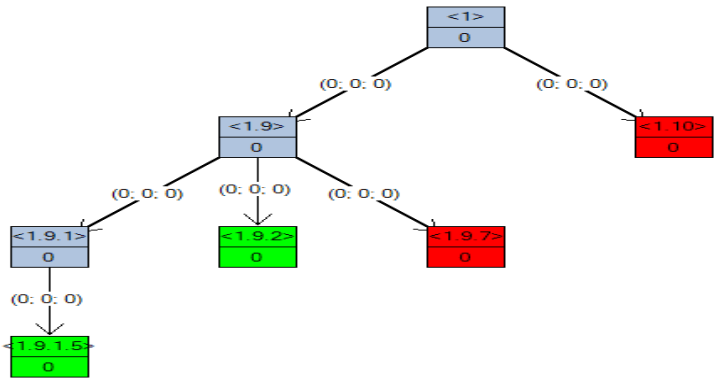

Ilustración 69: Arbol de búsqueda en Secret2016 con axioma ecuacional

Podemos observar que el árbol de búsqueda, el número y los estados son los mismos que en el modelo anterior, por lo que podemos confirmar que la herramienta funciona correctamente utilizando axiomas ecuacionales, para poder comparar correctamente los dos modelos, compararemos la salida que nos da la herramienta a los dos ataques encontrados.

```
< 1 . 9 . 1 . 5 > (
:: nil ::
[ nil |
   -(sk(a, #1:Nonce ; s)),
   -(sk(a, (b ; n(a, #2:Fresh)) ; s)), nil]  &
:: #0:Fresh ::
[ nil |
   -(#1:Nonce),
   +(sk(a, #1:Nonce ; s)), nil]  &
:: #2:Fresh ::
[ nil |
   -(#3:Nonce),
   +(sk(a, #3:Nonce ; s)),
   +(sk(a, (b ; n(a, #2:Fresh)) ; s)), nil] )
|
#3:Nonce !inI,
#1:Nonce !inI,
sk(a, #1:Nonce ; s) !inI,
sk(a, (b ; n(a, #2:Fresh)) ; s) !inI,
inst(#3:Nonce)
|
generatedByIntruder(#1:Nonce),
-(#1:Nonce),
+(sk(a, #1:Nonce ; s)),
generatedByIntruder(#3:Nonce),
-(#3:Nonce),
+(sk(a, #3:Nonce ; s)),
+(sk(a, (b ; n(a, #2:Fresh)) ; s)),
-(sk(a, #1:Nonce ; s)),
-(sk(a, (b ; n(a, #2:Fresh)) ; s))
|
nil
```

Ilustración 71: Respuesta de la herramienta al 1º ataque Secret2016 con axioma ecuacional

```
Maude> red initials(2) .
reduce in MAUDE-NPA : initials(2) .
rewrites: 189 in 0ms cpu (0ms real) (~ rewrites/second)
result ShortIdSystem: < 1 . 9 . 2 > (
:: nil ::
[ nil |
   -(sk(a, #1:Nonce ; s)),
   -(sk(a, (b ; n(a, #0:Fresh)) ; s)), nil]  &
:: #0:Fresh ::
[ nil |
   -(#1:Nonce),
   +(sk(a, #1:Nonce ; s)),
   +(sk(a, (b ; n(a, #0:Fresh)) ; s)), nil] )
|
#1:Nonce !inI,
sk(a, #1:Nonce ; s) !inI,
sk(a, (b ; n(a, #0:Fresh)) ; s) !inI,
inst(#1:Nonce)
|
generatedByIntruder(#1:Nonce),
-(#1:Nonce),
+(sk(a, #1:Nonce ; s)),
+(sk(a, (b ; n(a, #0:Fresh)) ; s)),
-(sk(a, #1:Nonce ; s)),
-(sk(a, (b ; n(a, #0:Fresh)) ; s))
|
nil
```

Ilustración 70: Respuesta de la herramienta al 2º ataque Secret2016 con axioma ecuacional

Podemos comprobar que la herramienta nos devuelve exactamente la misma salida que en el primer modelo, esto confirma nuestra teoría inicial. Hemos comprobado que el funcionamiento con ambos modelos es equivalente, aunque el método de las reglas ecuaciones tiene limitaciones que no poseen los axiomas ecuacionales.

Por ultimo para añadir más elementos a la hora de comparar estos modelos, estudiaremos los tiempos y reescrituras con cada uno de los métodos. La tabla resultado será la siguiente.

Versión	Estados	Profundidad	Tiempo	Rescrituras /seg	Relación tiempo	Relación escrituras
Regla ecuacional	7	4	1612	1377591	1	1,008
Axioma ecuacional	7	4	1588	1389209	1,015	1

A la vista de los resultados podemos concluir que las mejoras en cuanto a tiempo y rescrituras no es relevante en este protocolo por lo que para poder decantarnos por un método u otro sólo deberemos guiarnos por el que nos ofrezca mayores ventajas, y en este caso es utilizar la asociatividad como axioma ecuacional ya que no tendremos la limitación de profundidad ni direccionalidad que tiene la asociatividad como regla ecuacional.

5. Conclusiones y trabajo futuro

En este último apartado expondremos las conclusiones, obtenidas una vez obtenidos los resultados de la experimentación con los dos protocolos tratados, Secret2016 y NSL. Con estos datos podremos llegar a la conclusión final y proponer trabajos futuros [11] en los que se pueda utilizar la propiedad ecuacional en este trabajo tratada, así como parte del conocimiento adquirido en el mismo, estos trabajos podrán ser resueltos de manera similar a los vistos en el proyecto.

5.1. Conclusiones

Durante el proyecto se han conseguido modelar los protocolos propuestos descritos de forma esquemática en la documentación, tanto Secret2016 [9], como NSL [8], a raíz de esto se han podido modelar los ataques que habíamos planteado como objetivo, los mismos explotan la propiedad asociatividad, la cual es la piedra angular de nuestro proyecto.

Mediante el modelado de estos protocolos en los que se incluye el ataque deseado hemos podido comprobar el funcionamiento de la herramienta en un primer momento sin utilizar ni axiomas ecuacionales, ni reglas ecuacionales. Como era de esperar la herramienta ha interpretado estos protocolos como seguros al no utilizarse la propiedad que facilita el ataque.

Continuando con el modelado se abordaron los protocolos propuestos pero utilizando reglas ecuacionales, para así obtener un primer acercamiento, y en adición comprobar que el ataque propuesto en la documentación es posible, y lo más importante, es detectado por Maude-NPA, este primer acercamiento utiliza un método presente ya utilizado en otras ocasiones, que consiste en utilizar reglas ecuacionales para substituir axiomas ecuacionales, pero esto presenta ciertos inconvenientes que hemos podido detectar durante la elaboración del proyecto, los cuales abordaremos más adelante.

La última parte de la experimentación consistía en modelar los protocolos propuestos, pero explotando la nueva capacidad de la herramienta Maude-NPA, utilizando la asociatividad como axioma ecuacional.

Una vez modelados, los protocolos Secret2016 y NSL utilizando axiomas ecuacionales, podemos evaluar los datos y confirmar, primero que el número de ataque que encuentra la herramienta es el mismo en los dos métodos, segundo que los mensajes generados por el intruso son los mismos con los dos métodos, gracias a esta información somos capaces de resolver las cuestiones planteadas en este proyecto.

Podemos confirmar que los dos métodos nos proporcionaran los mismos resultados, por lo tanto, podemos confirmar que la nueva actualización 2.7.1 de Maude, utilizada por Maude-NPA, nos ofrece un método alternativo para modelar protocolos eficiente y seguro.

Por otra parte, durante el desarrollo hemos encontrado inconvenientes en la utilización de reglas ecuacionales para encontrar ataque que exploten la asociatividad, las reglas ecuacionales solo nos permiten definir la asociatividad en un sentido, y además tendremos que especificar la profundidad, esto es un gran inconveniente ya que al modelar un protocolo podemos encontrarnos con ataque omitidos.

5.2. Trabajo futuro

El estudio de esta propiedad y la inclusión en la herramienta de la asociatividad como axioma ecuacional, nos abre nuevas vías de investigación entre ellas podemos encontrar protocolos que poseen ataques conocidos que explotan la asociatividad pero que al ser modelados para analizar este ataque en concreto, no puede encontrar otros y viceversa. Intentaremos pues utilizar un único modelo capaz de encontrar todos los ataque que utilicen la asociatividad, como objetivo tomaremos un protocolo explicado en [11].

Por ultimo nos plantearemos como esta propiedad podría influir en protocolos ya modelados, en los que se podrían encontrar ataques nuevos y en explotar otras capacidades de la herramienta al buscar soluciones finitas a protocolos pues la asociatividad tiende a ser infinita esto lo podemos observar en la herramienta cuando nos lanza “warnings” .

Bibliografía

[1] S. Escobar, C. Meadows y J. Meseguer, Maude-NPA,Version 2.0, 2011.

[2] M. Clavel, F. Durán, S. Elker, P. Lincoln, N. Martí-Oliet, J. Meseguer y C. Talcott, All About Maude - A High - Performance Logical Framework, 2007.

[3] «Strand Spaces: What Makes a Security Protocolo Correct?,» *Journal of Computer Scurity,* vol. 7, pp. 191-230, 1999.

[4] M. Clavel, F. Durán, S. Elker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott y S. Escobar, «Unification and Narrowing in Maude 2.4,» de *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, 2009, pp. 380-390.

[5] J. Clark y J. Jacob, A Survey of Authentication Protocol, 1997.

[6] R. Needham y M. Schroeder, «Using encryption for authentication in large networks of computers,» *Communications of the ACM,* vol. 21, pp. 993-999, 1978.

[7] G. Lowe, «An attack on the Needham-Schroeder public-key authentication protocol,» *Information Processing Letters,* vol. 56, pp. 131-133, 1995.

[8] V. Cortier, S. Delaune y P. Lafourcade, «A survey of algebraic properties used in cryptographic protocols,» *Journal of Computer Security,* nº 14.

[9] S. Escobar, C. Meadows y J. Meseguer, Equational Cryptographic Reasoning in the Maude-NRL Protocol Analyzer, 2006.

[10] J. Heather, G. Lowe y S. Schneider, «How to Prevent Type Attacks on Security Protocols,» de *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, 2000, pp. 255-268.

[11] S. Escobar, C. Meadows y J. Meseguer, Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties, 2007.

[12] D. Dolev y A. Yao, «On the security of public key protocols,» vol. 29, pp. 198-208, 1983.

[13] M. Clavel, F. Durán, S. Elker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. Quesada, «Maude: specification and programming in rewriting logic,» *Theoretical Computer Science,* nº 285, pp. 187-243, 202.

[14] C. Meadows, P. Syverson y I. Cervesato, «Formal specification and analysis of the Group Domain Of Interpretation Protocol using NPATRL and the NRL Protocol Analyzer,» *Journal of Computer Security,* nº 12, pp. 893-931, 2004.

Anexo

Protocolo Needham-Schroeder-Lowe sin ataque

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  protecting DEFINITION-PROTOCOL-RULES .

--- Sort Information
        sorts Name Nonce Key .
        subsort Name Nonce Key < Msg .
        subsort Name < Key .
        subsort Name < Public .

--- Encoding operators for public/private encryption
        op pk : Key Msg -> Msg [frozen] .
        op sk : Key Msg -> Msg [frozen] .

--- Nonce operator
        op n : Name Fresh -> Nonce [frozen] .

--- Principals
        op a : -> Name . --- Alice
        op b : -> Name . --- Bob
        op i : -> Name . --- Intruder

--- Associativity operator
        op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
protecting PROTOCOL-EXAMPLE-SYMBOLS .

        var Z : Msg .
        var Ke : Key .

---Encryption/Decryption Cancellation
        eq pk(Ke,sk(Ke,Z)) = Z [variant] .
        eq sk(Ke,pk(Ke,Z)) = Z [variant] .

endfm
```

```
fmod PROTOCOL-SPECIFICATION is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .
        protecting DEFINITION-PROTOCOL-RULES .
        protecting DEFINITION-CONSTRAINTS-INPUT .

        var Ke : Key .
        vars X Y Z : Msg .
        vars r r' : Fresh .
        vars A B : Name .
        vars N N1 N2 : Nonce .

eq STRANDS-DOLEVYAO
      = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
        :: nil :: [ nil | -(X ; Y), +(X), nil ] &
        :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
        :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
        :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ] &
        :: nil :: [ nil | +(A), nil ]
[nonexec] .

eq STRANDS-PROTOCOL
        = :: r ::
                [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; N ; B)), +(pk(B, N)), nil ] &
        :: r ::
                [ nil | -(pk(B,A ; N)), +(pk(A, N ; n(B,r) ; B)), -(pk(B,n(B,r))), nil ]
[nonexec] .

eq ATTACK-STATE(o)
        = :: r ::
                [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]
                || n(b,r) inI, empty
                || nil
                || nil
                || nil
[nonexec] .
endfm
```

Protocolo Needham-Schroeder-Lowe con regla ecuacional.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  protecting DEFINITION-PROTOCOL-RULES .

--- Sort Information
        sorts Name Nonce Key Data Elm List .
        subsort Name Nonce Key Data Elm List < Msg .
        subsort Name < Key .
        subsort Name < Public .
        subsort Name < Data .
        subsort Data < List .

--- Encoding operators for public/private encryption
        op pk : Key Msg -> Msg [frozen] .
        op sk : Key Msg -> Msg [frozen] .

--- Nonce operator
        op n : Name Fresh -> Nonce [frozen] .

--- Principals
        op a : -> Name . --- Alice
        op b : -> Name . --- Bob
        op i : -> Name . --- Intruder

--- Associativity operator
        op _;_ : Msg  Msg  -> List [gather (e E) frozen] .

        op _;_ : Nonce Name  -> Data [gather (e E) frozen] ..

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
protecting PROTOCOL-EXAMPLE-SYMBOLS .

        var Z : Msg .
        var Ke : Key .
        vars Xe Ye Ze : Elm .

---Encryption/Decryption Cancellation
        eq pk(Ke,sk(Ke,Z)) = Z [variant] .
        eq sk(Ke,pk(Ke,Z)) = Z [variant] .

*** Bounded Associativity (for 3-depth)
  eq (Xe ; Ye) ; Ze = Xe ; (Ye ; Ze) [variant] .

endfm
```

```
fmod PROTOCOL-SPECIFICATION is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .
        protecting DEFINITION-PROTOCOL-RULES .
        protecting DEFINITION-CONSTRAINTS-INPUT .

        var Ke : Key .
        vars X Y Z : Msg .
        vars r r' : Fresh .
        vars A B : Name .
        vars N N1 N2 : Nonce .
        var V : Data .

eq STRANDS-DOLEVYAO
      = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
        :: nil :: [ nil | -(X ; Y), +(X), nil ] &
        :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
        :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
        :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ] &
        :: nil :: [ nil | +(A), nil ]
[nonexec] .

eq STRANDS-PROTOCOL
        = :: r ::
                [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; V ; B)), +(pk(B, V)), nil ] &
        :: r ::
                [ nil | -(pk(B,A ; V)), +(pk(A, V ; n(B,r) ; B)), -(pk(B,n(B,r))), nil ]
[nonexec] .

eq ATTACK-STATE(o)
        = :: r ::
                [ nil, -(pk(b,a ; V)), +(pk(a, V ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]
                || n(b,r) inI, empty
                || nil
                || nil
                || nil
[nonexec] .
endfm
```

Protocolo Needham-Schroeder-Lowe con axioma ecuacional y no limitado.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  protecting DEFINITION-PROTOCOL-RULES .

--- Sort Information
        sorts Name Nonce Key .
        subsort Name Nonce Key < Msg .
        subsort Name < Key .
        subsort Name < Public .

--- Encoding operators for public/private encryption
        op pk : Key Msg -> Msg [frozen] .
        op sk : Key Msg -> Msg [frozen] .

--- Nonce operator
        op n : Name Fresh -> Nonce [frozen] .

--- Principals
        op a : -> Name . --- Alice
        op b : -> Name . --- Bob
        op i : -> Name . --- Intruder

--- Associativity operator
        op _;_ : Msg  Msg  -> Msg [gather (e E) frozen assoc] .

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
protecting PROTOCOL-EXAMPLE-SYMBOLS .

        var Z : Msg .
        var Ke : Key .

---Encryption/Decryption Cancellation
        eq pk(Ke,sk(Ke,Z)) = Z [variant] .
        eq sk(Ke,pk(Ke,Z)) = Z [variant] .

endfm
```

```
fmod PROTOCOL-SPECIFICATION is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .
        protecting DEFINITION-PROTOCOL-RULES .
        protecting DEFINITION-CONSTRAINTS-INPUT .

        var Ke : Key .
        vars X Y Z M : Msg .
        vars r r' : Fresh .
        vars A B : Name .
        vars N N1 N2 : Nonce .

eq STRANDS-DOLEVYAO
      = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
        :: nil :: [ nil | -(X ; Y), +(X), nil ] &
        :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
        :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
        :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ] &
        :: nil :: [ nil | +(A), nil ]
[nonexec] .

eq STRANDS-PROTOCOL
        = :: r ::
                [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; M ; B)), +(pk(B, M)), nil ] &
        :: r ::
                [ nil | -(pk(B,A ; M)), +(pk(A, M ; n(B,r) ; B)), -(pk(B,n(B,r))), nil ]
[nonexec] .

eq ATTACK-STATE(o)
        = :: r ::
                [ nil, -(pk(b,a ; M)), +(pk(a, M ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]
                || n(b,r) inI, empty
                || nil
                || nil
                || nil
[nonexec] .
endfm
```

Protocolo Needham-Schroeder-Lowe con axioma ecuacional y limitado.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  protecting DEFINITION-PROTOCOL-RULES .

--- Sort Information
        sorts Name Nonce Key Data .
        subsort Name Nonce Key Data < Msg .
        subsort Name < Key .
        subsort Name < Public .
        subsort Name < Data .

--- Encoding operators for public/private encryption
        op pk : Key Msg -> Msg [frozen] .
        op sk : Key Msg -> Msg [frozen] .

--- Nonce operator
        op n : Name Fresh -> Nonce [frozen] .

--- Principals
        op a : -> Name . --- Alice
        op b : -> Name . --- Bob
        op i : -> Name . --- Intruder

--- Associativity operator
        op _;_ : Msg  Msg  -> List [gather (e E) frozen assoc] .

        op _;_ : Nonce Name  -> Data [gather (e E) frozen assoc] .

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
protecting PROTOCOL-EXAMPLE-SYMBOLS .

        var Z : Msg .
        var Ke : Key .
        vars Xe Ye Ze : Elm .

---Encryption/Decryption Cancellation
        eq pk(Ke,sk(Ke,Z)) = Z [variant] .
        eq sk(Ke,pk(Ke,Z)) = Z [variant] .

endfm
```

```
fmod PROTOCOL-SPECIFICATION is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .
        protecting DEFINITION-PROTOCOL-RULES .
        protecting DEFINITION-CONSTRAINTS-INPUT .

        var Ke : Key .
        vars X Y Z M : Msg .
        vars r r' : Fresh .
        vars A B : Name .
        vars N N1 N2 : Nonce .
        var V : Data .

eq STRANDS-DOLEVYAO
      = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
        :: nil :: [ nil | -(X ; Y), +(X), nil ] &
        :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
        :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
        :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ] &
        :: nil :: [ nil | +(A), nil ]
[nonexec] .

eq STRANDS-PROTOCOL
        = :: r ::
                [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; V ; B)), +(pk(B, V)), nil ] &
        :: r ::
                [ nil | -(pk(B,A ; V)), +(pk(A, V ; n(B,r) ; B)), -(pk(B,n(B,r))), nil ]
[nonexec] .

eq ATTACK-STATE(o)
        = :: r ::
                [ nil, -(pk(b,a ; V)), +(pk(a, V ; n(b,r) ; b)), -(pk(b,n(b,r))) | nil ]
                || n(b,r) inI, empty
                || nil
                || nil
                || nil
[nonexec] .
endfm
```

Protocolo Secret2016 con asociatividad mediante reglas ecuacionales.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
protecting DEFINITION-PROTOCOL-RULES .

--- Sort Information
        sorts Name Nonce Key .
        sorts List Elm .
        subsort Name < Key .
        subsort Name Nonce Key < Elm .
        subsort Elm < Msg .
        subsort List < Msg .
        subsort Name < Public .

--- Encoding operators for public/private encryption
        op pk : Key Msg -> Msg [frozen] .
        op sk : Key Msg -> Msg [frozen] .

--- Keys known by intruder
        op s : -> Name . --- Name for the Server
        op a : -> Name . --- Name for the Initiator
        op b : -> Name . --- Name for the Responder
        op i : -> Name . --- Name for the Intruder

--- Nonce operator
        op n : Name Fresh -> Nonce [frozen] .

--- Associativity operator
        op _;_ : Msg  Msg  -> List [frozen] .

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .

        var Z : Msg .
        var Ke : Key .
        vars Xe Ye Ze : Elm .

--- Encryption/Decryption Cancellation
        eq pk(Ke,sk(Ke,Z)) = Z [variant] .
        eq sk(Ke,pk(Ke,Z)) = Z [variant] .

---Bounded Associativity (for 3-depth)
        eq Xe ; (Ye ; Ze) = (Xe ; Ye) ; Ze [variant] .

endfm
```

```
fmod PROTOCOL-SPECIFICATION is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .
        protecting DEFINITION-PROTOCOL-RULES .
        protecting DEFINITION-CONSTRAINTS-INPUT .

        var Ke : Key .
        vars X Y Z : Msg .
        var r : Fresh .
        var N : Nonce .

eq STRANDS-DOLEVYAO
        =       :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
                :: nil :: [ nil | -(X ; Y), +(X), nil ] &
                :: nil :: [ nil | -(X ; Y), +(Y), nil ] &

--- Private encryption only his key
                :: nil :: [ nil | -(X), +(sk(i,X)), nil ]  &
--- Public encryption any key
                :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ]
[nonexec] .

eq STRANDS-PROTOCOL
  =     --- server
                :: r :: [nil | +(n(s,r)), nil ] &
        --- initiator
                :: r :: [nil | -(N), +(sk(a,N ; s)), +(sk(a,b ; (n(a,r) ; s))), nil] &
        --- responder
                :: nil :: [nil | -(sk(a,X ; s)), -(sk(a,(b ; Z) ; s)), nil]
[nonexec] .

eq ATTACK-STATE(o)
        = :: nil :: [nil, -(sk(a,X ; s)), -(sk(a,(b ; N) ; s)) | nil]
        || empty
        || nil
        || nil
        || nil
[nonexec] .

endfm
```

Protocolo Secreto6 con axioma ecuacional

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
protecting DEFINITION-PROTOCOL-RULES .

--- Sort Information
        sorts Name Nonce Key .
        subsort Name < Key .
        subsort Name Nonce Key < Msg .
        subsort Name < Public .

--- Encoding operators for public/private encryption
        op pk : Key Msg -> Msg [frozen] .
        op sk : Key Msg -> Msg [frozen] .

--- Keys known by intruder
        op s : -> Name . --- Name for the Server
        op a : -> Name . --- Name for the Initiator
        op b : -> Name . --- Name for the Responder
        op i : -> Name . --- Name for the Intruder

--- Nonce operator
        op n : Name Fresh -> Nonce [frozen] .

--- Associativity operator
        op _;_ : Msg  Msg  -> Msg [assoc frozen] .

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .

        var Z : Msg .
        var Ke : Key .

--- Encryption/Decryption Cancellation
        eq pk(Ke,sk(Ke,Z)) = Z [variant] .
        eq sk(Ke,pk(Ke,Z)) = Z [variant] .

endfm
```

```
fmod PROTOCOL-SPECIFICATION is
        protecting PROTOCOL-EXAMPLE-SYMBOLS .
        protecting DEFINITION-PROTOCOL-RULES .
        protecting DEFINITION-CONSTRAINTS-INPUT .

        var Ke : Key .
        vars X Y Z : Msg .
        var r : Fresh .
        var N : Nonce .

eq STRANDS-DOLEVYAO
        =       :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
                :: nil :: [ nil | -(X ; Y), +(X), nil ] &
                :: nil :: [ nil | -(X ; Y), +(Y), nil ] &

--- Private encryption only his key
                :: nil :: [ nil | -(X), +(sk(i,X)), nil ]  &
--- Public encryption any key
                :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ]
[nonexec] .

eq STRANDS-PROTOCOL
  =     --- server
                :: r :: [nil | +(n(s,r)), nil ] &
        --- initiator
                :: r :: [nil | -(N), +(sk(a,N ; s)), +(sk(a,b ; (n(a,r) ; s))), nil] &
        --- responder
                :: nil :: [nil | -(sk(a,X ; s)), -(sk(a,(b ; Z) ; s)), nil]
[nonexec] .

eq ATTACK-STATE(o)
        = :: nil :: [nil, -(sk(a,X ; s)), -(sk(a,(b ; N) ; s)) | nil]
        || empty
        || nil
        || nil
        || nil
[nonexec] .

endfm
```