Universidade do Minho
Departamento de informática

Tese de mestrado

Ferramentas de Verificação Formal de Protocolos Criptográficos

Inês Isabel Russo Prada pg15992

Orientador: Prof. José Carlos Bacelar Almeida

Documento apresentado no âmbito do Mestrado em engenharia informática, como requisito parcial para obtenção do grau de Mestre em Engenharia informática.

31 de Outubro de 2011

Agradecimentos

Às minhas irmãs Ana e Luísa, aos meus pais e avôs pela paciência e incentivo todos os dias.

Ao Prof. José Carlos Bacelar Almeida pelo contínuo apoio e presença, pela forma como tornou proveitosas e dinâmicas as orientações da tese. Por ter cultivado ainda mais o meu gosto no mundo tão vasto da criptografia e da verificação formal.

Resumo

Ao longo destes anos o número de aplicações distribuídas e o uso da Internet têm aumentado consideravelmente. Muitas destas aplicações críticas efectuam um conjunto de operações sensíveis, manipulando frequentemente dados privados e confidenciais. Torna-se deste modo importante que, antes de usufruir de uma estrutura deste tipo, se averigúe quais são as políticas de segurança da aplicação em questão de modo a que, mais tarde, o utilizador não tenha surpresas indesejadas.

Os protocolos criptográficos constituem um recurso importante nos componentes dos sistemas encarregues de fornecer as garantias de segurança pretendidas. Tratam-se de protocolos de comunicação normalmente pequenos, que fazem uso de técnicas criptográficas, e que têm objectivos bem especificados como sejam o estabelecimento de uma chave de sessão ou de garantias de autenticidade na comunicação. A natureza crítica desses protocolos justifica que se invista numa análise rigorosa (i.e. formal) desses protocolos, garantindo dessa forma que eles cumprem a função para que foram desenhados. No entanto, a análise desses protocolos tem-se revelado um problema complexo, mesmo quando se atribui um comportamento idealizado às operações criptográficas empregues. Por esse motivo, a comunidade científica tem vindo a propor novas metodologias e ferramentas que auxiliam no processo de verificação formal de protocolos criptográficos.

Neste trabalho, iremos apresentar um conjunto representativo de ferramentas para a verificação formal de protocolos criptográficos. Após a exposição das suas principais características e dos respectivos modelos que as suportam, iremos emprega-las na análise de fragmentos de um protocolo concreto que se tem implantado como um *standard* de facto na internet: o protocolo de autenticação centralizada (Single Sign-On) "OpenID".

Palavras-chave: Autenticação, AVISPA, Criptoverif, OpenID, Protocolos criptográficos, Protocolos de Segurança, Proverif, Sistemas de Autenticação Global, Verificação Formal

Abstract

Over these years the number of distributed applications and the Internet use have increased considerably. Many of these critical applications perform a set of sensitive operations that often manipulate private and confidential data. In this way, it's important before we use one of these critical applications, determine the security policies of the application in question, just to the user doesn't have surprises later.

The cryptographic protocols are an important resource of the systems components that have the charge of provide the desired security guarantees. These cryptographic protocols, usually small, make use of cryptographic techniques, whose aims are well specified, such as the settlement of a session key or some guarantees of communication authenticity. The critical nature of these protocols justifies that we must do a rigorous analysis (i.e. formal) of these protocols. In this way, we guarantee that they fit the function for what they were designed. However, the analysis of these protocols has showed to be a complex problem, even when it is attributed an idealized behavior for the cryptographic operations employed. That's why the scientific community has proposed new methodologies and tools that help in the formal verification process of cryptographic protocols.

In this work, we introduce a representative set of tools for the formal verification of cryptographic protocols. After we demonstrate their main characteristics and models that support them, we will use those tools in the analysis of the fragments of one concrete protocol that has implemented like a standard indeed on the internet: the centralized authentication protocol (Single Sign-On) "OpenID".

Keywords: Authentication, AVISPA, Criptoverif, Cryptographic protocols, Formal Verification, OpenID, Proverif, Security Protocols , Single Sign-On Systems

Conteúdo

Lista de Figuras

Lista de Tabelas

1. Introdução

1.1 Contextualização do problema

Apesar das questões de segurança não serem novidade, o aumento na utilização de aplicações distribuídas e da internet faz aumentar dúvidas sobre que garantias é legítimo possuir relativamente à segurança da informação trocada nessas aplicações. Todos os dias, sem nos apercebermos, usamos um número considerável de aplicações distribuídas para executar serviços como, transferências bancárias, comércio electrónico, voto electrónico ou outro tipo de serviços. Para este tipo de aplicações críticas, a segurança merece particular destaque. Neste ambiente distribuído, cujos canais de comunicação muitas vezes não são fiáveis, torna-se essencial garantir propriedades de segurança de modo a aumentar a confiança dos utilizadores relativamente ao uso dessas mesmas aplicações. Deste modo, durante os últimos vinte anos, pequenas aplicações distribuídas, nomeadamente protocolos de segurança baseados em técnicas criptográficas, começaram a ser usados.

Um protocolo de segurança tem como objectivo atingir um conjunto de propriedades de segurança que podem variar consoante as necessidades e o próprio ambiente da aplicação. No entanto a história mostrou que, mesmo aqueles protocolos criptográficos que utilizam um conjunto de operações criptográficas dadas como ideais, podem apresentar falhas e não serem seguros como se considerava. Além disso, um protocolo de segurança pode ser executado por uma grande população, incluindo um atacante, que pode explorar as vulnerabilidades e comprometer todo o propósito do protocolo.

Na tentativa de resolver este problema, a verificação formal, que inclui um conjunto de técnicas assente em bases matemáticas, permitindo-nos assim ter certezas sobre os resultados obtidos, passou a ser utilizada. Com a ajuda destas e outras técnicas, governos e organismos internacionais de normalização definiram um conjunto de procedimentos e normas de segurança. No entanto, tais "padrões" de segurança não estão presentes em todas as aplicações críticas, sendo por vezes complicado para o utilizador saber o quanto confiáveis são. Este é um problema muito grave, pois certas aplicações críticas lidam com serviços e informação muito "sensível", pondo em risco os utilizadores. Deste modo, nesta tese propomos o uso de ferramentas que permitem a verificação formal, de forma a os utilizadores terem certezas. Durante alguns anos as provas teriam de ser feitas unicamente à mão. Actualmente, existe um conjunto de ferramentas gratuitas que nos permite, de forma automatizada, executar muitos dos passos do processo da verificação, simplificando todo o trabalho. Além disso, uma prova manual por vezes é muito difícil e até mesmo impraticável, face ao tempo que temos para verificar muitos dos protocolos de segurança actuais, que cada vez são mais complexos. Com a utilização de ferramentas que permitem a verificação formal, todo o processo de verificação e análise se torna muito mais simples e rápido de realizar. De salientar que todo esse processo é muito importante nas etapas iniciais de desenvolvimento de uma aplicação, pois quanto mais cedo se detectarem erros melhor, visto que não haverá quaisquer desperdícios de recursos a implementar soluções

erradas.

Actualmente há uma grande variedade de modelos formais para a especificação e verificação de protocolos de segurança, sendo importante analisar as suas características e determinar para que tipos de problemas se adaptam melhor. As ferramentas gratuitas que permitem a análise formal baseiam-se em diferentes modelos formais e possuem características distintas. Nesta tese, aprofundamos e comparamos três dessas ferramentas, nomeadamente, o CryptoVerif, o ProVerif e a AVISPA.

Nos últimos anos, o protocolo OpenID tem ganho alguma notoriedade. Trata-se de um protocolo descentralizado e simples, que implementa um sistema de autenticação global, que surgiu como uma resposta simples e eficaz para a problemática de gestão de identidades digitais[1], com o objectivo de resolver as necessidades dos utilizadores. Algumas redes sociais tais como, Twitter, Orkut, Facebook, permitem a autenticação via OpenID. No entanto, algumas dúvidas sobre a segurança do OpenID têm surgido, nomeadamente, sobre a correcção do protocolo openID, o quanto fiáveis são os provedores de identidade OpenID[2] e sobre o sigilo de informação, nomeadamente, credenciais de autenticação ou até mesmo informação pessoal do utilizador. Muitos autores defendem que estes sistemas de autenticação global podem aumentar a segurança, pois os problemas de segurança associados a uma má gestão de credenciais desaparecem. Por outro lado, outros autores afirmam que, independentemente do sistema deste tipo, qualquer implementação de um mecanismo que permita a autenticação global enfraquece a segurança, pois tudo o que se tem de fazer é quebrar uma única palavra-passe para aceder a todos os serviços [6].

Torna-se, por isso, interessante analisar e questionar a segurança num sistema de autenticação como o OpenID. Sendo a segurança e a confiabilidade de tais sistemas essenciais, quer para a sua aceitação, quer para a sua evolução, o uso da verificação formal torna-se uma ajuda preciosa. No entanto, sendo um protocolo com alguma complexidade é muito melhor recorrer a ferramentas que suportem a verificação formal. Estas ferramentas, que utilizam métodos matemáticos e lógicos, podem ser bastante úteis, ajudando na procura de falhas e sugerindo, por vezes, possíveis correcções nos protocolos que são objecto de análise.

1.2 Objectivos

Propomos os seguintes objectivos:

- Averiguar quais as potencialidades da verificação formal;
- Determinar as vantagens de se realizar uma prova de segurança através de uma ferramenta que suporte verificação formal, em vez de uma prova feita à mão;

[1] forma de representar digitalmente uma identidade que pode ser vista como um conjunto de dados e características, agregadas ou não, relacionados com um sujeito. Essa mesmo, pode não ser necessariamente uma pessoa, mas, como por exemplo, uma máquina.

[2] responsável por fornecer uma identidade OpenID em URL ou XRI aos seus utilizadores.

- Analisar os diferentes modelos formais propostos para a especificação de protocolos de segurança;
- Estudar algumas das ferramentas que suportam verificação formal, nomeadamente, CriptoVerif, ProVerif e Avispa, baseadas em diferentes modelos formais, e determinar quais os cenários em que se adaptam melhor;
- Utilizar as ferramentas abordadas anteriormente para analisar o protocolo de autenticação OpenID 2.0 ;
- Sensibilizar para o uso de ferramentas que permitam a análise formal de protocolos de segurança.

1.3 Estrutura do documento

A estrutura desta dissertação é a seguinte:

- No Capítulo 2 é apresentado o problema da verificação de segurança em protocolos criptográficos e as diferentes ferramentas que foram objecto de estudo neste trabalho;
- O Capítulo 3 é dedicado à apresentação dos sistemas *Single Sign-On* e do protocolo OpenID;
- No Captítulo 4 apresenta-se a análise de fragmentos do protocolo OpenID com recurso às ferramentas já referidas;
- Por último, conclui-se no Capítulo 5 com uma reflexão sobre o trabalho realizado.

2 . Verificação formal de protocolos criptográficos

2.1 Introdução

Muitas das aplicações críticas existentes, devido à sua arquitectura distribuída e ao facto de serem executadas num ambiente hostil, levam à necessidade de mecanismos para se garantirem determinadas propriedades de segurança, que são normalmente estabelecidas com a ajuda de pequenas aplicações distribuídas, designadas por, protocolos criptográficos (ou protocolos de segurança). Estes protocolos recorrem à criptografia de modo a que os diversos participantes (agentes) possam trocar informação de forma segura. Informalmente um protocolo de segurança pode ser visto como um acordo entre dois ou mais agentes com o objectivo de garantir determinadas propriedades de segurança, mesmo que um intruso malicioso tenha acesso ao canal de comunicação em questão.

Durante muitos anos, achou-se que os protocolos de segurança que utilizavam as operações criptográficas dadas como ideais seriam suficientes para garantir a sua fiabilidade, prescindindo de qualquer espécie de análise. Com o decorrer do tempo esta ideia mostrou ser completamente errada. Muitos dos protocolos dados como seguros mostraram ser alvo de ataques, mesmo num contexto de criptografia perfeita em que as operaçõos criptográficas utilizadas eram dadas como ideais, ou seja, num contexto em que as primitivas criptográficas se comportam como *caixa preta* que satisfazem algumas propriedades, tais como: uma mensagem cifrada só pode ser recuperada pelos possuidores da chave de descodificação. Verificou-se que apesar das primitivas criptográficas usadas num protocolo serem perfeitamente seguras, na totalidade do protocolo podiam existir pontos fracos. Deste modo, torna-se essencial um mecanismo para averiguar se os requisitos de segurança são cumpridos.

Visto isto, afirmamos que desenvolver protocolos criptográficos pode ser algo muito mais complexo e até um verdadeiro desafio, sendo necessário mecanismos para análise de protocolos criptográficos. No entanto, antes de qualquer análise e verificação formal torna-se necessário averiguar, com cuidado, todos os requisitos de segurança pretendidos para a aplicação. Tal como Boehm referiu, são necessárias boas especificações , caso contrário, é impossível qualquer processo de verificação [29]. Deste modo, a fase de especificação, que consiste em descrever um sistema considerando as suas propriedades desejadas, é essencial. A especificação de um sistema exige, assim, que se tenha um bom conhecimento do problema de modo a termos certezas sobre quais as propriedades relevantes no sistema, caso contrário não é possível alcançar o objectivo desejado.

Em suma, a validação de um protocolo criptográfico pode ser preenchido por métodos formais que englobam um conjunto de métodos matematicamente precisos. Desta forma, podemos dar declarações exatas sobre os resultados da análise de um protocolo. De destacar que, durante as duas úlltimas décadas a comunidade de segurança tem feito significativos avanços em métodos formais e ferramentas com suporte à verificação formal para análise de protocolos criptográficos. Deste modo, os protocolos criptográficos podem ser analisados manualmente ou com

recurso a ferramentas automáticas que suportem análise formal. No entanto, sendo a análise feita unicamente à mão, uma tarefa por vezes demorosa e difícil, verificou-se ser muito mellhor usar ferramentas que suportam uma análise formal de protocolos de segurança, libertando a pessoa que esteja a efectuar a prova de alguns passos monótonos ou até complicados de realizar sem apoio a nenhum recurso. Além disso, a escala e complexidade de muitos protocolos de segurança actuais aumentou consideravelmente, sendo cada vez mais difícil fazer uma prova manual sem qualquer recurso automático. Tendo em consideração o cenário e as limitações humanas torna-se complicado analisar e validar com rigor esses protocolos num curto espaço de tempo. Deste modo, o uso de ferramentas que suportam a análise formal de protocolos torna-se cada vez mais indicado.

2.2 Propriedades de segurança

Uma propriedade de segurança corresponde a um dos objectivos que um protocolo criptográfico pretende atingir. Falámos em cumprimento de propriedades de segurança num protocolo, pois o conceito segurança é bastante abstracto. Os protocolos podem ter diferentes níveis de definições, que variam com o grau de abstracção de certas políticas de segurança. No entanto, na generalidade, a segurança é definida com recurso a algumas propriedades simples, como seja, a confidencialidade[1], autenticação [2] e não repúdio[3] [33, 20].

2.3 Protocolos criptográficos

Os protocolos de segurança podem ser descritos através de uma sequência de mensagens, que constituem um conjunto de regras, de modo a atingirem os objectivos, nomeadamente, o cumprimento das propriedades de segurança.

A este conjunto de regras designamos por especificação de protocolo. Essas mesmas regras podem ser relativas a um conjunto de acções dos agentes que irão definir um grafo de transições de estados. Um caminho deste grafo corresponderá a um traço do protocolo, ou seja, uma sequência de ações/eventos possíveis do protocolo.

Um exemplo simples da descrição de um protocolo de segurança podia ser, por exemplo, dado pela seguinte especificação:

```
A -> B: {Na}_K
B -> A: {Nb}_K
A -> B: {Nb}_K1 , where K1=hash(Na.Nb)
```

A esta especificação corresponde um protocolo que tem como objectivo definir uma chave de sessão (K1) entre dois agentes A e B. Os agentes A e B, possuidores de uma chave simétrica

[1] em nenhum momento, o intruso pode conhecer determinada informação trocada entre agentes.

[2] determinado valor acordado ou agentes deve ser autêntico.

[3] o emissor não pode negar a autenticidade da mensagem.

K partilhada entre eles, irão trocar mensagens entre eles de forma a definirem um valor único para a chave de sessão. O protocolo é iniciado pelo agente A que envia um nonce[4] (Na) cifrado pela chave K, em que apenas os agente possuidor dessa chave consegue decifrar tal mensagem. Depois o agente B envia outro nonce (Nb) a A cifrado com essa mesma chave. De seguida, o agente A deve enviar a B o nonce recebido cifrado pela chave de sessão criada que corresponde ao hash da concatenação de Na e Nb. Finalmente, o agente B deve receber o mesmo valor do nonce que gerou.

Este protocolo tem como objectivo criar uma chave de sessão entre dois agentes segundo algumas condições de segurança que correspondem a determinadas propriedades de segurança, nomeadamente:

- Confidencialidade - O atacante não pode conhecer a chave de sessão acordada entre os agentes, devendo permanecer em segredo;

- Autenticidade de valores criados e segredo - Cada um dos agentes (A ou B) tem prova que os valores de Nb e Na foram respectivamente enviados por B e A. Adicionalmente, B tem garantia que A conclui com sucesso o protocolo (conhece K1)

De salientar, que as regras como as que se apresentaram pretendem capturar "um esquema" da execução do protocolo, em que os identificadores usados devem ser considerados como "variáveis". Este esquema vai poder ser instanciado um número arbitário de vezes, possivelmente de forma concorrente, dando origem ao que normalmente é designado por uma sessão do protocolo. Uma sessão, ou instância do protocolo, corresponde a uma imagem do protocolo depois da substituição das variáveis por valores concretos.

2.4 Verificação formal de determinadas propriedades de segurança

Na tentativa de aumentar a confiança dos utilizadores, a verificação formal começou a ser utilizada na correção de protocolos de segurança, tendo-se mostrado bastante útil. Uma linguagem de especificação é usada para descrever o protocolo e os seus requisitos de segurança, e um sistema de raciocínio formal pode ser utilizado para verificar se os requisitos de segurança do protocolo são cumpridos. De salientar que a sintaxe e semântica da notação da linguagem de especificação têm como base conceitos matemáticos, definindo assim um modelo matemático do sistema em causa. Assim, a verificação dos sistemas é feita através de uma prova formal do modelo matemático e abstracto do sistema.

2.4.1 Especificação e análise de protocolos criptográficos

No passado, a maioria das linguagens formais de especificação de protocolos de segurança eram limitadas. Escalabilidade, expressividade e facilidade de uso eram alguns dos problemas.

[4]corresponde a um identificador, normalmente um número grande aleatório, que é usado uma só vez.

Actualmente, existem várias linguagens de especificação e, além disso, outras técnicas de análise foram desenvolvidas tornando todo o processo de especificação e análise uma tarefa menos tediosa, difícil e propensa a erros [28].

De forma sucinta, a especificação do protocolo consiste numa sequência de mensagens trocadas entre intervenientes, a uma sucessão finita de regras da forma A-> B: M, onde M é a mensagem trocada entre agentes (neste caso A e B). O significado (informal) pretendido é que A envia para B uma mensagem M em um canal público e inseguro [20]. Além disso, a linguagem permite especificar as ações realizadas por cada um dos agentes (e.g. geração de números aleatórios, verificação de determinadas condições, etc), assim como todas as capacidades do adversário, ambiente de execução, como seja o conhecimento inicial de cada um dos agentes e a especificação das propriedades de segurança pretendidas.

2.4.1.1 Modelos

No estudo de protocolos criptográficos, Blanchet considera [12] que apenas existem duas frameworks para a verificação de protocolos de segurança: o Modelo Formal e o Modelo Computacional.

	Modelo Formal	Modelo computacional
Propriedades de segurança modeladas	com ferramentas alto nível	através de noções de baixo nível
Primitivas ciptográficas	símbolos de funções numa álgebra de termos	funções de bitstrings
Provas	mais fáceis e menos propensas a erros	mais reais e detalhadas => mais complexas e mais sujeitas a erros
Segurança baseada nos recursos do atacante	não	sim

Tabela 2.1 Modelo Formal vs Modelo Computacional

No Modelo Formal, também designado por Modelo Simbólico ou Modelo Dolev-Yao, as mensagens são interpretadas como termos de uma linguagem formal. Como consequência o modelo fornece uma idealização forte de operações criptográficas reais, representando-as através de símbolos de função na álgebra de termos. Esta idealização simplifica a construção da prova, negliceando certos detalhes, tais como restrições computacionais, comportamento probabilístico, e probabilidades de erro. Além disso, opta pelo uso de premissas bastante fortes numa álgebra de termos com regras de cancelamento. No entanto, ao trabalharmos em alto nível, faz com que não tenhamos tanta liberdade descritiva e a prova seja menos realista. Por outro lado, essa limitação evita erros que podem ser cruciais para a prova.

Em contraste, com o modelo anterior temos o Modelo Computacional, onde as provas são construídas, tal como na teoria da complexidade, por redução. O atacante é modelado como

uma máquina de Turing[5] probabilística e as definições de segurança criptográfica definidas em termos de teoria da probabilidade e teoria da complexidade. Nesta abordagem as mensagens trocadas são normalmente modeladas simplesmente por sequências de bits (bitstrings), tratando-se assim duma abordagem que se pode qualificar de mais baixo nível do que no modelo formal. Naturalmente que esse facto explica também porque é que as provas de segurança se tornam numa tarefa muito mais complexa. No modelo computacional as provas são realizadas por redução: partindo-se de um sistema que captura a noção de segurança pretendida sobre o protocolo em causa (o que se designa por "jogo de segurança"), realizam-se transformações cujo impacto seja negligenciável (relativamente a um parâmetro de segurança) nas características observáveis desse sistema. A segurança é estabelecida quando se atinge uma configuração que pressuponha a resolução de um problema tido como intratável. O número de passos envolvidos na redução pode ser considerável e os argumentos que justificam cada um desses passos podem ser de uma complexidade considerável. Em conclusão, podemos afirmar que as provas no modelo computacional são de uma natureza mais complexa do que a do modelo formal. Por outro lado, as provas feitas no modelo computacional são mais realistas, já que as mensagens são bitstrings e o adversário pode ser modelado como qualquer algoritmo eficiente.

Nas diferenças entre os dois modelos destacamos: a representação das mensagens e o poder que eles dão ao atacante. No modelo computacional, as mensagens são modeladas como distribuições de probabilidade sobre bitstrings(indexadas pelo parâmetro de segurança). O atacante é modelado com um algoritmo de poder computacional realista: probabilístico de tempo polinomial, PPT. Contrariamente, no Modelo Formal as mensagens são construídas de acordo com uma determinada gramática. As mensagens são símbolos que podem representar chaves, valores aleatórios, textos, ou estruturas mais complexas construídas a partir de outras mais simples. Ao atacante é dado apenas um poder limitado para manipular estas estruturas, tais como a concatenação, separação ou decifrar uma mensagem cifrada caso possua a chave correcta, o que pode não ser o mais realista [10, 12, 23, 24].

2.5 Ferramentas para análise formal de protocolos criptográficos

Actualmente, existe uma gama considerável de ferramentas que permitem análise formal de protocolos criptográficos, baseadas em diferentes modelos e implementando diferentes sistemas de raciocínio, devendo ser usada aquela que se adapte melhor ao problema. Todas estas ferramentas são relativamente novas pois apenas recentemente se propuseram modelos formais para protocolos de segurança. A razão poderá ter a ver com a falta de modelos formais adequados para comunicações distribuídas num ambiente hostil.

De salientar que a análise a um protocolo de segurança corresponde a uma mera abstracção desse mesmo. Deve-se exigir um processo cauteloso de modo a não introduzirmos "irregularidades" ou omitirmos pontos importantes na avaliação do protocolo de segurança em questão.

[5] a máquina de Turing, ou máquina universal foi concebido pelo matemático Alan Turing. Corresponde a um modelo abstrato de um computador, que apenas se restringe aos seus aspectos lógicos.

Será importante referir que as ferramentas verificam apenas determinadas propriedades de segurança. Como tal, não podemos dizer que "o protocolo é seguro" em termos absolutos, pois estamos apenas a analisar o protocolo para um cenário específico tendo em conta apenas determinadas propriedades de segurança. Assim só podemos dizer que o protocolo é seguro para um determinado cenário.

2.5.1 AVISPA

AVISPA é uma ferramenta que providência um conjunto de aplicações para construção e análise de modelos formais de protocolos de segurança. A ferramenta implementa a sua própria linguagem de alto nível, nomeadamente o HLPSL (*High Level Protocol Specification Language*), para definir e descrever uma abstração de um protocolo de segurança. Também diponibiliza um conjunto de *back-ends* para formalmente validar essas mesmas especificações. Mais concretamente, a ferramenta traduz automaticamente todas as especificações HLSPL num formato intermediário (IF) através do tradutor *hlps12if* e depois um dos quatro *back-ends* pode ser escolhido pela pessoa que está a realizar a prova para analisar a especificação IF gerada. Os quatro *back-ends* são On-the Fly Model Checker (OFMC), SAT based model checker (SATMC), Constraint Logic Attack Searcher (CL-ATSE) e Tree Automata-base Protocol Analyser(TA4SP)[31].

2.5.1.1 Linguagem de especificação

A notação Alibe-Bob é uma das mais conhecidas linguagens de especificação para descrever os protocolos de segurança, talvez por ser muito intuitiva, sucinta, e ao mesmo tempo expressiva. No entanto, não é suficiente para capturar a sequência de eventos necessários na especificação de protocolos de larga escala da internet. Deste modo, surgiu o HLSPL, a linguagem de especificação do AVISPA. Trata-se de uma linguagem de especificação de alto nível adoptada pelo projecto AVISPA, e cuja versão original foi baseada na notação Alice-bob, resolvendo algumas das limitações da notação tornando-a com características que a tornam adequada para especificar protocolos modernos e de uma escala industrial. Foi desenvolvida com o objectivo de tornar todo o processo de especificação mais expressivo, rápido, fácil e o mais acessível, mesmo a iniciantes em métodos formais. Devido à sua característica expressiva, modular e baseada em regras, torna a especificação de padrões de controle de fluxo, estruturas de dados, propriedades de segurança complexas, bem como diferentes primitivas criptográficas (nonces,hashes e assinaturas) e propriedades algébricas (XOR,exp), um processo mais simples. No entanto, de salientar que apesar do HLSPL suportar a especificação de um número padrão de primitivas e criptográficas e propriedades algébricas nem todos os *back-ends* suportam todas ou algumas propriedades algébricas [17, 34].

2.5.1.2 Formato intermédio (IF)

Corresponde ao resultado de uma especificação HLPSL quando sujeito a um tradutor nomeadamente o *hlps12if* que implementa um conjunto de regras de re-escrita. As especificações IF definem um conjunto de transições com um estado inicial, regras de transição que ditam as mudanças de estados ou comportamentos e um predicado que define se um determinado estado é ou não um estado de ataque.

2.5.1.3 *Back-ends* do AVISPA

As especificações IF são analisadas por um dos quatro *back-ends* que implementam técnicas de análise diferentes , nomeadamente:

- On the Fly Model Checker (OFMC) - permite a análise e verificação de protocolos usando técnicas simbólicas (por exemplo *lazy intruder*[6]) e baseadas em restrição no número limitado de sessões através da exploração do sistema de transições descrito por uma especificação IF. Usa actualmente um algoritmo de procura profundidade. Este *back-end* suporta a análise de propriedades algébricas de operadores criptográficos e modelos de protocolo tipados e não tipados [9, 34, 8, 32, 7].

- Constraint-Logic-based Attack Searcher (CL-AtSe) - permite uma análise formal através de uma abordagem baseada em restrições com o objectivo de realizar a verificação formal. O algoritmo de procura pode ser, quer em largura quer em profundidade, dependendo da opção escolhida. Esta técnica é bastante apelativa, pois conta com várias optimizações, nomeadamente, com algumas heurísticas de simplificação poderosa e técnicas de eliminação de redundância, reduzindo o espaço de procura sem excluir possíveis ataques. Tudo isto torna todo o processo mais rápido, pois elimina passos redundantes. Além disso, o CL-AtSe é construído de uma forma modular, sendo possível adicionar extensões para manipulação de propriedades algébricas de operadores de criptografia. Também suporta a deteção de ataques por falhas de tipos e lida com a associatividade de mensagens concatenadas[34, 8, 32, 7].

- SAT-based Model-Checker (SATMC) - Este back-end destina-se a analisar modelos de protocolos tipados e permite a verificação de protocolos de segurança para um número limitado de sessões. Este back-end constrói uma fórmula proposicional que representa o protocolo e utiliza um *solver SAT*. Actualmente, o SATMC não suporta equações algébricas. Deste modo, operadores que suportam de algumas equações ou propriedades algébricas, como por exemplo exp ou xor, não permitem que o back-end efectue uma análise devido a não ser conclusiva[8, 32, 7].

- Tree Automata based on Automatic Approximations for the Analysis of Security Protocols (TA4SP) - permite a verificação de protocolos sobre um conjunto ilimitado de

[6] A ideia base desta técnica é evitar a enumeração de possibilidades desnecessárias.

sessões, utilizando regras de re-escrita e linguagens árvore regular. Esta abordagem tem sido testada com sucesso em propriedades de segredo, em que para essas propriedades o TA4SP pode mostrar se um protocolo tem falhas ou se é seguro para qualquer número de sessões [34, 8, 32, 7].

2.5.1.4 Canais de comunicação e linguagem HLSPL

A ferramenta AVISPA não se restringe apenas ao modelo de intruso *Dolev-Yao* como maior parte das ferramentas em alto nível, mas também ao modelo *Over-the-air* (OTA). Sucintamente, no modelo de intruso Dolev-Yao (DY), o atacante pode capturar e (re)enviar dados, decrifrar e cifrar se for conhecedor da chave correcta (funciona como uma caixa-preta de criptografia perfeita), compor novas mensagens, usar informação pública e gerar novos dados ainda não usados. O adversário Dolev-Yao é uma abstração bastante útil, não sendo necessário preocuparmo-nos com qual o cripto-sistema usado. No entanto, de certo modo, acaba por ser restritivo, pois o facto deste usar um criptossistema abstracto faz com que não seja possível capturar interações entre o protocolo criptográfico e o cripto-sistema [15].

OTA é outra técnica de modelagem mais fraca segundo um modelo de protocolos cujo canal de comunicção é sem fio. Nesta técnica apesar de ser possível enviar mensagens, o intruso não tem determinadas habilidades, não podendo evitar que as mensagens originais cheguem ao destino[28].

2.5.1.5 Propriedades de segurança no Avispa

AVISPA consegue apenas verificar um número limitado de objectivos de segurança, no entanto, é suficiente para a grande maioria de protocolos existentes. Para isto o AVISPA disponibiliza um conjunto de factos (goal secret,witness e request) que correspondem meramente a eventos no traço de um modelo de protocolo para ser possível posteriormente verificar algumas propriedades de segurança que são as seguintes [27, 32]:

- segredo - o atacante não consegue determinar o segredo.
  Para definir que um determinado segredo x se mantém secreto podemos recorrer ao goal **secret**(x,id,{A,B}) a cujo primeiro parâmetro corresponde a variável que é secreta, o segundo parâmetro que serve apenas para distingir os diferentes *goals* da propriedade segredo e o terceiro parâmetro que corresponde ao conjunto de agentes conhecedores do segredo.

- autenticação - acto de confirmar algo como autêntico.
  O hlpsl implementa alguns goals de forma a auxiliar a verificação de uma autenticação, nomeadamente:

  – **witness**(A,B,id,E): para uma propriedade de autenticação de uma propriedade de A por B em E, declarando que o agente A é testemunha da informação E. O id do facto corresponde a uma mera identificação para sabermos qual o goal em causa;

– **request**(B,A,id,E): para uma propriedade de autenticação forte[7] de A por B, declarando que o agente B solicita uma verificação do valor E. Tal como no outro o id serve meramente como identificação do goal.

– **wrequest**(B,A,id,E): análogo ao facto anterior, no entanto para uma propriedade de autenticação fraca.

De salientar que o evento *(w)request* deve ser precedido pelo evento respectivo *witness*.

Com o AVISPA, também podemos fazer uma análise de forma a descobrir se existem ataques por falhas de tipos, mas apenas é possível com os back-ends que suportam análise de tipos, activando esta opção especificando ***–type_model=yes***.

A versão actual do AVISPA, não permite suporte a repetição de sessões, no entanto o back-end OFMC tem uma boa aproximação permitindo detectar alguns ataques por repetição. A opção deve ser activada através de **–sessco**.

O AVISPA apesar de procurar falhas em poucas propriedades de segurança dá-nos garantias sobre esses mesmos resultados.

2.5.1.6 Caso de estudo

Nesta secção iremos explicar a especificação do protocolo de segurança descrito na secção 2.3 e verificar se algumas propriedades de segurança são cumpridas. Iremos definir duas regras básicas para descrever o comportamento dos agentes do protocolo (A e B). Refira-se que o carácter '%' no HLPSL marca o início dos comentários (até ao final da linha).

- O conjunto de ações e regras, do agente A é descrito pela seguinte regra básica:

```
role alice( % conjunto de parâmetros conhecidos previamente
           A,B: agent, % agentes do protocolo
            K: symmetric_key, % chave simétrica dos agentes
           SND,RCV : channel(dy), % canal do tipo DY usado para os agentes trocarem mensagens
           Hash : hash_func)  % função hash
 % esta sequência de regras pertence ao agente A
 played_by A def=
        local    % Variáveis locais
                State: nat, % estado de transição
                Na, Nb : text, % tipo text - usado para nonces
                K1: message % message- usado para tipo genérico

        % estado inicial
        init State := 0
        transition
                 % o estado tem de ser 0 e deve receber start para começar o protocolo
                1. State = 0 /\ RCV(start) =|>
                 % o estado passa a ser 2 e cria o nonce Na
                   State':= 2 /\ Na':= new()
                 % envia o novo nonce criado cifrado com a chave simétrica
                   /\ SND({Na'}_K)
                   % o agente A define que quer ser o par de B,
```

[7]Em que A tem prova que o outro interveniente do protocolo é B e, vice-versa, B tem prova que o outro interveniente é A.

```
                    %concordando com o valor de Na
                     /\  witness(A,B,auth_key_Na,Na')
                   % recebe o nonce Nb cifrado com a sua chave simétrica
                 2. State = 2 /\ RCV({Nb'}_K)) =|>
                     % o estado passa a ser 4 e calcula hash da concatenação de Na e Nb (K1)
                    State':= 4 /\ K1':= Hash(Na.Nb')
                    % envia o Nb cifrado pela nova chave de sessão, nomeadamente K1
                    /\ SND({Nb'}_K1')
                    % o agente A define que quer ser o par de B,
                    %concordando com o valor de Nb
                     /\ witness(A,B,auth_key,Nb')
end role
```

HLPSL é uma linguagem baseada em regras e cujas especificações de cada participante são descritas em diferentes módulos, cada uma designada por regra básica. Cada regra básica vai conter informação relativa ao conhecimento inicial e ao comportamento de cada agente, devendo conter assim uma lista de parâmetros que descreva o conhecimento inicial e qual o agente a que pertence tal sequência de ações e conhecimento. O comportamento de cada agente pode ser descrito com a ajuda de transições que nos permitem descrever a sequência de ações do agente. Perante um determinado conjunto de pré-condições ocorre um determinado conjunto de eventos. Cada transição tem um lado esquerdo que descreve o que deve ser verdadeiro para a transição ocorrer e um lado direito que define as consequências dessa mesma transição.

Na primeira transição, o agente A recebe um "start" (definido no HLPSL) que corresponde a um sinal para A poder iniciar o protocolo. Depois de receber tal sinal, o agente A pode criar um Nonce(devendo ser do tipo *text*) e enviar ao agente B. De salientar que todos os valores novos devem ser sinalizados com uma plica (e.g. X' e não X), caso contrário não são inicializados com o novo valor. Na segunda transição, o agente A recebe um nonce não conhecido anteriormente, daí o Nb', e depois envia esse mesmo valor cifrado com a chave de sessão, que corresponde ao hash da concatenação de Na e Nb. De salientar ainda que na segunda parte da transição temos de usar Nb' e não Nb. Isto deve-se ao facto de o valor de Nb só ser actualizado depois de ser verificado que tal transição é válida.

- A regra básica do agente B é a seguinte:

```
role bob(
        % conjunto de estado
         A, B: agent,
          K: symmetric_key,
         SND,RCV: channel(dy),
         Hash : hash_func)
 % esta sequência de regras pertence ao agente B
  played_by B def=
        local % variáveis locais
                State: nat,
                Na, Nb : text,
                  K1: message

        init State := 1
        transition
```

```
       1. State = 1 /\ RCV({Na'}_K)) =|>
          State' := 3 /\ Nb':= new()
          /\ SND({Nb'}_K)
          /\ K1':= Hash(Na'.Nb')
            %  a chave de sessão só pode ser conhecida por A e B
           /\ secret(K1',k1,{A,B})
            % o agente B aceita o valor de Na e baseia-se na garantia que A existe e concorda
            % com esse valor
           /\ request(B,A,auth_key_Na,Na')
                2. State = 3 /\ RCV({Nb}_K) =|>
            % o agente B aceita o valor de Nb e baseia-se na garantia que A existe e concorda
            % com esse valor
                State':=5 /\ request(B,A,auth_key,Nb)
end role
```

A observar que definimos nas útimas regras básicas os *goals* de autenticação *witness* e *request*, pois são dois factos relacionados com a autenticação forte. Estes são necessários para futuramente verificarmos a autenticação do valores, ou seja, conseguirmos verificar que este valor foi calculado pelo agente B e que é o mesmo valor para os dois agentes do protocolo. Além disso, definimos o facto *secret* que especifica que a chave de sessão apenas pode ser conhecida pelo agente A e B.

- A interação entre os agentes pode ser definida pela seguinte regra, que corresponde à composição da regra do agente A (regra básica alice) e agente B (regra básica bob).

```
role session(
            % parâmetros iniciais
             A,B: agent,
             K : symmetric_key,
             Hash: hash_func)
     def=
       local SA,RA,SB,RB : channel(dy)

      composition
             alice(A,B,K,SA,RA,Hash)
             /\ bob(A,B,K,SB,RB,Hash)
end role
```

O operador $/\backslash$ indica que as regras podem ser executadas em paralelo.

- Por fim temos a regra de topo. Tipicamente a regra de topo contém o nome de todos os agentes, chaves públicas ou qualquer outra chaves, funções conhecidas por mais do que um agente e o conhecimento do intruso de forma a descrever o ambiente e propriedades globais do protocolo.

  Neste exemplo, a nossa regra de topo é definida da seguinte maneira:

```
role environment() def=
        const
                 % agentes e intruso
                a , b , i : agent,
                % chaves simétricas
                kai, kbi, kab : symmetric_key,
                % identificação para os goals
                auth_key_Nb,auth_key_Na, k1 : protocol_id,
```

```
                % função hash
                h : hash_func

        % conhecimento do intruso
        intruder_knowledge = {a,b,i,kai,kbi,h}

        composition
                %  possíveis sessões
                session(a,b,kab,h)
                /\ session(a,i,kai,h)
                /\ session(i,b,kib,h)
end role
```

- Agora, podemos definir quais as propriedades de segurança a serem verificadas:

```
goal
         % autenticidade de Nb
        authentication_on auth_key_Nb
        % autenticidade de Na
        authentication_on auth_key_Na
        % confidencialidade
        secrecy_of k1
end goal
```

  Sendo os “goals” meros eventos no traço, deve ser verificado se essa "ordem"de eventos/ ações é respeitada.

- A declaração final da especificação corresponde à “environment()”, sendo a instânciação da regra de topo.

Por fim convertemos o HLPSL na linguagem intermediária If e escolhemos um dos back-ends que suportem todas as “funcionalidades” usadas. Neste caso, qualquer um dos back-ends, pode ser usado.

Não foi encontrado nenhum ataque usando todos os back-ends. As propriedades de segurança especificadas foram verificadas. A chave de sessão criada é apenas conhecida pelos agentes e o os valor Na e Nb são definidos pelos mesmos agentes A e B. Isto deve-se ao facto, de os agentes conhecerem previamente a chave simétrica do agente, garantindo autenticidade do segredo e dos agentes pois apenas o possuidor da chave consegue cifrar/decifrar a mensagem.

2.5.2 ProVerif

ProVerif é uma ferramenta automática de verificação de protocolos criptográficos baseada no modelo formal, tendo suporte para um conjunto de primitivas criptográficas tal como: criptografia simétrica e assimétrica; assinaturas digitais; funções de *hash*; provas não interativas de conhecimento zero; entre outras. O ProVerif disponibiliza certas primitivas criptográficas, providenciando equações quando essas requerem relações algébricas entre termos. No entanto, não suporta todas as equações, tendo algumas limitações no concerne a lidar com propriedades algébricas, como por exemplo, o operador XOR. Todavia, no caso de ser necessário trabalhar

sobre algumas propriedades como XOR e do Diffie-Hellman (DH), o ProVerif suporta a inclusão de outras ferramentas, neste caso, o XOR-ProVerif e DH-ProVerif, que resolvem algumas das limitações referidas [14, 7].

2.5.2.1 Linguagem de especificação

Os protocolos a analisar podem ser codificados num formato baseado em cláusulas de Horn, ou numa linguagem de processos (uma extensão ao π *calculus*, designada por *spi-calculus*). No segundo formato de input, os processos são automaticamente traduzidos num conjunto de cláusulas de Horn, sendo essa a representação sobre a qual é realizada a verificação. De qualquer forma, neste documento centramo-nos no formato baseado no π *calculus*.

A notação *spi-calculus* serve para descrever computações concorrentes cuja configuração pode mudar durante uma computação. Os processos *spi-calculus* neste contexto representam os participantes (agentes) que trocam mensagens especificadas no protocolo criptográfico e podem ser executados em paralelo [14].

2.5.2.2 Canais de comunicação

O canal de comunicação é definido unicamente segundo propriedades do modelo de intruso "Dolev-Yao". A criptografia usada é assumida como perfeita e o atacante assume o papel de um "adversário activo" . Mais precisamente, ele pode interceptar e ler qualquer mensagem que é enviada na rede e enviar outra mensagem que contenha informações que soube através de ações anteriores. No entanto, o atacante está limitado a aplicar as operações criptográficas especificas pelo utilizador, não podendo aplicar qualquer outro algoritmo de tempo polinomial [11, 15]. Acontece que, devido à diversidade de protocolos actuais, pode acontecer que o modelo de intruso "Dolev-Yao" não seja o mais indicado.

2.5.2.3 Propriedades de segurança no ProVerif

O ProVerif consegue verificar as seguintes propriedades de segurança [21, 14]:

- segredo - adversário não consegue obter o conhecimento sobre determinada informação. A notação **query attacker(segredo).** é utilizada para verificar se um determinado segredo não pode ser comprometido. O ProVerif tenta provar que o estado "segredo" é desconhecido para o atacante, devendo ser inalcançável a esse mesmo. Assim, **not attacker(segredo)** é verdade apenas se esse segredo não é derivável para o adversário, dando um resultado **RESULT not attacker:(segredo[]) is true**.

- Algumas propriedades de correspondência[8] e autenticação.
  Na especificação de um protocolo podemos ter vários estágios que devem ser alcançados para se atingir um determinado objectivo. No ProVerif podemos recorrer a sintaxe ***event*** *e(T1,..Tn)* que declara *e* como um evento com argumentos com tipos T1...Tn. A

[8] impõe uma relação de causalidade sobre determinados eventos.

verificação é feita tendo em consideração a sequência de eventos que ocorre num traço: a propriedade de correspondência estabelece que se um evento tiver sido executado, então determinados outros eventos devem já ter sido executados. A correspondência ***query*** *x:t1,y:t2,z:t3;* ***event****(e(x, y))* ***==> event****(e'(y, z))*, verifica se o *evento e'* acontece antes do *evento e* e a correspondência injectiva ***query*** *x1:t1,... ,xn:tn;* ***inj-event****(e(M1,... ,Mj))* ***==> inj-event****(e'(N1,...,Nk))* verifica se o *evento e* é sempre precedido pelo *evento e'* e se todo o traço contém pelo menos tantos *eventos e* como *eventos e'*.

- equivalências entre processos que diferem apenas de termos.
  A equivalência observacional de processos é um conceito poderoso que nos permite efectuar provas sobre propriedades que não podem ser expressas como propriedades de acessibilidade ou correspondência. Dois processos P e Q são observacionalmente equivalentes ($P \approx Q$), quando um adversário não consegue distinguir P de Q. No entanto, a verificação de equivalência é muitas vezes demasiado complexa, pelo que o ProVerif nesses casos não fornece respostas conclusivas. Deste modo, o ProVerif apenas consegue provar algumas equivalências observacionais. Em rigor, internamente o ProVerif prova uma propriedade muito mais forte que a equivalência observacional de P e Q, pois o passo de redução deve ser executado da mesma maneira em P e Q. Caso o ProVerif não consiga provar uma equivalência observacional então tenta reconstruir um ataque. Este possível ataque explica porque a prova falha e pode ajudar a pessoa que está a verificar a prova a perceber o porquê da equivalência observacional não ser assegurada. No entanto, não prova que a equivalência observacional não é assegurada. Por isso mesmo, o ProVerif para equivalências observacionais nunca conclui como sendo falso (RESULT [Query] is false), mas sim que tal prova falha e que nada se pode concluir (RESULT [Query] cannot be proved).

- segredo forte - o intruso não consegue distinguir o segredo de um valor aleatório
  Esta propriedade de segurança pode ser provada recorrendo ao conceito de equivalência observacional, sendo bastante importante para capturar a capacidade do atacante para apreender dados parciais do segredo. Esta noção é bastante importante se estivermos a trabalhar sobre um conjunto determinado de soluções. Por exemplo, um processo P que usa um booleano, só existem dois valores possíveis (verdadeiro ou falso). No entanto, pode torna-se interessante averiguar se o atacante consegue distinguir esses dois valores, devendo existir uma equivalência observacional ($P\{true/b\} \approx P\{falsa/b\}$) de forma a que o atacante não possa determinar se é verdadeiro ou falso.

  Em ProVerif o segredo forte para valores X1... Xn é dada por **noninterf X1,...,Xn** ou então através da sintaxe **choice**, tipicamente muito menos eficiente. No entanto, na presença de equações algébricas, **noninterf** geralmente leva a ataques falsos, devendo assim **noninterf** b **among** (true, false) ser codificada como **let** b = **choice** [true, false] **in P**, onde P corresponde ao processo em causa. No entanto, se na especificação aparecer **choice** não pode aparecer **query**, **noninterf** ou **weaksecret**. Por esta razão, o ProVerif pode obrigar a ser necessário criar diferentes especificações a fim de provar várias propriedades do

mesmo protocolo.

A ferramenta ProVerif também tem a capacidade de detectar ataques por falhas de tipo. A linguagem *spi-calculus* é fortemente tipada, sendo possível à pessoa que está a realizar a prova definir os tipos dos dados mais adequados. Por omissão, o ProVerif tem definida a opção **set ignoreTypes = true.** ou **set ignoreTypes = all.** o que significa que os tipos são ignorados. Assim, todas as funções relativas à conversão de tipos, nomeadamente **fun tc(t) : t'[data,typeConverter]**, completamente ignoradas. A verificação de tipos pode ser, por vezes, importante para averiguar se os ataques detectados são justificados por falhas nos tipos (type flaw attack), devendo a opção respectiva ser activada nessas situações (**set ignoreTypes = attacker** ou **set ignoreTypes = false**). Um ataque de falha de tipos manipula os dados brutos de um protocolo de comunicação para causar uma má interpretação de dados ao nível de mensagem. Assim, com esta opção activa, não é possível atribuir, por exemplo, um *bitstring* para um *nonce*, pois representam tipos completamente diferentes.

No caso da ferramenta não conseguir provar propriedades de segurança em questão, tenta reconstruir possíveis ataques, isto é, determinar um traço de execução do protocolo que contrarie as propriedades respectivas. As provas por traço correspondem a um ataque real, contrariamente às derivações que devido às aproximações internas realizadas pelo ProVerif podem não corresponder a um verdadeiro ataque. Uma derivação consiste na mera representação interna de como o atacante pode quebrar a propriedade desejada.

ProVerif é uma ferramenta "correcta", mas "não completa". Quando a ferramenta afirma que uma propriedade é satisfeita, então o modelo realmente garante essa propriedade. No entanto, pode não ser capaz de provar uma propriedade de segurança mesmo que ela efectivamente se verifique. Além disso, o ProVerif pode abstrair detalhes importantes da criptografia, e, portanto, não é ,na generalidade, tão correcto como uma ferramenta baseada no Modelo Computacional da criptografia. No caso de não querermos abstrair certos detalhes, uma ferramenta de verificação baseado no modelo de segurança computacional, tal como a ferramenta CryptoVerif, pode ser usada.

2.5.2.4 Caso de estudo

Nesta secção iremos explicar a especificação do protocolo de segurança descrito na secção 2.3 e verificar se algumas propriedades de segurança são cumpridas. De salientar que caracteres dentro de '(*' e '*)' correspondem a um comentário na especificação *spi calculus* do ProVerif.

- Antes de especificar qualquer comportamento dos agentes, em que nesta especificação correspondem a processos, torna-se necessário descrever quais os operadores de mensagens e operações criptográficas utilizadas, assinalando para o efeito a respectiva assinatura de tipos.

```
(*Definir o canal por onde os agentes comunicam*)
free c : channel.
(*definir o tipo nonce*)
type nonce.
```

```
(*Criptografia de chave simtrética*)
(*definir o tipo key*)
type key.

fun senc(nonce,key): bitstring.
reduc forall x: nonce , y : key; sdec(senc(x,y),y) = x.

(*função hash*)
fun hash(nonce,nonce): bitstring.

(* função de conversão de tipos*)
fun b2key(bitstring): key  [data,typeConverter].
```

Neste exemplo, usamos regras de rescrita (destructores) em vez de equações, porque não precisamos de definir operações algébricas entre termos e porque são muito mais eficientes. De salientar que fazemos conversão de tipos quando necessário, pois o ProVerif é fortemente tipado.

- Depois disto já podemos definir quais as propriedades que queremos verificar. Neste exemplo, vamos averiguar se a chave de sessão é secreta e se a autenticação do segredo é verificada. Para isso definimos as seguintes "queries" que nos ajudam a verificar a ordem correcta de eventos e a determinar eventos "impossíveis" nos traços do protocolo.

```
(* Queries - segredo *)
free secretK : nonce[private].
query attacker(secretK).

(* Queries - autenticação *)
event beginparam(nonce).
event endparam(nonce).

query x:nonce; event(endparam(x)) ==> event(beginparam(x)).
```

- Agora, já temos a capacidade para definir o comportamento de agentes. O agente A e B podem ser definidos através dos seguintes Processos A e B respectivamente.

```
let processoA(chave: key) =
    (*iniciar o protocolo*)
    in(c,());
    (*criar Na do tipo nonce*)
    new Na:nonce;
    (*envio do nonce Na*)
    event beginparam(Na);
    out(c,senc(Na,chave));
    in(c,Nbc:bitstring);
    let (Nb:nonce)= sdec(Nbc,chave) in
    let (h:bitstring)=hash(Na,Nb) in
    let (k:key) = b2key(h) in
    (* envio do nonce Nb*)
    event beginparam(Nb);
    out(c,senc(Nb,k)).

let processoB(chave:key) =
    in(c,Nac:bitstring);
    let (Na:nonce)= sdec(Nac,chave) in
    new Nb:nonce;
    out(c,senc(Nb,chave));
```

```
in(c,Mc:bitstring);
let (h:bitstring)=hash(Na,Nb) in
let (k:key) = b2key(h) in
let (Nbn:nonce)=sdec(Mc,k) in
if (Nbn=Nb) then
  (*Na e Nb devem ser definidos pelos agentes*)
   event endparam(Na);
   event endparam(Nb);
   (*a chave de sessão deve ser secreta*)
   out(c,senc(secretK,k)).
```

Na descrição destes processos, a primitiva "in" denota a leitura de uma mensagem (por um canal determinado); "out" a escrita; e "new" a geração de um valor aleatório. Note ainda a declaração de eventos nas fases apropriadas dos processos.

- Por fim, definimos um processo que agrega os anteriores, estabelecendo a forma como esses processos interagem.

```
process
        new chave:key;
        out(c,());
        ((!processoA(chave))|(!processoB(chave)))
```

Note que o sistema permite consistir de múltiplas execuções simultâneas do protocolo, não limitando o número de protocolos concorrentes. Por esse motivo utiliza-se o operador de replicação "!".

Após a execução da script apresentada pelo ProVerif, e tal como no caso do AVISPA, nenhum ataque é encontrado. Pode-se então também concluir que após a execução do protocolo apenas o agente A e B ficam conhecedores da chave simétrica estabelecida.

2.5.3 CryptoVerif

CriptoVerif é uma ferramenta automática baseada no modelo computacional que implementa um mecanismo genérico para especificar assumpções de segurança de primitivas criptográficas, como criptografia simétrica, autenticador de mensagem (mac), criptografia de chave pública, assinaturas e funções hash. A ferramenta produz provas válidas para um determinado número de sessões polinomial no parâmetro de segurança, na presença de um adversário activo[9]. Nesta aproximação do modelo computacional, trabalhamos no baixo nível, em que as mensagens são bitstrings e as primitivas criptográficas são funções de bitstrings para bitstrings.

Trata-se de uma ferramenta indicada para validar um protocolo, pois a ferramenta tenta todas as maneiras possíveis para validar um protocolo. Além disso, sendo uma ferramenta baseada no modelo computacional, as provas feitas nesta mesma ferramenta são mais descritivas e mais próximas das adoptadas pela comunidade científica da área[12].

[9]O adversário pode interceptar todas as mensagens enviadas na rede e enviar/computar mensagens na rede.

2.5.3.1 Linguagem de especificação e mecanismo de raciocínio

Os protocolos de segurança podem ser especificados por equivalência observacional[10]. Blanchtet desenvolveu um *process calculus*, inspirado principalmente no π *calculus*, em que possui uma semântica probabilística e cujos processos executam em tempo polinomial (em função do parâmetro de segurança). As provas são apresentadas segundo uma sequência de jogos cujo objectivo é transformar o jogo inicial de forma incremental num jogo final cujo requisito de segurança deverá ser trivialmente satisfeito. O jogo inicial corresponderá ao critério de segurança sobre protocolo a ser provado e a diferença de probabilidades em dois jogos consecutivos deve ser negligenciável. Dessa forma, conclui-se que a diferença de probabilidades entre o primeiro e último jogo é também negligenciável, estabelecendo-se assim a segurança da técnica. Esses jogos são representados através de uma aproximação para modelar formalmente sistemas concorrentes nomeadamente o *process calculi* que é inspirado no π *calculus*[12, 30]. Também pode recorrer à lógica modal para derivar e determinar conclusões sobre o protocolo. Além disso, o CryptoVerif pode avaliar a probabilidade de sucesso de um ataque contra o protocolo como uma função da probabilidade de quebrar cada primitiva criptográfica e para um número definido de sessões (segurança exacta).

2.5.3.2 Características básicas

O CriptoVerif é uma excelente ferramenta que permite derivar provas rigorosas de segurança, disponibilizando um conjunto de funcionalidades que nos permitem guiar o processo de descoberta da prova. Algumas das construções sintácticas básicas da linguagem são as seguintes[13]:

- **Tipos:** correspondem a um conjunto de bitstrings em que podem ser :
  - **fixed**- o conjunto de bitstrings tem o mesmo tamanho ;
  - **bounded**- conjunto de bitstrings com tamanho limitado;
  - **large**- significa que o tipo é suficientemente grande para que a probabilidade de colisões entre escolhas aleatórias nesse tipo possa ser ignorada.

  Exemplo: **type T [fixed,large].**
- **Funções** que podem ser declaradas como:
  - **commut** a função é comutativa, isto é, f(x,y)= f(y,x);
  - **compos** a função é injectiva e todos os seus inversos podem ser computados em tempo polinomial;
  - **uniform** a função mapeia uma distribuição uniforme numa distribuição uniforme. Exemplo: **fun mult(A,A): A [commut].**

[10]Q é observacionalmente equivalente a Q', quando o adversário tem uma probabilidade negligenciável de distinguir Q de Q'

- **Probabilidades** que podem ser usadas como argumentos de funções.
  Exemplo: **proba p.**

- **Equivalências:** A equivalências são essenciais para a sequência de jogos de uma prova. O CriptoVerif tenta usar todas as equivalências com o objectivo de reduzir um problema num mais simples. **equiv L <=(p)=>R.** significa que a probabilidade para uma uma máquina de Turing probabilistica de distinguir L de R é p. L e R definem conjuntos de funções.

Além disso, também tem um número padrão de primitivas criptográficas predefinidas.

2.5.3.3 Propriedades de segurança no CryptoVerif

O CriptoVerif consegue provar algumas propriedades, nomeadamente[13] :

- **segredo** em que o intruso não pode conseguir distinguir o segredo de um conjunto de números aleatórios independentes através de várias "queries" de teste. A *query* utilizada para efectuar tal verificação é dada por **query secret x**, em que x corresponde ao segredo.

- **Segredo de uma-sessão** em que o intruso não pode conseguir distinguir o segredo de um conjunto de números aleatórios independentes através de uma única "query" de teste. A *query* utilizada para efectuar tal verificação é dada por **query secret1 x**, em que x corresponde ao segredo.

- **correspondências e autenticação** que podem ser verificadas através da confirmação de uma sequência de eventos. De forma a mostrar que o *evento e* é sempre precedido pelo *evento e'*, deve ser verificado **query event e ==> e'.** As correspondências funcionam como em linguagem simbólicas. Estas definem que se um evento foi executado então outros eventos devem ter sido executados.

Além disso, também tem um número padrão de primitivas criptográficas predefinidas. De salientar, que a linguagem de especificação não é fortemente tipada, os tipos correspondem a um mero conjunto de bitstrings. Assim, não é possível verificar ataques de falhas de tipos.

O CryptoVerif apresenta-se uma ferramenta precisa e correcta para validar propriedades de seguraça. No entanto, não é completa. Deste modo, o facto de não conseguir arranjar uma prova não obriga a que a propriedade seja refutável.

2.5.3.4 Caso de estudo

- De forma a modelar o protocolo apresentado na secção 2.3 recorremos a definição de algumas primitivas criptográficas . Consideramos duas assumpções criptográficas: assumimos que que a criptografia da chave simétrica é indistinguível sobre IND-CPA[13] (indistinguishable under chosen plaintext attacks) e que a função de hash é modelada segundo o *random-oracle model* (i.e. é modelada com uma função aleatória pura).

```
(* Tipos de canais*)
channel c0, c1, c2, start, cc.
(*outros tipos*)
type nonce [large,fixed].
type key [bounded,fixed].

(*criptografia de chave simétrica*)
type keyseed [fixed].
type seed [large,fixed].
proba Penc.
proba Penc1.

(*primitiva criptográfica pre-definida para chave inicial*)
expand IND_CPA_sym_enc(keyseed,key,nonce,bitstring,seed,kgen,enc,dec,injbot,Z,Penc).

type host [bounded].
type maxenc [bounded].
(*primitiva criptográfica pre-definida para chave de sessão*)
expand IND_CPA_sym_enc(keyseed,key,nonce,bitstring,seed,kgenn,encn,decn,injbotn,Z2,Penc1).

(*Função de hash*)
type hashkey [fixed].
type D [fixed].
param qH [noninteractive].
expand ROM_hash(hashkey,nonce,key, hash).

channel hc1, hc2.
let hashoracle = ! qH in(hc1, x: nonce); out(hc2, hash(hk,x)).

(* Funções auxiliares- concatenação *)
const Z2concat:nonce.
fun concat(nonce, nonce): nonce [compos].

fun bitn(bitstringbot): nonce [compos].
fun nkey(nonce): key [compos].

forall y:nonce,z:nonce;
        Z2(concat(y,z)) = Z2concat.


(* Query's*)
event endparam(nonce).
event beginparam(nonce).

query x:nonce; event endparam(x) ==>  beginparam(x).
query x:nonce; event endparam(x) ==>  true.


(*b é indistinguível de um array de números random independentes*)
query secret K.

let processA =
    in(c0,());
    new Na: nonce;
    new s : seed;
    event beginparam(Na);
    out(c1,enc(Na,chave,s));
    in(c1,m:bitstring);
    let (Nbbt:bitstringbot) = dec(m,chave) in
    let (Nb1:nonce)=bitn(Nbbt) in
    let kn = concat(Na,Nb1) in
    let (k':key) = nkey(kn) in
    new se : seed;
```

```
    event endparam(Na).

let processB =
        in(c1,m:bitstring);
        let (Nabt:bitstringbot) = dec(m,chave) in
        let (Na: nonce) = bitn(Nabt) in
        new Nb: nonce;
        new s:seed;
        event beginparam(Nb);
        out(c1,enc(Nb,chave,s));
        in(c2,msg:bitstring);
        let kn = concat(Na,Nb) in
    let (k1':key) = nkey(kn) in
    new se : seed;
        let (Nbbt:bitstringbot) = decn(msg,k1') in
        let (Nb2: nonce) = bitn(Nbbt) in
                if Nb=Nb2 then
                   event endparam(Nb);
                   let K:key=k1'.

process
        (in(start, ());
        new hk: hashkey;
        new gk:keyseed;
        let chave:key = kgen(gk) in
    out(cc,());
             (processA |processB | hashoracle))
```

A linguagem de descrição de processos é muito próxima com a do ProVerif, pelo que dispensa uma apresentação detalhada. De referir apenas que agora o operador de replicação dispõe de um parâmetro (limitado por um polinómio sobre o parâmetro de segurança) que limita o número de repetições permitidas. A utilização da opção "[noninteractive]" permite definir que a consulta do oráculo (hashoracle) pode ser feita pelo adversário sem ter de interagir obrigatoriamente com o protocolo testado. Notadas as semelhanças entre sintaxe do ProVerif e CryptoVerif, convém salientar que o mecanismo de prova e a prova gerada é totalmente diferente. Além disso, como podemos observar o CryptoVerif permite-nos ser mais "detalhados", sendo possível adicionar detalhes como probabilidade de quebrar a propriedade IND-CPA, tamanhos de chave, entre outros. As restantes ferramentas analisadas não nos permitem um nível de análise tão fina.

As sequências de jogos geradas pelo CryptoVerif vão finalmente definir uma prova formal de segurança do protocolo especificado para as propriedades de segurança descritas anteriormente. As transições entre jogos correspondem a pequenos passos. No caso do exemplo apresentado, a prova foi completamente automática. No entanto, para protocolos (ligeiramente) mais complexos, é por vezes necessário guiar o processo de construção com comandos apropriados, ou mesmo conduzir todo o processo de forma interactiva. Por outro lado, o CryptoVerif está vocacionado para validar protocolos e não para encontrar ataques, contrariamente às restantes ferramentas que são muito efectivas em encontrar ataques em protocolos com falhas. De salientar a diferença de tamanho e complexidade da prova gerada pelo CryptoVerif relativamente as outras ferramentas(uma parte da prova está no anexo).

3 . Sistema Single Sign-On (SSO)

3.1 Introdução

Autenticação é a capacidade de confirmar se uma determinada identidade[1] é quem afirma ser. Trata-se de uma das propriedades de segurança mais importantes, pois grande parte das aplicações existentes necessita de mecanismos de autenticação por forma a fornecer serviços ou informações apenas a identidades digitais[2] específicas. Existem vários métodos de autenticação, de entre eles:

- A autenticação pode ser feita através de algo que o utilizador sabe. Normalmente é feita através de uma credencial de autenticação que consiste na combinação de um nome do utilizador e uma palavra-passe. A segurança depende da dificuldade de descobrir tal palavra-passe, que aumenta com o tamanho e aleatoriedade dessa mesma;
- A autenticação pode ser feita através de algo que o utilizador possui, sendo os dispositivos físicos mecanismos de autenticação (e.g. *smartcards*, como o cartão de cidadão);
- A autenticação pode ser realizada através de características do próprio utilizador, através de atributos específicos e únicos de cada utilizador. Dados biométricos, tais como, impressões digitais, análise de retina e reconhecimento de voz podem ser usados;
- A autenticação também pode ser realizada tendo em conta o lugar em que o utilizador se encontra. Adaptadores de rede, caller-IDs, e sistemas baseados em posicionamento global via satélite, permitem-nos autenticar um utilizador consoante a sua localização.

Antes de mais, torna-se importante descrever, mesmo de uma maneira simplista, o processo de autenticação. Este método consiste de uma formal geral em comparar os dados, nomeadamente, as credenciais de autenticação introduzidas pelo utilizador com os dados guardados no servidor. Só se os dados coincidirem é que a identidade em questão pode aceder à informação e serviços pretendidos. Normalmente, o primeiro método de autenticação apresentado é dos mais utilizados devido à sua facilidade de implementação e não ser muito custoso face aos outros tipos de autenticação existentes. Neste documento focaremos, apenas, o primeiro mecanismo de autenticação.

Actualmente a autenticação tem tido bastante destaque, sendo a propriedade de autenticação uma das funções de segurança mais importantes que uma aplicação deve fornecer de modo a proteger serviços/informações de identidades falsificadas. Nos últimos anos, o número de *providenciadores de serviços (SP)*, ou seja, o número de entidades responsáveis por fornecer serviços ou informação a determinado conjunto de utilizadores, aumentou consideravelmente.

[1] pode ser visto como um conjunto de dados e características, agregadas ou não, relacionados com um sujeito, sendo que este pode não ser necessariamente uma pessoa mas como por exemplo, uma máquina.

[2] forma de representar digitalmente uma identidade.

No entanto, existe um problema associado a esses mesmos SPs, nomeadamente, o uso de arquitecturas de segurança distribuídas que geralmente exigem mecanismos de segurança e, por consequência, necessitam de determinados recursos (chaves, credenciais) e algoritmos de autenticação. Os utilizadores têm para cada SP em que estão registados um conjunto de credenciais de autenticação, aumentando o número de credenciais por utilizador e, assim, tornando cada vez mais difícil a gestão dessas mesmas. A solução mais comum de muitas pessoas é utilizarem as mesmas credenciais de acesso para todos os SP em que estejam registados, reduzindo-os deste modo ao mesmo nível de segurança. O nível de segurança de todo o sistema fica reduzido ao mesmo nível de segurança da parte mais fraca, comprometendo a segurança de todo um sistema. Deste modo, aplicações com necessidades de diferentes políticas de segurança ficam reduzidas à mesma. Outra parte das pessoas acabam por guardar as palavras-passe em lugares que pensam ser "seguros", podendo estar a comprometer a sua segurança. De forma a tentar resolver este problema, surgiu o mecanismo ***Single Sign-On (SSO)***, em que o utilizador apenas se autentica uma vez, ficando automaticamente conectado a todas as aplicações, nomeadamente aos SPs, se assim o desejar. Numa arquitectura SSO o servidor funciona como ponto de autenticação para um domínio definido. Assim, usando o sistema de autenticação global, ou sistema que implementa o mecanismo SSO, os utilizadores não têm de se preocupar em gerir credenciais de autenticação nem de ter o trabalho de se autenticar sempre que desejam aceder a novas aplicações/sistemas, sendo suficiente uma única autenticação para acederem a um determinado conjunto de serviços que implementem o mecanismo SSO. A ideia básica do sistema SSO consiste em liberar outras partes do sistema ou utilizadores das obrigações de autenticação e autorização, ficando estas a cargo do sistema SSO. Deste modo, os utilizadores já não precisam de memorizar uma gama de palavras-passe para acederem a diferentes serviços. O objectivo do SSO é , portanto, minimizar o número de registos e autenticações efectuados pelo utilizador de forma, a aumentar a usuabilidade da rede como um todo [26, 18] e tornar todo o processo de autenticação mais simples, rápido e seguro para o utilizador.

3.2 Taxonomia e arquitectura de um sistema SSO

Ao longo dos últimos anos várias arquitecturas SSO e diferentes abordagens foram desenvolvidas de forma a resolver as necessidades dos utilizadores. Um aspecto comum é que o utilizador se autentica perante o componente SSO (que pode estar no mesmo computador) através de um "dado" mestre que pode ser uma palavra-passe, dado biométrico ou até mesmo um dispositivo físico (Ver figura 3 .1) de forma a ter a autorização para aceder aos SPs. Para ser possível efectuar a autorização de uma determinada identidade é obrigatório efectuar uma autenticação, pois a autorização não pode ser realizada sem autenticação (enquanto que o inverso não é obrigatoriamente verdade). Deste modo, o componente SSO tem que ter acesso a um conjunto de identificadores, que relacionam o utilizador aos SPs, em que está registado. Estes identificadores, designados por **identidades SSO**, podem assumir várias formas e ser guardados num ficheiro ou numa base de dados, no mesmo computador ou noutro(s) computador(es). De

forma a melhorar a mobilidade, essas identidades SSO não devem estar guardadas no mesmo computador do utilizador, mas guardadas num computador acessível ao utilizador. No entanto, no caso das identidades SSO estarem guardadas em mais do que um computador é necessário definir mecanismos de forma a garantir coerência de informação. Além disso, apesar de ser melhor em termos de usuabilidade, outras questões de segurança se levantam. As identidades SSO devem, portanto, ser guardadas e transmitidas de forma segura de modo a não comprometer essas mesmas.

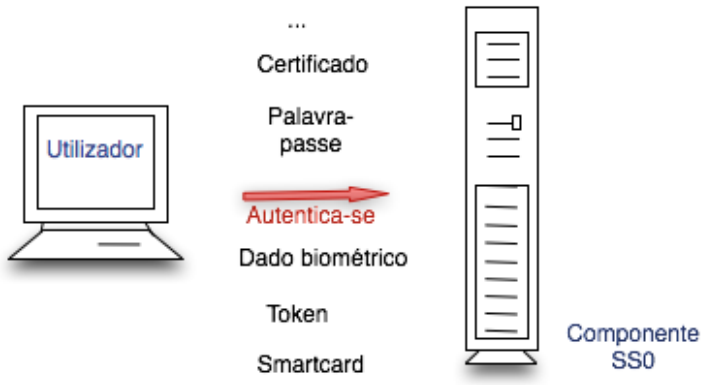

Figura 3.1 Mecanismo de autenticação do utilizador (pessoa ou máquina) ao componente SSO

Consoante as necessidades dos utilizadores podemos optar por diferentes cenários SSO. Existem diferentes classificações, no entanto, vou dividir o sistema SSO em dois tipos: Pseudo-SSO e True-SSO. Na abordagem **Pseudo-SSO**, depois do utilizador se autenticar ao componente SSO, esse mesmo fica responsável, para além da gestão das credenciais de autenticação, por autenticar o utilizador automaticamente aos SPs distintos. No entanto, caso o utilizador (pessoa ou máquina) esteja a aceder ao serviço pela primeira vez, deverá interagir de modo a que as respectivas credenciais possam ser, se desejado, guardadas e associadas ao serviço em questão. Nesta abordagem, todas as vezes que o utilizador se conecta a um SP ocorre um processo de autenticação separado, pois os SPs podem ter a sua própria solução de autenticação e administração. Além disso, uma identidade SSO corresponde exactamente a um SP, podendo haver múltiplas identidades SSO para um único SP (Ver figura 3.2 - de salientar que, nas figuras as identidades SSO correspondem às credenciais de autenticação) [26].

O MobiPassword [5] é um possível exemplo de um programa de gestão de acesso para MS Windows. Ele permite o armazenamento seguro de informações pessoais importantes, como IDs de utilizador e senhas, contas financeiras e PINs. Este armazena todas as suas senhas em um banco de dados e tem a possibilidade de realizar logins automaticamente.

Na abordagem **True-SSO**, os SPs têm necessidade de ter a mesma solução de autenticação

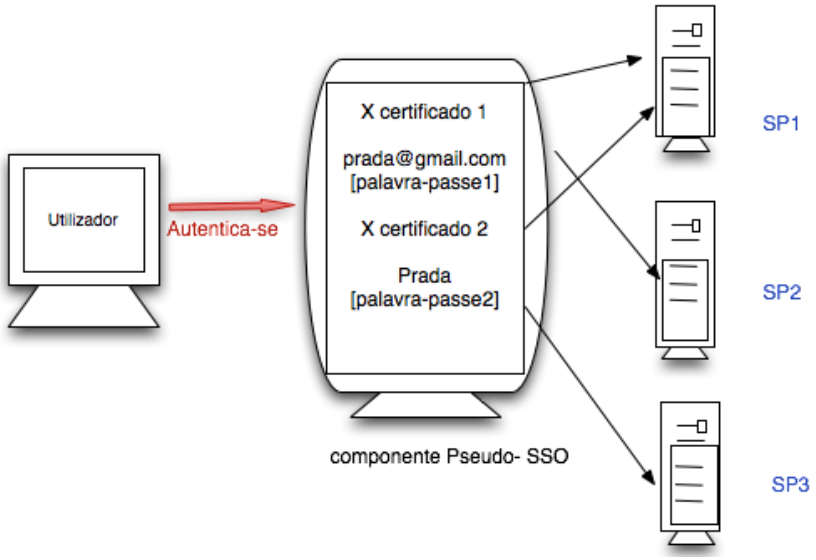

Figura 3.2 Cenário Pseudo-SSO

de modo a serem capazes de interagir com o único Sign-On do sistema, sendo o formato das identidades SSO dependente do sistema, que normalmente é uniforme como apresentado na figura 3.3. O providenciador de serviço de autenticação (ASP) será responsável pela autenticação entre as entidades e SPs para um conjunto definido de credenciais de autenticação. Caso um utilizador pretenda aceder a um serviço, deve-se autenticar perante o ASP, sendo este que estabelece a associação com os SPs pretendidos. De seguida, o ASP deverá enviar *authentication assertions*, ou seja, notificações sobre o estado de autenticação, com o objectivo de convencer o SP, que o utilizador está autenticado. De salientar, que a comunicação entre ASPs e SPs deve ser obrigatoriamente segura [26].

Um exemplo de uma arquitectura True-SSO corresponde ao ao Microsoft Passport [1] que permite aos utilizadores registados, possuidores de um e-mail válido e uma senha, terem acesso a todos os serviços do Passport network depois de autenticados.

3.2.1 Parâmetros a ter em consideração numa arquitectura SSO

Cada arquitectura SSO tem os seus pontos fortes e os fracos, sendo portanto essencial considerar cuidadosamente qual o ambiente e as necessidades antes de optar por uma determinada solução SSO. De forma a facilitar o processo de escolha, podemos resumir alguns dos critérios a ter em consideração, nomeadamente, desempenho, mobilidade/flexibilidade, usuabilidade, segurança, privacidade, escalabilidade, compatibilidade e manutenção [25, 18, 19].

- O **desempenho** pode ser medido em tempos de execução. Quanto mais rápido for o processo de autenticação e autorização para o utilizador, melhor será o desempenho;
- Neste contexto, posso definir **flexibilidade** como a facilidade que os utilizadores têm em

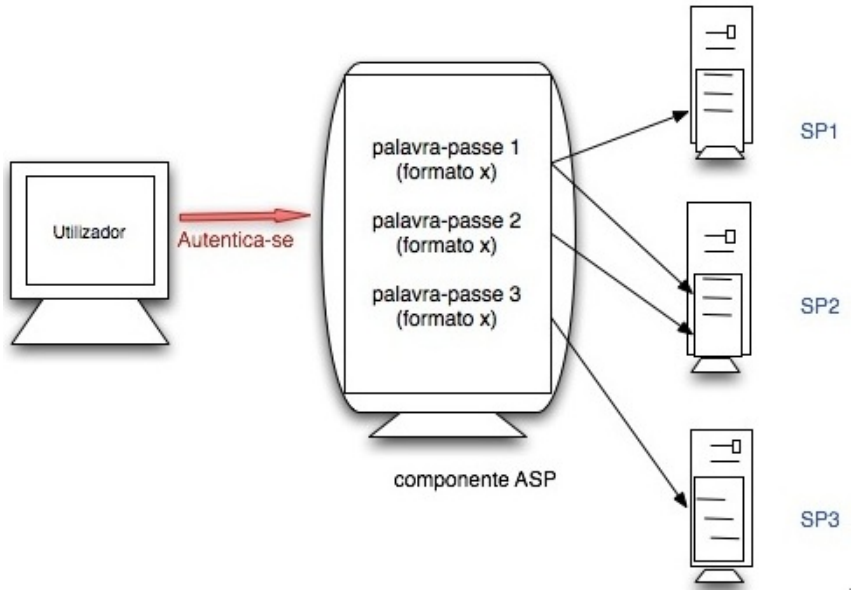

Figura 3.3 CenárioTrue-SSO

aceder às credenciais de autenticação. Uma aplicação diz-se não móvel se as credenciais estiverem apenas guardadas num único computador que usa. De salientar que o aumento de locais com credenciais de autenticação também faz aumentar a complexidade do sistema SSO e na diminuição do desempenho do sistema SSO (um dos problemas fulcrais de sistemas distribuídos, isto é, garantir coerência de dados em todos os sistemas).

- A **usuabilidade** mede-se pela facilidade de interacção do utilizador com o sistema SSO;
- A **segurança** de um sistema SSO é um dos critérios mais importantes e a que daremos mais relevo neste documento. Apesar de todas as vantagens inerentes ao sistema SSO, é evidente que ter uma única palavre-passe pode comprometer a segurança e a privacidade dos utilizadores, caso descoberta, podendo levar a resultados muito negativos. O atacante poderia ter acesso a um conjunto de serviços ou informações privadas, comprometendo toda a segurança do sistema. O sistema SSO deve portanto implementar mecanismos de segurança de forma a não permitir ataques às credenciais de autenticação, sendo essencial averiguar como esse conjunto de credenciais é manipulado pelo sistema. Deste modo, a segurança pode ser medida através da validação de determinadas propriedades de segurança que devem estar presentes no sistema. Essas propriedades são garantidas com a ajuda de protocolos de segurança, podendo estes ser o objecto de análise. O tema central desta tese centra-se precisamente no estudo da verificação de tais propriedades;
- A **privacidade** é outro ponto pertinente. Neste contexto, podemos medir privacidade como a facilidade de deduzir informação privada através de identidades SSO. Na abordagem Pseudo-SSO, o anonimato pode ser nulo, pois as identidades SSO são específicas

ao SP e alguns SPs podem exigir informação pessoal como o email. Em sistemas True-SSO as identidades SSO podem ser definidas de forma a não conter qualquer informação pessoal;

- A **escalabilidade** mede-se pelo número de servidores de autenticação. De salientar que o número de servidores de autenticação pode introduzir complexidade no sistema SSO;

- A **compatibilidade** pode definir-se pela facilidade de integrar novos mecanismos de autenticação;

- A **manutenção** pode caracterizar-se pelo tempo e esforço que é gasto na manutenção de um sistema SSO. Normalmente, é muito menos dispendioso implementar sistemas de Pseudo-SSO, porque eles não têm uma solução de segurança comum. Por outro lado, se o sistema Pseudo-SSO implementado sofrer alterações, nomeadamente, uma alteração no mecanismo de autenticação de um dos SPs faz com que haja uma alteração no componente Pseudo-SSO também, tornando-se muito mais custosa essa modificação. Para a arquitectura True-SSO o cenário é o oposto, mas no entanto a sua manutencão costuma ser muito mais cara.

3.3 OpenID

O OpenID consiste num conjunto de protocolos descentralizado e simples para "Single Sign-On" e gestão de identidades digitais na Internet, que começou a ser desenvolvido nos inícios de 2005. Actualmente está em vigor a versão 2.0 que é responsável por providenciar um modo para provar que determinado utilizador é possuidor de determinada identidade digital, provando deste modo a identidade do utilizador. O OpenID, através de um conjunto de tecnologias, providencia um mecanismo de autenticação global aos utilizadores de aplicações que implementam o OpenID, sem a necessidade de que essas mesmas aplicações acedam a credenciais de autenticação ou informação pessoal. No entanto, este protocolo só pode ser usado por aplicações possuidoras de uma estrutura baseada em web. Actualmente, a sua utilização tem aumentado, abrangendo várias corporações entre elas a Google, IBM, Microsoft, Facebook ou a Yahoo. No entanto, muitos dos utilizadores que usufruem dos serviços do OpenID não sabem o que realmente estão a usar. Uma razão para a sua maior utilização pode estar relacionada, para além da necessidade de um sistema que implemente SSO, com o facto deste protocolo pertencer à comunidade *open source*. Deste modo, qualquer pessoa pode usar o OpenID de forma gratuita, sem a obrigatoriedade de um registo ou aprovação por parte de uma organização. De salientar que normalmente as arquitecturas do OpenID são True-SSO, pois geralmente baseiam-se num único identificador, que irá relacionar o utilizador a diferentes aplicações que implementem OpenID[6]

3.3.1 Apresentação

Neste momento, já temos uma visão sobre o funcionamento de um sistema que implementa SSO e possuímos a noção de alguns dos requisitos que uma aplicação deve comportar, de modo a atingir determinadas funcionalidades. No entanto, para a análise deste sistema específico é preciso dissecar e perceber o sistema OpenID. Deste modo, descrevemos as suas características, funcionalidades e funcionamento mais pormenorizadamente, de forma a podermos validá-lo num cenário específico.

O OPenID 2.0 é um protocolo que implementa um mecanismo de SSO e de gestão de identidades digitais de forma a facilitar todo o procedimento de autorização e autenticação para utilizadores de aplicações com uma estrutura web. Uma das suas funcionalidades primordiais é a capacidade de um utilizador web usufruir de um sistema de autenticação para aceder a um conjunto de providenciadores de serviços (SP) que implementem OPenID. Assim, sempre que os utilizadores tentem aceder a uma terceira parte confiável OPenID (Relying Party ou RP), ou seja, a um site ou outro recurso on-line que exige acesso seguro aos seus serviços ou informações (como um SP que implemente o OPenID), será o provedor de identidades OPenID (OpenID Provider ou OP) responsável pela confirmação dos utilizadores. O OP é um componente do OPenID responsável por fornecer um mecanismo de autenticação a todos os SP que implemente OPenID, confirmando a identidade dos utilizadores de um RP e liberando os RPs desta tarefa. Os sites que implementam OPenID não precisam obrigatoriamente de lidar com credenciais ou mecanismos de autenticação para os seus utilizadores e os utilizadores não necessitam obrigatoriamente de criar contas novas para aceder a todos os RPs pretendidos. Apenas é necessário a autenticação por um site que funcione como provedor de identidade OPenID. Esse provedor fica responsável por confirmar a identidade do utilizador para outro site que suporta OPenID. Ao mesmo tempo, o OPenID é bastante flexível, pois permite ao utilizador mudar, se assim o pretender, de providenciador OPenID, porque a identidade continua sob o controlo do utilizador [22, 16].

Actualmente, muitas redes sociais funcionam como provedor de identidades OPenID, entre elas, a maior rede social no momento, o Facebook, que presentemente permite a identificação e autenticação por OPenID dos seus utilizadores, em sites que suportem OPenID[6].

3.3.1.1 Descrição do protocolo

A *especificação de autenticação do OpenID 2.0* é responsável por descrever todo o fluxo de comunicação, identificadores, segurança ou outras características fundamentais, para autenticar um utilizador numa determinada RP através de uma confirmação do OP. Aquilo que não for abrangido na presente especificação pode ser definido por outros serviços, que serão implementados acima deste protocolo, de forma a atingir melhor desempenho, segurança, privacidade ou interoperabilidade. Assim, o OPenID está desenhado para providenciar um serviço de autenticação portável e descentralizado para confirmar a identidade OPenID. Também é de salientar que o OPenID 2.0 requer que todas as mensagens sejam trocadas sobre o protocolo HTTP, definindo dois tipos de comunicação entre a RP e o OP:

- Comunicação directa: comunicação iniciada por um RP para um OP em que um RP faz um pedido directamente a um OP que também responde directamente. É usado para estabelecer associações[3] e verificações de *authentication assertions*[4]. As solicitações (Pedidos directos) são enviadas em mensagens codificadas em HTTP POST e as respostas a essas mesmas (Respostas directas) em HTTP Response cujas mensagens precisam de estar na forma *Key-Value* mais especificamente em "text/plain".

- Comunicação indirecta: as mensagens passam obrigatoriamente pelo utilizador cujo seu browser deve redirecionar a mensagem, através de HTTP POST ou HTTP response code. Normalmente a resposta é enviada usando a mesma comunicação indirecta. É utilizada para pedidos e respostas de autenticação.

3.3.1.2 Funcionamento do protocolo de autenticação

Sabendo que as *especificações de autenticação OPenID 2.0* são o cerne deste protocolo, torna-se necessário perceber como funciona. Antes de tudo, para o utilizador poder usufruir do OPenID, deve possuir um identificador OPenID que deve ser único para cada utilizador. Para isso deve-se registar previamente num provedor de identidades OPenID do seu agrado. Só depois de satisfeito tal requisito, é que o utilizador pode aceder ao site em que se pretende autenticar e que suporte OPenID, e realizar o processo de autenticação e autorização pelo mecanismo OPenID. Depois disto, o utilizador deve introduzir no formulário correcto existente no site o seu identificador OPenID, que será sujeito a uma função de normalização, com o objecto de uniformizá-lo e remover qualquer irregularidade. De seguida, a RP irá descobrir, usando o protocolo Yadis[5] ou um documento HTML apropriado, o endereço do servidor do provedor OPenID associado àquele identificador OPenID. A URL assim obtida constitui a identidade do OP final, pois o protocolo OpenID não requer que o servidor do provedor OpenID corra exactamente no mesmo endereço. Quando determinado o endereço, a RP vai redirecionar o browser do utilizador para esse endereço para identificar os seus utilizadores. De forma a ser possível este redirecionamento, a RP envia ao OP um pedido de autenticação OPenID, através de uma *HTTP redirect message*. De seguida, o OP usa um método de autenticação, que varia consoante o OP pois não é especificado pelo protocolo OPenID, e o OP autentica ou não o utilizador. De uma maneira geral, o OP pede ao utilizador a palavra-passe, que deve coincidir com a palavra-passe que o OP tem guardada. Se a palavra-passe não coincidir, então o browser do utilizador deve ser redireccionado para a RP, com uma falha de autenticação e o procedimento começa do início. Se coincidir, então, pode ser autenticado, e no caso do site precisar de informação adicional como credenciais ou detalhes da identidade, o provedor de identidades também deve perguntar ao utilizador se confia no site em questão e se autoriza o envio da informação que lhe está a ser pedida. Se a resposta for negativa, então o browser do utilizador é novamente

[3]Uma "associação"é um segredo compartilhado entre o OP e RP . Uma vez estabelecida, ela é usada para verificar mensagens de um protocolo.

[4]são dados que foram autenticados por um meio específico num determinado momento

[5]protocolo de comunicação para descoberta de serviços, como o OpenID, conectado a uma identidade Yadis

redireccionado para a RP com uma mensagem indicando que a autenticação foi rejeitada. Por outro lado, se for positiva, o site é redireccionado para uma página da RP com a respectiva informação fornecida pelo provedor de identidade. Por fim, a RP deve averiguar se as credenciais vieram do provedor de identidades respectivo, podendo verificar a origem dessa mensagem através da verificação da assinatura da resposta de autenticação. Na figura 3.4 podemos ver o funcionamento de uma forma geral do OPenID, segundo a perspectiva do utilizador, omitindo alguns passos.

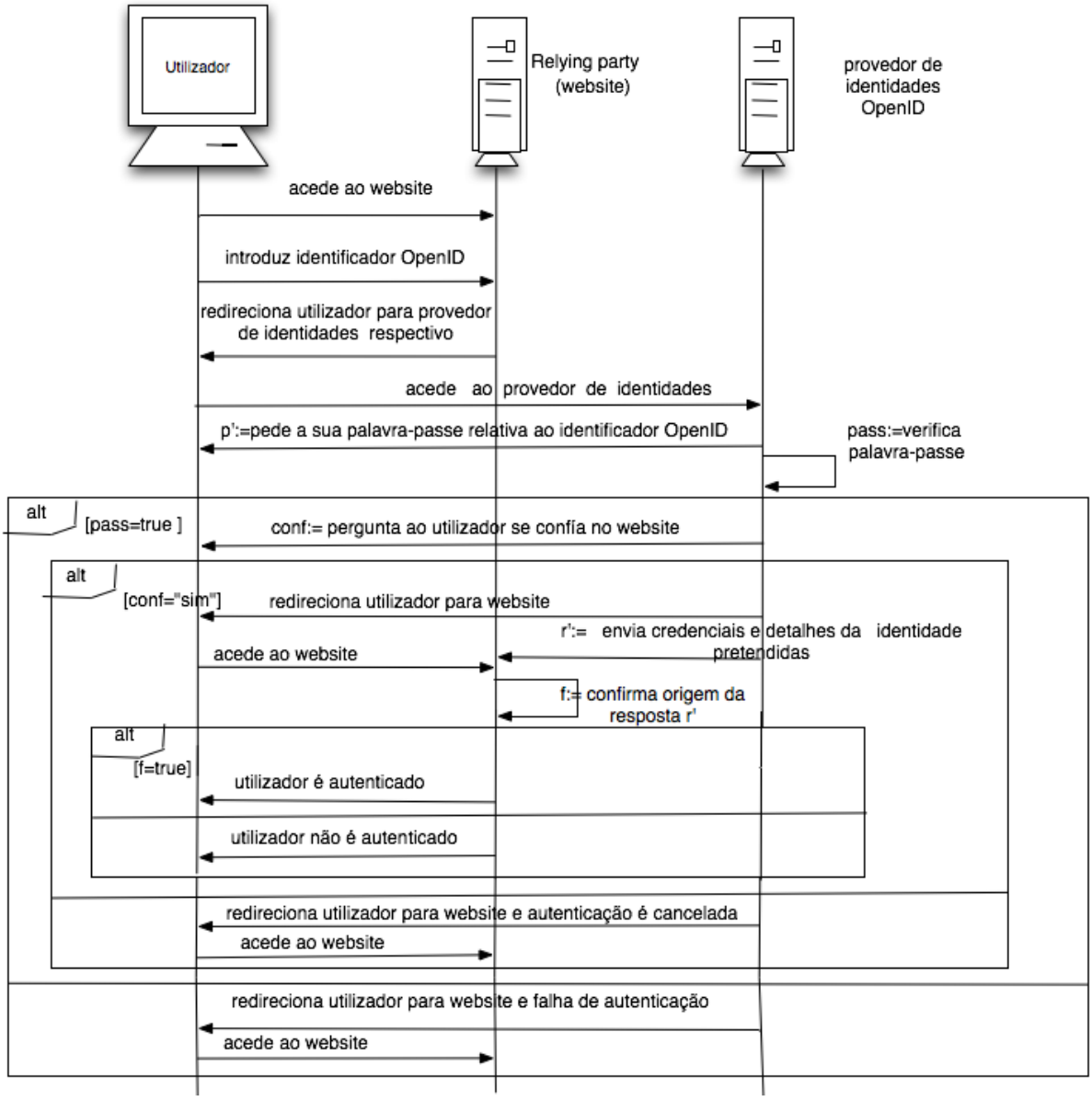

Figura 3.4 Esquema do funcionamento do OPenID segundo a perspectiva do utilizador

3.3.1.3 Normalização e processo de descoberta da URL do OP final

Já se referiu que numa fase inicial, o protocolo OpenID necessita de descobrir o endereço do OP que será invocado no processo de autenticação. Este mecanismo é composto por duas fases: normalização do identificador fornecido pelo utilizador e descoberta do endereço do OP. Detalhamos um pouco cada um destes processos.

O principal objectivo da normalização é uniformizar ou remover quaisquer irregularidades no identificador introduzido pelo utilizador. A operação de normalização deve retornar a um identificador que corresponda apenas a um URL ou a um Identificador de Recursos Extensível (XRI). Depois de ser efectuado o processo de normalização temos um endereço para contactar a OP. No entanto, o protocolo OPenID não exige que o servidor do OP esteja a correr obrigatoriamente nesse mesmo endereço. Deste modo, o RP deve "pedir" ao OP o endereço que deve ser utilizado, sendo necessário para isso a RP ter conhecimento da versão do protocolo OPenID que corre no servidor. A RP pode assim usar três diferentes métodos para determinar o endereço do servidor do provedor OPenID , que vai depender do tipo de identificador, após ter sido sujeito ao processo de normalização, nomeadamente :

- Se o identificador for um Identificador de Recursos Extensível (XRI), então o protocolo *XRI* deve ser usado para obter um documento *eXtensible resource descriptor sequence* (XRDS) ;

- Se o identificador for um URL e se a RP permitir o protocolo Yadis [4], então esse mesmo protocolo será usado nessa descoberta. Caso contrário, deve ser usado o documento Html da URL respectiva, para determinar o endereço correcto.

De salientar, que caso a RP não consiga determinar o OP, então o processo de autenticação, via OPenID, não pode ser realizado.

3.3.1.4 Abstração do protocolo

Detalhamos agora o fluxo de mensagens trocadas pelo protocolo. Na etapa 1, mostrada na Figura 3.5, o utilizador faz um pedido para ser autenticado num determinado RP, usando para isso um identificador. No segundo passo, a RP através do identificador já normalizado, tenta localizar o OP final usando os métodos de descoberta já descritos. O terceiro passo corresponde ao RP obter o documento XRDS, que nos indica qual o OP a ser usado. Os passos 4 e 5 são opcionais, sendo no entanto recomendados pela especificação de autenticação do OPenID 2.0 [2]. Estes passos correspondem à negociação de uma chave (k) através do algoritmo de partilha de chaves Diffie-Hellman (DH) para ser usada como chave na comunicação entre RP e OP. No passo 4, a RP envia um pedido de associação (Association Request) com os seguintes dados: IDreq que corresponde ao identificador da associação, T que indica o tipo de algoritmo DH utilizado, P que corresponde ao módulo de um número primo, G que corresponde ao gerador DH e PKRP que corresponde à respectiva chave pública do RP. Com estes passos, já não se torna necessário ao RP verificar todas as respostas de autenticação (authentication response),

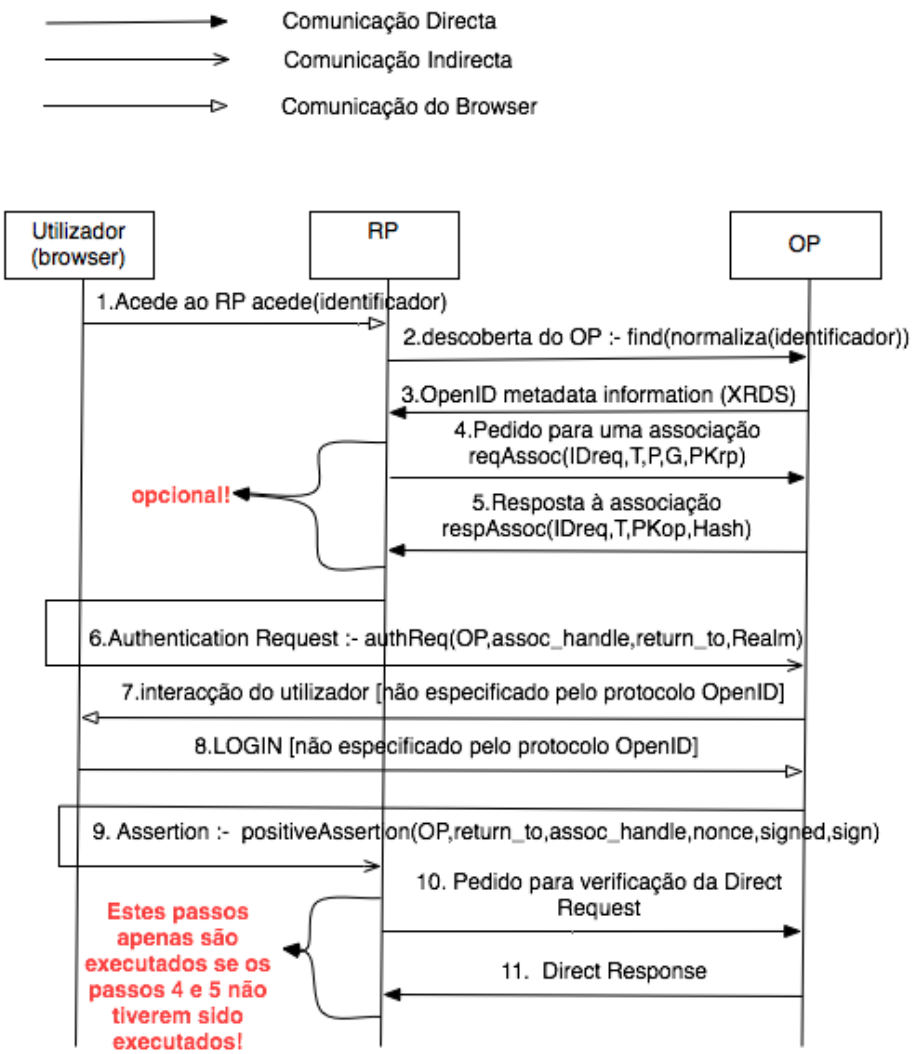

Figura 3.5 mensagens trocadas entre RP e OP sobre o protocolo OPenID

tornando assim todo o processo muito mais rápido pois já não é necessário criar chaves diferentes para cada pedido de autenticação. Como já foi referido anteriormente, a associação (mensagem *reqAssoc*) deve ser enviada como uma mensagem HTTP POST, pois trata-se de uma comunicação directa entre a RP e o OP. Depois, o OP envia uma resposta de autenticação, podendo enviar os seguintes parâmetros: IDreq que corresponde ao identificador da associação em causa, T que é o tipo de algoritmo DH a ser usado, PKop que corresponde à chave pública do OP e Hash que corresponde ao valor de uma hash que resulta do XOR da chave criada (K) com o MAC (*message authentication code*)[6] dessa mesma chave. O valor do IDreq deve ser

[6]garante a integridade das mensagens trocadas

guardado quer no OP quer no RP, para depois poder ser usado para localizar a chave associada e gerar a respectiva *assertion message* e hash para confirmar a integridade da chave partilhada. Depois, a RP envia indirectamente uma *Authentication Request* para a OP, passando antes pelo utilizador (passo 6 na figura 3.5). A mensagem enviada pode conter, além de outros parâmetros, uma identificação do OP (argumento OP da função authReq), um dado que deve ser usado para responder ao *request* (assoc_handle), um valor correspondente à URL para o qual o OP deve retornar o utilizador com a resposta que indica o estado do pedido (return_to) e um *Realm* que indica qual o domínio ou URL em que o utilizador pretende ser autenticado. Nos dois passos seguintes, nomedamente nos passos 7 e 8 da figura 3.5, o OP tenta autenticar o utilizador. Este procedimento não se encontra especificado no protocolo OPenID. Deste modo, estes passos ficam à escolha do próprio OP. De seguida, se essa autenticação for mal sucedida uma *assertion* negativa deve ser redirecionada para o utilizador e enviada para o RP , não podendo o processo de autenticação/autorização continuar. Caso seja bem sucedida, uma *assertion* positiva deve ser enviada. Por motivos óbvios, torna-se essencial verificar obrigatoriamente a origem dessa *assertion* positiva, sendo o utilizador autenticado apenas se tal origem for confirmada. Essa *assertion* é dada como válida se cumprir determinados requisitos, entre eles:

- a assinatura da *assertion* ser válida e todos os campos, que requerem assinatura, devem estar obrigatoriamente assinados. Esse mesmo processo de validação pode ser feito usando um dos dois métodos :

  - Verificando através de uma associação:
    A RP gera uma assinatura da mesma maneira como o OP gerou e, em seguida, compara a assinatura gerada com a resposta da assinatura. Se as assinaturas não coincidirem, então a *assertion* é inválida. Deste modo, devem ser enviados os parâmetros necessários para criar uma assinatura e a hash da assinatura da OP, o que implica que anteriormente deve haver um acordo de chaves(passo 4 e 5 da figura 3.5).
  - Verificando directamente com o provedor OPenID:
    A verificação da assinatura é feita pela OP. Deste modo, a RP deve enviar um pedido directo para o OP de forma a poder efectuar essa verificação. Nesse processo, o OP utiliza uma associação privada, que foi gerada quando emitiu a *assertion*. Assim, para a verificação da assinatura feita pelo OP, a terceira parte confiável envia uma solicitação directa ao OP. Para verificar a assinatura, o OP utiliza uma associação privada que foi gerada quando emitiu a *assertion* positiva. Assim, no passo 10 da Figura 3.5, a RP envia um pedido directo ao OP para obter uma verificação da assinatura. No passo 11, o OP deve enviar uma resposta directa a esse pedido, que deve conter o resultado da verificação da mensagem.

De salientar, que apenas detalhei alguns dos parâmetros e passos que considerei mais pertinentes para este estudo. No entanto, todos os dados podem ser consultados na especificação do protocolo OPenID [2].

3.3.1.5 Sobre as assinaturas no OpenID 2.0

Para uma associação é usado, normalmente, um autenticador de mensagem (MAC) para garantir a autenticação de mensagens no OpenID. O MAC difere de uma assinatura digital uma vez que o remetente e o receptor compartilham a mesma chave e os seus MACs são gerados e verificados através das mesmas chaves, garantindo unicamente autenticidade e integridade de dados. O receptor pode confirmar que a assinatura foi feita pelo emissor e que os dados são fiáveis. No entanto, contrariamente às assinaturas digitais, não garante a propriedade de não-repúdio, podendo um emissor negar a autenticidade de uma mensagem enviada por ele mesmo. Optam por manter a terminologia adoptada na especificação OpenID de se referir a utilização dos MACs como "assinatura". Convém no entanto manter presente que não segue a designação standard na comunidade criptográfica.

O OpenID 2.0 suporta assim duas assinaturas HMAC SHA1 ou HMAC SHA256 [3] como construções específicas para o cálculo de MACs. De uma forma genérica, resultam do hash da concatenação do OP, RP, identidade do utilizador, *Nonce* e chave K , cujos parâmetros são acordados previamente (passos 4 e 5 da figura 3.5) ou gerados pelo provedor de OPenID.

4 . Estudo de caso

4.1 Introdução

Ao longo destes últimos anos, opiniões sobre a segurança do OpenID têm divergido. Muitos autores defendem que qualquer sistema que implementa SSO não pode ser viável pois o atacante, ao descobrir a única palavra-passe mestre, pode ter acesso a uma gama de serviços e informações privadas. Mas isto é uma crítica ao próprio conceito de SSO, pelo que não nos parece razoável enquanto problema de segurança de um protocolo que se propõe centralizar os mecanismos de autenticação de serviços. O que se pretende neste trabalho é antes analisar o desenho do protocolo OpenID à luz das ferramentas disponíveis de verificação de protocolo. Desta forma, temos oportunidade de ilustrar a utilização dessas ferramentas num protocolo "realista". No processo de modelação do protocolo, vamo-nos concentrar nos fragmentos descritos como opcionais na especificação do protocolo. A ideia é então estudar o impacto desses fragmentos nas garantias oferecidas pelo protocolo e se, o resultado de análise o justificar, recomendar "boas práticas" relativamente à adopção desses fragmentos. Além disso, sabendo que o OpenID é um protocolo que implementa o SSO e que ultimamente tem ganho destaque, sendo mesmo utilizado por algumas das maiores corporações actuais, torna-se pertinente averiguar a existência de possíveis ataques.

4.2 Verificação formal através de ferramentas especializadas

Nesta parte da tese iremos analisar com a ajuda de ferramentas mencionadas no Capítulo 2, a abstração do protocolo de autenticação do protocolo OpenID 2.0 descrita no capítulo anterior. Iremos analisar o protocolo segundo dois possíveis cenários, executando e não executando os passos opcionais. Todo o código desenvolvido está devidamente comentado e anexado nesta tese.

4.2.1 Verificação com AVISPA e ProVerif

4.2.1.1 Modelação omitindo os passos 4 e 5 da Figura 3.5

Para analisar o protocolo, uma abstração do protocolo foi modelada em HLPSL (AVISPA) e spi-calcus (ProVerif). Dividi o protocolo OpenID segundo um conjuntos de regras de cada agente.

- Conjunto de regras que representa de uma forma abstracta o mecanismo responsável pela normalização do identificador OpenID que o utilizador introduz, e pela descoberta do OP relativo a essa mesma identidade. Assim, temos uma função de normalização que perante um determinado identificador devolve um agente e uma função para descobrir o OP relativo a essa identidade normalizada. Mais tarde, deve ser esclarecido que esse

agente corresponde ao OP final devido. No AVISPA corresponde à regra básica *role discover* e no ProVerif ao processo *processDiscover*.

```
Avispa:

 role discover ( D,R : agent,
           Normaliza: text -> agent,
                SND_PL, RCV_PL : channel(dy)
                )
         played_by D def=

         local State:nat,
                   Id: text

    init State := 19
    transition
     % recebe dados, entre eles, o Identificador OPenID do utilizador
     1. State = 19 /\ RCV_PL(Id'.D.R) =|>
    % envia o OP respectivo ao Id recebido
        State':=21 /\ SND_PL(Id'.Normaliza(Id').R)
 end role

 ProVerif:

 let processDiscover() =
    (*lÊ identificador *)
    in(cdr,m:identifier);
    (*normaliza*)
    let mn= normalize(m) in
    (*calcula op*)
    let op_end=providerList(mn) in
    (*envia dado ao RP*)
    out(cdr,op_end).
```

- Conjunto de regras que descrevem o comportamento de um utilizador que acede a um site que implementa openID (RP). O utilizador deve enviar o seu identificador e descobrir pelo RP qual é o seu OP final. Depois deve conectar-se ao OP final. Deste modo, deve ser estabelecido um mecanismo de autenticação de modo a que o utilizador possa comunicar de forma segura. Nos mesmos exemplos, utilizei o mesmo algoritmo de chave partilhada, enunciado na secção 2.3. A relembrar que o mecanismo de autenticação não está definido na especificação do OpenID, sendo da responsabidade do OP. Por fim, a *assertion positiva* enviada pelo OP deve ser redirecionada para o RP. No AVISPA corresponde à regra básica *role utilizador* e no ProVerif ao processo *processUser*.

```
Avispa:

role utilizador(
         U,R: agent,
                 K : symmetric_key, % chave simétrica (OP e utilizador)
                 Hash : hash_func, % função hash
                 Id : text, % Identificador
                 SND_RU, SND_OU, RCV_RU, RCV_OU : channel(dy))

        played_by U def=

        local State : nat,
              Nb,Nc: text, % "nonces" para o mecanismo de autenticação
```

```
14            Na : text, % "nonce" para a assertion
15            X,Ks : message,
16            O  : agent
17          init State:=0
18
19          transition
20          % start - sinal para começar o protocolo
21     1. State = 0 /\ RCV_RU(start) =|>
22      % envia os dados ao site, ou seja, RP (R)
23        State' := 2 /\ SND_RU(Id.U.R)
24      % recebe dados do OP, nomeadamente,o mesmo identificador,
25      %o valor do OP com que deve
26      % comunicar e o valor do R respectivo
27      2. State = 2 /\ RCV_RU(Id.O'.R) /\ RCV_OU(start) =|>
28      %  cria um "nonce" Nb e envia ao OP(O), cifrado pela chave simétrica
29         State' := 4 /\ Nb' := new() /\ SND_OU(O'.U.R.{Nb'}_K)
30      % recebe o "nonce" Nc do OP cifrado pela chave simétrica
31      3. State=4/\ RCV_OU({Nc'}_K) =|>
32      % cria uma chave de sessão que deve ser o Hash dos valores definidos pelo utilizador e OP
33      % e envia o o Nc cifrado pela criada de forma a confirmar valores estabelecidos
34      % além disso, o utilizador deve definir através do facto witness que quer se o par de OP
35      % concordando com o valor de Nc' numa autenticação forte
36         State' := 6 /\ Ks' := Hash(Nb.Nc') /\ SND_OU({Nc'}_Ks') /\ witness(U,O,autenticar,Nc')
37      % recebe a mensagem assertion positiva  e dps reencaminha a mensagem para a RP
38      4. State = 6/\ RCV_OU(O.R.U.Na'.X') =|>
39         State' := 8 /\ SND_RU(O.R.U.Na'.X')
40  end role
41
42  ProVerif:
43
44  let processUser(u:user,id:identifier,key:symmetric_key) =
45      (*manda o seu identificador ao provider*)
46          out(cur,id);
47          (*recebe do rp o OP final e identidade de RP*)
48      in(cur,op_end:provider);
49      in(cur,rp:relyP);
50      (*comunica com o OP correcto, envia dados necessarios*)
51      out(cup,op_end);
52      out(cup,rp);
53      out(cup,u);
54      (*mecanismo de autenticação- aleatorio*)
55      new Nb: nonce;
56      out(cup,senc(Nb,key));
57      out(cup,encrypt(secretNb,Nb));
58      in(cup,m_nc:bitstring);
59          let (Nc:nonce,K:symmetric_key)= sdec(m_nc,key) in
60                   event beginAuthparam(Nc);
61      (*definir password para comunicação*)
62      let (Ks:symmetric_key) = hash_funcS(concat(Nb,Nc)) in
63      (*confirmar o NC*)
64      out(cup,senc(Nc,Ks));
65      (*recebe uma assertion posita do OP*)
66      in(cup,assert:bitstring);
67      (*reenvia tal assertion ao RP*)
68      out(cur,assert).
69
```

- Conjunto de regras que descrevem o comportamento do OP OpenID. Inicialmente corresponde ao pedido de autenticação do utilizador. Depois de confirmada a identidade do utilizador, o OP cria uma *assertion positiva* e envia ao utilizador. Esse mesmo deve redirecionar a mensagem para o RP que envia a mensagem recebida novamente ao OP de

forma a confirmar a sua validade. No AVISPA corresponde à regra básica *role provIDs* e no ProVerif ao processo *processOP*.

```
AVISPA:

role provIDs(O,U: agent,
                     K : symmetric_key,
                         Hash : hash_func,
                     SND_UO, SND_PO, RCV_UO, RCV_PO : channel(dy)
                                          )
        played_by O def=

        local State : nat,
                  R : agent,
                  Ko : symmetric_key,
                  Na,Nb,Nc : text,
                  X,Ks : message

    init State:=11

    transition
    % recebe o nonce Nb do utilizador
    1. State = 11 /\ RCV_UO(O.U.R'.{Nb'}_K) =|>
    % envia o nonce Nc e calcula a chave de sessão
       State' := 13 /\ Nc' := new() /\SND_UO({Nc'}_K)
       /\ Ks':=Hash(Nb'.Nc') /\ secret(Ks',chave_sessao,{U,O})
    % recebe o nonce que que enviou com a devida chave de sessão estabelecida
    2. State = 13 /\ RCV_UO({Nc}_Ks) =|>
    % cria uma assertion positiva , cuja assinatura corresponde
    %  a hash de concatenação do valor do OP, RP
    % mais os parâmetros Nonce e K gerado pelo OP
       State' := 15 /\ Na' := new() /\ Ko' := new()
       /\ SND_UO(O.R.U.Na'.Hash(O.R.U.Na'.Ko')) /\
    % através do facto request é defenido que o agente O
    % aceita o valor do Nc e baseia-se na garantia que
    % U existe e concorda com esse valor
       request(O,U,autenticar,Nc)
   % recebe a request mensagem para validá-la, devendo ser validada
    3. State = 15 /\ RCV_PO(O.R.U.Na.Hash(O.R.U.Na.Ko)) =|>
   % facto witness: U quer se o par de O, "concordando" com assertion positiva
       State' := 17 /\ witness(U,O,assinatura, Hash(O.R.U.Na.Ko))
end role

ProVerif:

let processOP(op:provider,key:symmetric_key) =
   (*recebe "pedido" do utilizador para se autenticar*)
   in(cup,op_end:provider);
   in(cup,rp:relyP);
   in(cup,id:user);

   if op_end=op then  (*| confirmar *)
  (* passos 7 e 8 mecanismo de autenticação*)
       in(cup,m_nb:bitstring);
       new Nc : nonce; out(cup,senc(Nc,key));
       let (Nb:nonce,K:symmetric_key)= sdec(m_nb,key) in
        if K=key then
               out(cup,encrypt(secretNc,Nc));
                 (*definir password para autenticar*)
        let (Ks:symmetric_key) = hash_funcS(concat(Nb,Nc)) in
         in(cup,m_ks:bitstring);
         let Nc2 = sdec(m_ks,Ks) in
```

```
60            (*só deve prosseguir se passwords forem iguais*)
61               if Nc2=Nc then
62                   event endAuthparam(Nc);
63      (*cria assertion positiva*)
64                   new Na : nonce ; new Ko : symmetric_key ;
65
66                   out(cup,hash_func(concatA(op,rp,id,Na,Ko)));
67      (*valida assertion*)
68             in(crp,s:bitstring);
69             event beginSignparam(s).
```

- Correspondem ao conjunto de sequência de ações do RP OpenID. O RP recebe o identificador do utilizador e reecaminha para o primeiro mecanismo mencionado de forma a conseguir determinar o OP final correcto. Quando determinado, irá enviar os dados devidos ao utilizador para que este se possa conectar ao OP respectivo. Mais tarde, caso receba uma *assertion positiva* do utilizador deve enviar um pedido de confirmação ao OP. No AVISPA corresponde à regra básica *role site* e no ProVerif ao processo *processRP*.

```
1   AVISPA:
2
3   role site(
4        R,OH,O : agent,
5        SND_UR, SND_OR, SND_L, RCV_UR, RCV_OR,RCV_L : channel(dy))
6
7       played_by R def=
8
9       local State : nat,
10            Na, Id : text,
11            X : message,
12            U : agent
13
14      init State := 1
15      transition
16
17      1. State = 1 /\ RCV_UR(Id'.U'.R) /\ RCV_L(start) =|>
18      % reencaminha para o discover de forma a determinar qual o OP
19         State' := 3 /\ SND_L(Id'.OH.R)
20      % recebe o "valor do endereço" do OP
21      2. State = 3 /\ RCV_L(Id.O.R) =|>
22      % envia o valor do OP ao utilizador
23         State' := 5 /\ SND_UR(Id.O.R)
24      % recebe a assertion positiva
25      3. State = 5 /\ RCV_UR(O.R.U.Na'.X')  /\ RCV_OR(start) =|>
26       % reencaminha a assertion positiva ao OP
27         State' := 7 /\ SND_OR(O.R.U.Na'.X') /\ request(R,U,assinatura,X')
28
29  end role
30
31  ProVerif:
32
33  let processRP(RP: relyP) =
34      (*lê identificador*)
35      in(cur,m:bitstring);
36      (*tenta obter OP *)
37      out(cdr,m);
38      in(cdr,op_end:provider);
39      (*envia os dados ao utilizador*)
40      out(cur,op_end);
41      out(cur,RP);
42      (*recebe a assertion posita do utilizador*)
```

```
43      in(cur,assert:bitstring);
44      (*envia assertion de modo a OP poder verificar a sua validade*)
45      out(crp,assert);
46      event endSignparam(assert).
```

4.2.1.2 Modelação omitindo os passos 11 e 12 da Figura 3.5

Toda a modelação é bastante análoga, tendo-se usado o mesmo mecanismo de autenticação. A única diferença está no facto de a verificação da *assertion* ter sido feita através de uma associação, descripto todo o processo no Capítulo 3.3.

4.2.2 Verificação com CyptoVerif

Como pudemos constatar no Capítulo 2, com o Cryptoverif é possível descrever definições de segurança precisas e de efectuarmos uma prova formal segundo uma sequência de jogos. No entanto, o CryptoVerif é uma ferramenta muito mais complexa e muitos pormenores descritivos não foram considerados primordiais nesta prova. Além disso, o mecanismo de prova consiste em tentar validar todas as maneiras possíveis para validar um protocolo, sendo mais indicada para validar protocolos do que propriamente para encontrar ataques.

4.2.3 Análise de resultados

Nesta parte tentaremos averiguar falhas em algumas das propriedades de segurança, nomeadamente:

1. Autenticação dos valores acordados entre agentes;
2. Confidencialidade ;
3. Integridade de dados.

Os dois cenários de OpenID foram modelados por forma a tentarmos determinar ataques com a ajuda das duas ferramentas Avispa e ProVerif. A observar que no AVISPA recorremos ao *back-end cl-atse* por contar com determinadas optimizações e por permitir trabalhar sobre algumas equações algébricas, o que não era possível com o *SATMC* e o TA4SP. Em ambas as ferramentas, e para as duas modelações, algumas propriedades de segurança não são cumpridas. Nomeadamente, a confidencialidade de dados trocados e autenticação da assinatura. Em todas as ferramentas e nos dois cenários deram as mesmas falhas. Isto deve-se pelo facto de o mecanismo opcional apenas remover a necessidade de pedidos directos de verificação sempre que o OP e RP estabelecem uma associação, não tendo grande impacto em termos de segurança.

Descrição do ataque determinado pelas ferramentas: Como podemos ver na figura 4.1, o intruso consegue estabelecer contacto com a RP. Depois o RP envia-lhe o endereço do OP final, mensagem à qual o intruso responde de forma a ser solicitado para uma autenticação e consegue depois enviar a *request positiva* ao RP e futuramente vai conseguir autenticar-se ao RP. Para isto, o atacante podia efectuar, por exemplo, um ataque de redirecionamento que pode

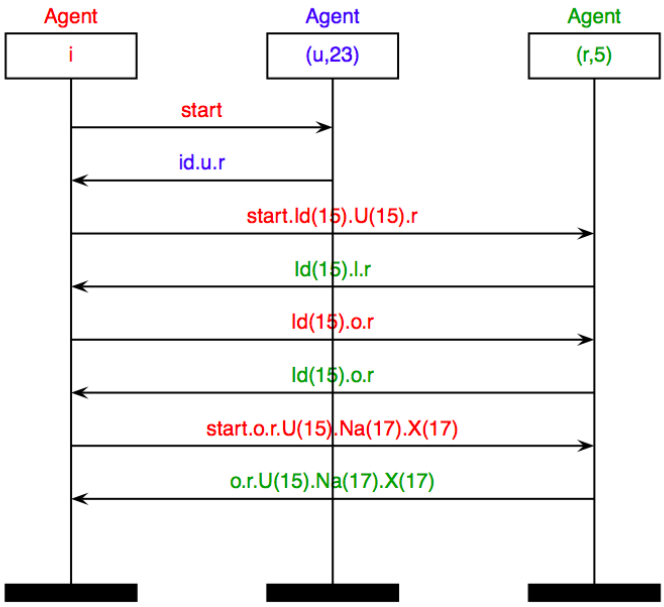

Figura 4.1 Imagem do ataque disponibilizado pela aplicação web do AVISPA para o primeiro cenário

ser alcançado quando o intruso corrompe a cache de DNS em um servidor DNS que aceita atualizações dinâmicas inseguros ou atacar a infra-estrutura de routing.

A existência de um possível ataque pode estar no facto de o atacante ter a capacidade de jogar todos os papeis, à excepção do papel da RP. Além disso, a integridade de mensagens não é protegida, bem como os dados trocados não são confidenciais pois não existe obrigatoriamente nenhum mecanismo de cifragem/ decifragem.

5 . Conclusões

Todas as ferramentas abordadas na tese mostraram-se essenciais e importantes na análise de protocolos de segurança. Com características e abordagens diferentes, mostraram adaptar-se a diferentes tipos de problemas. O CryptoVerif, sendo uma ferramenta baseada no Modelo Computacional, providencia provas mais realistas. No entanto, usando esta ferramenta, muitas provas são muito difíceis de fazer, nomeadamente, as que lidam com provas de redução mais complexas. Além disso, com esta ferramenta, muitas das vezes é necessário um esforço acrescido para comprender a sequência de jogos que corresponde à prova formal de segurança gerada. Por outro lado, as provas realizadas num Modelo Formal, apesar de omitirem certos pormenores, são muito mais naturais e fáceis de fazer. O AVISPA e o ProVerif são baseadas neste modelo, embora incorporem funcionalidades e mecanismos de raciocínio diferentes. No geral, achei o AVISPA uma ferramenta muito mais apelativa, sendo a sua linguagem de especificação muito mais simples. No entanto, o AVISPA não efectua certas verificações, o que não mostra ser um grande problema tendo em consideração a maior parte dos protocolos existentes.

Nesta tese, recorremos a duas dessas ferramentas (AVISPA e ProVerif) para analisar o protocolo OpenID 2.0. Foram detectadas algumas falhas de segurança no protocolo. Se um ataque de redirecionamento puder ser alcançado então o atacante pode aceder a uma conta de um utilizador. Além disso, as mensagens são trocadas sem a obrigatoriedade de um mecanismo que permita a confidencialidade. Assim um atacante pode ler todos os dados em texto claro, podendo até colectar dados sobre serviços a serem utilizados ou até informação adicional trocada entre os intervenientes. De modo a minimizar a possibilidade de ataques as mensagens trocadas entre os intervenientes deveriam ser cifradas, podendo ser usado, por exemplo, o protocolo SSL/TLS. Mesmo que um ataque de redirecionamento seja alcançado, se o atacante não tiver a capacidade de decifrar as mensagens transmitidas então essa informação é meramente lixo.

As ferramentas mostraram ser bastante úteis, simplificando todo o processo de análise de protocolos criptográficos. Todavia mais melhorias podem ser feitas, não só em mecanismos de raciocínio ou de especificação, mas por exemplo em editores de texto para tornar todo o processo de escrita menos propenso a erros e trabalhoso. Além disso, ao usufruirmos de tais ferramentas é imprescindível saber a sua precisão para avaliar propriedades de segurança. Torna-se importante averiguar a sua especificidade e sensibilidade, tal como, a necessidade de uma análise e estudo do protocolo e ambiente para a abstração do protocolo em questão, de modo a não conter irregularidades.

6. Apêndices

6.1 Resultados de análise - exemplo

6.1.1 Prova gerada pelo Avispa (back-end OFMC)

```
% OFMC
% Version of 2006/07/19
SUMMARY
  SAFE
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /opt/avispa-1.1/testsuite/results/example2.if
GOAL
  as_specified
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.83s
  visitedNodes: 105 nodes
  depth: 8 plies
```

6.1.2 Prova gerada pelo ProVerif

```
Process:
{1}new chave: key;
{2}out(c, );
(
    {3}!
    {4}in(c, );
    {5}new Na: nonce;
    {6}event beginparam(Na);
    {7}out(c, senc(Na,chave));
    {8}in(c, Nbc: bitstring);
    {9}let Nb: nonce = sdec(Nbc,chave) in
    {10}let h: bitstring = hash(Na,Nb) in
    {11}let k: key = h in
    {12}event beginparam(Nb);
    {13}out(c, senc(Nb,k))
) | (
    {14}!
    {15}in(c, Nac: bitstring);
    {16}let Na_4: nonce = sdec(Nac,chave) in
    {17}new Nb_5: nonce;
    {18}out(c, senc(Nb_5,chave));
    {19}in(c, Mc: bitstring);
    {20}let h_6: bitstring = hash(Na_4,Nb_5) in
    {21}let k_7: key = h_6 in
    {22}let Nbn: nonce = sdec(Mc,k_7) in
    {23}if Nbn = Nb_5 then
    {24}event endparam(Na_4);
    {25}event endparam(Nb_5);
    {26}out(c, senc(secretK,k_7))
)

-- Query event(endparam(x_8)) ==> event(beginparam(x_8))
nounif attacker(senc(x_34,chave[]))/-5000
Completing...
Starting query event(endparam(x_8)) ==> event(beginparam(x_8))
goal reachable: begin(beginparam(Nb_5[Nac = senc(Na[!1 = @sid_115],chave[]),
!1 = @sid_116])) & begin(beginparam(Na[!1 = @sid_115])) ->
end(endparam(Nb_5[Nac = senc(Na[!1 = @sid_115],chave[]),!1 = @sid_116]))
goal reachable: begin(beginparam(Nb_5[Nac = senc(Na[!1 = @sid_117],
chave[]),!1 = @sid_118])) & begin(beginparam(Na[!1 = @sid_117])) -> end(endparam(Na[!1 = @sid_117]))
RESULT event(endparam(x_8)) ==> event(beginparam(x_8)) is true.
-- Query not attacker(secretK[])
nounif attacker(senc(x_147,chave[]))/-5000
Completing...
Starting query not attacker(secretK[])
RESULT not attacker(secretK[]) is true.
```

6.1.3 Prova parcial gerada pelo CryptoVerif

```
Game 1 is
in(start, ());
new hk: hashkey;
new gk: keyseed;
let chave: key = kgen(gk) in
out(cc, ());
(
  in(c0, ());
  new Na_34: nonce;
  new s_35: seed;
  event beginparam(Na_34);
  out(c1, enc(Na_34, chave, s_35));
  in(c1, m_36: bitstring);
  let Nbbt_37: bitstringbot = dec(m_36, chave) in
  let Nb1: nonce = bitn(Nbbt_37) in
  let kn_38: nonce = concat(Na_34, Nb1) in
  let k': key = nkey(kn_38) in
  new se_39: seed;
  event endparam(Na_34);
  event beginparam(Nb1)
) | (
  in(c1, m_40: bitstring);
  let Nabt: bitstringbot = dec(m_40, chave) in
  let Na_41: nonce = bitn(Nabt) in
  new Nb: nonce;
  new s_42: seed;
  event beginparam(Nb);
  out(c1, enc(Nb, chave, s_42));
  in(c2, msg: bitstring);
  let kn_43: nonce = concat(Na_41, Nb) in
  let k1': key = nkey(kn_43) in
  new se_44: seed;
  let Nbbt_45: bitstringbot = decn(msg, k1') in
  let Nb2: nonce = bitn(Nbbt_45) in
  if (Nb = Nb2) then
  event endparam(Nb);
  let K: key = k1'
) | (
  ! !_33 <= qH
  in(hc1[!_33], x: nonce);
  out(hc2[!_33], hash(hk, x))
)

Applying simplify yields

Game 2 is
in(start, ());
new hk: hashkey;
new gk: keyseed;
let chave: key = kgen(gk) in
out(cc, ());
(
  in(c0, ());
  new Na_34: nonce;
  new s_35: seed;
  event beginparam(Na_34);
  out(c1, enc(Na_34, chave, s_35));
  in(c1, m_36: bitstring);
  let Nbbt_37: bitstringbot = dec(m_36, chave) in
  let Nb1: nonce = bitn(Nbbt_37) in
  let kn_38: nonce = concat(Na_34, Nb1) in
  let k': key = nkey(kn_38) in
  event endparam(Na_34);
  event beginparam(Nb1)
) | (
  in(c1, m_40: bitstring);
  let Nabt: bitstringbot = dec(m_40, chave) in
  let Na_41: nonce = bitn(Nabt) in
  new Nb: nonce;
  new s_42: seed;
  event beginparam(Nb);
  out(c1, enc(Nb, chave, s_42));
  in(c2, msg: bitstring);
  let kn_43: nonce = concat(Na_41, Nb) in
  let k1': key = nkey(kn_43) in
  let Nbbt_45: bitstringbot = decn(msg, k1') in
  let Nb2: nonce = bitn(Nbbt_45) in
  if (Nb = Nb2) then
  event endparam(Nb);
  let K: key = k1'
) | (
  ! !_33 <= qH
```

```
  in(hc1[!_33], x: nonce);
  out(hc2[!_33], hash(hk, x))
)

Applying remove assignments of useless yields

Game 3 is
in(start, ());
new hk: hashkey;
new gk: keyseed;
let chave: key = kgen(gk) in
out(cc, ());
(
  in(c0, ());
  new Na_34: nonce;
  new s_35: seed;
  event beginparam(Na_34);
  out(c1, enc(Na_34, chave, s_35));
  in(c1, m_36: bitstring);
  let Nbbt_37: bitstringbot = dec(m_36, chave) in
  let Nb1: nonce = bitn(Nbbt_37) in
  event endparam(Na_34);
  event beginparam(Nb1)
) | (
  in(c1, m_40: bitstring);
  let Nabt: bitstringbot = dec(m_40, chave) in
  let Na_41: nonce = bitn(Nabt) in
  new Nb: nonce;
  new s_42: seed;
  event beginparam(Nb);
  out(c1, enc(Nb, chave, s_42));
  in(c2, msg: bitstring);
  let kn_43: nonce = concat(Na_41, Nb) in
  let k1': key = nkey(kn_43) in
  let Nbbt_45: bitstringbot = decn(msg, k1') in
  let Nb2: nonce = bitn(Nbbt_45) in
  if (Nb = Nb2) then
  event endparam(Nb);
  let K: key = k1'
) | (
  ! !_33 <= qH
  in(hc1[!_33], x: nonce);
  out(hc2[!_33], hash(hk, x))
)

RESULT Proved event endparam(x) ==> true
RESULT Proved event endparam(x) ==> beginparam(x)
```

6.2 OpenID

6.2.1 Modelação omitindo os passos 4 e 5 da Figura 3.5

6.2.1.1 AVISPA

```
% omissão dos passos 4 e 5
% conjunto de regras relativa ao utilizador
role utilizador(
          U,R: agent,
                  K : symmetric_key, % chave simétrica, que apenas o utilizador e OP devem ter
                  Hash : hash_func, % função hash
                  Id : text, % Identificador
                  SND_RU, SND_OU, RCV_RU, RCV_OU : channel(dy)
                  )

         played_by U def=

         local State : nat,
               Nb,Nc: text, % "nonces" para o mecanismo de autenticação
           Na : text, % "nonce" para a assertion
           X,Ks : message,
           O  : agent
```

```
18          init State:=0
19
20          transition
21          % start - sinal para começar o protocolo
22          1. State = 0 /\ RCV_RU(start) =|>
23          % envia os dados ao site, ou seja, RP (R)
24             State' := 2 /\ SND_RU(Id.U.R)
25          % recebe dados do OP, nomeadamente,o mesmo identificador, o valor do OP com que deve
26          % comunicar e o valor do R respectivo
27      2. State = 2 /\ RCV_RU(Id.O'.R) /\ RCV_OU(start) =|>
28      %  cria um "nonce" Nb e envia ao OP(O), cifrado pela chave simétrica
29         State' := 4 /\ Nb' := new() /\ SND_OU(O'.U.R.{Nb'}_K)
30      % recebe o "nonce" Nc do OP cifrado pela chave simétrica
31      3. State=4/\ RCV_OU({Nc'}_K) =|>
32      % cria uma chave de sessão que deve ser o Hash dos valores definidos pelo utilizador e OP
33      % e envia o o Nc cifrado pela criada de forma a confirmar valores estabelecidos
34      % além disso, o utilizador deve definir através do facto witness que quer se o par de OP
35      % concordando com o valor de Nc' numa autenticação forte
36         State' := 6 /\ Ks' := Hash(Nb.Nc') /\ SND_OU({Nc'}_Ks') /\ witness(U,O,autenticar,Nc')
37      % recebe a mensagem assertion positiva  e dps reencaminha a mensagem para a RP
38      4. State = 6/\ RCV_OU(O.R.U.Na'.X') =|>
39         State' := 8 /\ SND_RU(O.R.U.Na'.X')
40  end role
41
42  % OpenID provider
43  role provIDs(O,U: agent,
44                       K : symmetric_key,
45                           Hash : hash_func,
46                       SND_UO, SND_PO, RCV_UO, RCV_PO : channel(dy)
47                                            )
48          played_by O def=
49
50          local State : nat,
51                    R : agent,
52                    Ko : symmetric_key,
53                    Na,Nb,Nc : text,
54                    X,Ks : message
55
56      init State:=11
57
58      transition
59      % recebe o nonce Nb do utilizador
60      1. State = 11 /\ RCV_UO(O.U.R'.{Nb'}_K) =|>
61      % envia o nonce Nc e calcula a chave de sessão
62         State' := 13 /\ Nc' := new() /\SND_UO({Nc'}_K) /\ Ks':=Hash(Nb'.Nc')
63          /\ secret(Ks',chave_sessao,{U,O})
64      % recebe o nonce que que enviou com a devida chave de sessão estabelecida
65      2. State = 13 /\ RCV_UO({Nc}_Ks) =|>
66      % cria uma assertion positiva , cuja assinatura corresponde
67      %(devido a ausência dos passos 4 e 5) a hash de concatenação
68      % do valor do OP, RP mais os parâmetros Nonce e K gerado pelo OP
69         State' := 15 /\ Na' := new() /\ Ko' := new() /\ SND_UO(O.R.U.Na'.Hash(O.R.U.Na'.Ko'))
70      % através do facto request é defenido que o agente O aceita o valor do Nc e baseia-se na garantia que
71      % U existe e concorda com esse valor
72         /\ request(O,U,autenticar,Nc)
73     % recebe a request mensagem para validá-la, devendo ser validada
74      3. State = 15 /\ RCV_PO(O.R.U.Na.Hash(O.R.U.Na.Ko)) =|>
75     % facto witness: U quer se o par de O, "concordando" com assertion positiva
76         State' := 17 /\ witness(U,O,assinatura, Hash(O.R.U.Na.Ko))
77
78   end role
79
80  role site(
```

```
81        R,OH,O : agent,
82        SND_UR, SND_OR, SND_L, RCV_UR, RCV_OR,RCV_L : channel(dy))
83
84      played_by R def=
85
86      local State : nat,
87            Na, Id : text,
88            X : message,
89            U : agent
90
91      init State := 1
92      transition
93
94      1. State = 1 /\ RCV_UR(Id'.U'.R) /\ RCV_L(start) =|>
95      % reencaminha para o discover de forma a determinar qual o OP
96         State' := 3 /\ SND_L(Id'.OH.R)
97      % recebe o "valor do endereço" do OP
98      2. State = 3 /\ RCV_L(Id.O.R) =|>
99      % envia o valor do OP ao utilizador
100        State' := 5 /\ SND_UR(Id.O.R)
101     % recebe a assertion positiva
102     3. State = 5 /\ RCV_UR(O.R.U.Na'.X') /\ RCV_OR(start) =|>
103      % reencaminha a assertion positiva ao OP
104        State' := 7 /\ SND_OR(O.R.U.Na'.X') /\ request(R,U,assinatura,X')
105 end role
106
107  role discover ( D,R : agent,
108            Normaliza: text -> agent,
109                 SND_PL, RCV_PL : channel(dy)
110                 )
111          played_by D def=
112
113          local State:nat,
114                    Id: text
115
116     init State := 19
117     transition
118
119     % recebe dados, entre eles, o Identificador OPenID do utilizador
120      1. State = 19 /\ RCV_PL(Id'.D.R) =|>
121     % envia o OP respectivo ao Id recebido
122         State':=21 /\ SND_PL(Id'.Normaliza(Id').R)
123
124  end role
125
126  role session(O, R, U, OH : agent,
127               K: symmetric_key,
128               H: hash_func,
129               Dados: text-> agent,
130               Id: text)
131    def=
132    local SND_RU, SND_OU, RCV_RU, RCV_OU,
133          SND_UR, SND_OR, SND_L, RCV_UR,
134          RCV_OR, RCV_L, SND_UO, SND_PO,
135          RCV_UO, RCV_PO, SND_PL, RCV_PL : channel(dy)
136    % uma sessão deve ser estabelecida pela execução de todos as acções dos agentes
137    composition
138          utilizador(U, R, K, H, Id,SND_RU, SND_OU, RCV_RU, RCV_OU) /\
139          provIDs(O, U, K, H, SND_UO, SND_PO, RCV_UO, RCV_PO) /\
140          site(R,OH,O, SND_UR, SND_OR, SND_L, RCV_UR, RCV_OR, RCV_L)
141     /\ discover(OH,R,Dados, SND_PL, RCV_PL)
142
143 end role
```

```
144
145 role environment()
146         def=
147                 const o,r,u,l : agent,
148                 assinatura,chave_sessao, autenticar :protocol_id,
149                 kuo, kio, kui : symmetric_key,
150                 h : hash_func,
151                 dados:  text -> agent,
152                 id : text
153
154                 % conhecimento do intruso
155                 intruder_knowledge = {o,r,u,l, kio, kui, h,i}
156
157                 composition
158                 session(o,r,u,l, kuo, h,dados,id)
159                 /\ session(o,r,u,i, kuo, h, dados,id)
160                 /\ session(o,r,i,l, kio, h,dados,id)
161                 /\ session(o,i,u,l, kuo, h,dados,id)
162                 /\ session(i,r,u,l, kui, h,dados,id)
163 end role
164
165 goal
166 % verificar a propriedade de segurança da se a chave de sessão
167 secrecy_of chave_sessao
168 % verificar a propriedade de autenticação do procedimento de autenticação
169 authentication_on autenticar
170 % verificar a propriedade da assinatura da assertion positiva
171 authentication_on assinatura
172 end goal
173
174 environment()
```

6.2.1.2 ProVerif

```
1  set traceDisplay = long.
2
3  free cup , cur , crp , cdr : channel.
4
5  (*tipos de dados*)
6  type identifier.
7  type provider.
8  type user.
9  type relyP.
10 type ident_norm.
11 type nonce.
12 type symmetric_key.
13
14
15 fun encrypt(bitstring,nonce) : bitstring.
16 reduc forall x: bitstring , y:nonce; decrypt(encrypt(x,y),y) = x.
17
18 (*delvove o provider quando recebe o identificador normalizado*)
19 fun providerList(ident_norm) : provider [data].
20 (*normalização de um determinado identificador*)
21 fun normalize(identifier) : ident_norm.
22 (*concatenação - diferentes funções porque as mesmas funções
23 têm de ter obrigatoriamente a mesma aridade e tipos*)
24 fun concat(nonce,nonce): nonce.
25 fun concatA(provider,relyP,user,nonce,symmetric_key) :bitstring.
26 (*funções hash*)
27 fun hash_funcS(nonce): symmetric_key.
28 fun hash_func(bitstring) : bitstring.
```

```
29
30  (* chave - mecanismo de autenticação para U-OP *)
31
32  fun senc(nonce,symmetric_key): bitstring.
33  reduc forall x:nonce, y:symmetric_key ; sdec(senc(x,y),y)= x.
34
35  (* Queries - segredo *)
36
37  free secretNc : bitstring [private].
38  free secretNb : bitstring [private].
39  query attacker(secretNc).
40  query attacker(secretNb).
41
42  (* Queries - autenticação *)
43
44  event beginAuthparam(nonce).
45  event endAuthparam(nonce).
46
47  event beginSignparam(bitstring).
48  event endSignparam(bitstring).
49
50  query x:nonce; event(endAuthparam(x)) ==> event(beginAuthparam(x)).
51  query x:bitstring; event(endSignparam(x)) ==> event(beginSignparam(x)).
52
53  (* Processos A e B (mensagens trocadas entre agentes )*)
54
55  let processUser(u:user,id:identifier,key:symmetric_key) =
56      (*manda o seu identificador ao provider*)
57          out(cur,id);
58          (*recebe do rp o OP final e identidade de RP*)
59      in(cur,op_end:provider);
60      in(cur,rp:relyP);
61      (*comunica com o OP correcto, envia dados nececessarios*)
62      out(cup,op_end);
63      out(cup,rp);
64      out(cup,u);
65      (*mecanismo de autenticação- aleatorio*)
66      new Nb: nonce;
67      out(cup,senc(Nb,key));
68      out(cup,encrypt(secretNb,Nb));
69      in(cup,m_nc:bitstring);
70          let (Nc:nonce,K:symmetric_key)= sdec(m_nc,key) in
71                    event beginAuthparam(Nc);
72      (*definir password para comunicação*)
73      let (Ks:symmetric_key) = hash_funcS(concat(Nb,Nc)) in
74      (*confirmar o NC*)
75      out(cup,senc(Nc,Ks));
76      (*recebe uma assertion posita do OP*)
77      in(cup,assert:bitstring);
78      (*reenvia tal assertion ao RP*)
79      out(cur,assert).
80
81  let processDiscover() =
82      (*lÊ identificador *)
83      in(cdr,m:identifier);
84      (*normaliza*)
85      let mn= normalize(m) in
86      (*calcula op*)
87      let op_end=providerList(mn) in
88      (*envia dado ao OP*)
89      out(cdr,op_end).
90
91  let processRP(RP: relyP) =
```

```
92      (*lê identificador*)
93      in(cur,m:bitstring);
94      (*tenta obter OP *)
95      out(cdr,m);
96      in(cdr,op_end:provider);
97      (*envia os dados ao utilizador*)
98      out(cur,op_end);
99      out(cur,RP);
100     (*recebe a assertion posita do utilizador*)
101     in(cur,assert:bitstring);
102     (*envia assertion de modo a OP poder verificar a sua validade*)
103     out(crp,assert);
104     event endSignparam(assert).
105
106 let processOP(op:provider,key:symmetric_key) =
107     (*recebe "pedido" do utilizador para se autenticar*)
108     in(cup,op_end:provider);
109     in(cup,rp:relyP);
110     in(cup,id:user);
111
112     if op_end=op then  (*| confirmar *)
113    (* passos 7 e 8 mecanismo de autenticação*)
114         in(cup,m_nb:bitstring);
115         new Nc : nonce; out(cup,senc(Nc,key));
116         let (Nb:nonce,K:symmetric_key)= sdec(m_nb,key) in
117          if K=key then
118                 out(cup,encrypt(secretNc,Nc));
119                  (*definir password para autenticar*)
120          let (Ks:symmetric_key) = hash_funcS(concat(Nb,Nc)) in
121           in(cup,m_ks:bitstring);
122           let Nc2 = sdec(m_ks,Ks) in
123           (*só deve prosseguir se passwords forem iguais*)
124              if Nc2=Nc then
125                  event endAuthparam(Nc);
126     (*cria assertion positiva*)
127                 new Na : nonce ; new Ko : symmetric_key ;
128                 out(cup,hash_func(concatA(op,rp,id,Na,Ko)));
129     (*valida assertion*)
130            in(crp,s:bitstring);
131            event beginSignparam(s).
132
133
134 (* interação de todos os processos, nomeadamente,
135  de todos os conjuntos de entidades com determinado comportamento *)
136 process
137     new op : provider;
138     new rp : relyP;
139     new u: user;
140     new id : identifier;
141     new key:symmetric_key; (*chave simétrica usada na autenticação entre utilizador e OP*)
142   ((!processUser(u,id,key))|(!processDiscover())|(!processRP(rp))|(!processOP(op,key)))
143
```

6.2.2 Modelação com os passos 4 e 5 da Figura 3.5

6.2.2.1 AVISPA

```
1  % OpenID -> com acordo de chaves diffie-hellman
2
3  role utilizador(
4           U,R: agent,
```

```
                  K : symmetric_key, % chave de autenticação do mecanismo des autenticação opcional
                  Hash : hash_func, % hash para mecanismo de autenticação opcional
                  Id : text,
                  SND_RU, SND_OU, RCV_RU, RCV_OU : channel(dy)
                  )

         played_by U def=
         local State : nat,
           O : agent,
           Na,Nb,Nc: text,
           X,Ks : message
         init State:=0

         transition
         % start - sinal para começar o protocolo
         1. State = 0 /\ RCV_RU(start) =|>
         % envia a "sua identidade" openID ao RP
            State' := 3 /\ SND_RU(Id.U.R)
         % recebe dados do OP, nomeadamente,o mesmo identificador, o valor do OP com que deve
         % comunicar e o valor do R respectivo
     2. State = 3 /\ RCV_RU(Id.U.R.O') /\ RCV_OU(start) =|>
        % autentica-se OP - opcional o mecanismo (descrito 2.2)
        State' := 6  /\ Nb' := new() /\ SND_OU(O'.U.R.{Nb'}_K)
     3. State= 6 /\ RCV_OU({Nc'}_K) =|>
     % cria uma chave de sessão que deve ser o Hash dos valores definidos pelo utilizador e OP
     % e envia o o Nc cifrado pela criada de forma a confirmar valores estabelecidos
     % além disso, o utilizador deve definir através do facto witness que quer se o par de OP
        State' := 8 /\ Ks' := Hash(Nb.Nc') /\ SND_OU({Nc'}_Ks')
        /\ witness(U,O,autenticar,Nc')
      % recebe a mensagem assertion positiva
     4. State = 8/\ RCV_OU(O.R.U.Na'.X') =|>
      % e dps reencaminha a mensagem para a RP
        State' := 9/\ SND_RU(O.R.U.Na'.X')

end role

role provIDs(O,U: agent,
                     K : symmetric_key,
                     Hash : hash_func, % hash para mecanismo de autenticação opcional
                     SND_UO, SND_PR, RCV_UO, RCV_PR : channel(dy),
                     Hrp : hash_func % hash para acordo dh
                                     )
         played_by O def=
         local State:nat,
               R : agent,
                   Ko: symmetric_key,
                   Dfp , G,P: message,
                   Kdf,HKdf,Pkr,Pkp,B,Y,I : message,
                   Na,Nb,Nc: text,
                   X,Ks : message

     init State:=3

     transition
     % Protocolo de partilha de chaves diffie-hellman de chave pública
     % recebe chave pública de RP
     1. State = 3 /\ RCV_PR(O.R'.G'.P'.Pkr') =|>
        State' := 5 /\  B':= new()
        % calcula ((g^b) mod p) e chave diffie helman = (Pkr^b)
        /\ Pkp':= exp(G,B')  /\  Kdf':=exp(Pkr',B)
        % aproximação  (gY)X e (gX)Y
        % /\ Kdf':= mod(I',P)
        % calcula a chave diffie hellman
```

```
68          /\ HKdf':= Hrp(Kdf') /\ SND_PR(O.R'.Pkp'.HKdf')
69           % facto witness: O quer se o par de R, "concordando" com a a hash da chave
70          /\ witness(O,R',autenticardf,HKdf')
71          % chave dff apenas conhecida para R e O
72          /\ secret(Kdf',chave_dff,{R',O})
73       2. State=5 /\  RCV_UO(O.U.R.{Nb'}_K) =|>
74           % envia o nonce Nc e calcula a chave de sessão
75          State' := 7 /\ Nc':= new() /\ SND_UO({Nc'}_K) /\ Ks':=Hash(Nb'.Nc')
76          % a chave de sessão só deve ser conhecida por U e O
77           /\ secret(Ks',chave_sessao,{U,O})
78       % através do facto request é defenido que o agente O aceita o valor do Nc e baseia-se na garantia que
79       % U existe e concorda com esse valor
80       3. State = 7 /\ RCV_UO({Nc}_Ks) =|>
81       % cria uma assertion positiva , cuja assinatura corresponde a hash de concatenação
82       % do valor do OP, RP mais os parâmetros Nonce e Kd e envia ao utilizador
83          State' := 10 /\ Na' := new()  /\ SND_UO(O.R.U.Na'.Hash(O.R.U.Na'.Kdf))
84          % facto witness: U quer se o par de O, "concordando" com assertion positiva
85          /\ witness(U,O,assinatura,Hash(O.R.U.Na'.Kdf))
86          /\ request(O,U,autenticar,Nc)
87    end role
88
89    role site(
90         R, OH , O : agent,
91         SND_UR, SND_PR, SND_OH, RCV_UR, RCV_PR,RCV_OH : channel(dy),
92         G,P: message,
93         Hrp:hash_func,
94         Hash: hash_func)
95
96       played_by R def=
97       local State : nat,
98             U : agent,
99             Id: text,
100            Dfs,Kdf,HKdf,HKdr,I,Pkp,Pkr,A,Xdf: message,
101            Na,Nb,Nc: text,
102                    X: message
103
104      init State := 1
105
106      transition
107      1. State = 1 /\ RCV_UR(Id'.U'.R) /\ RCV_OH(start) =|>
108         State' := 2 /\ SND_OH(Id'.R)
109     2. State = 2 /\ RCV_OH(Id.O.R) /\ RCV_PR(start) =|>
110        % troca de chaves diffie-hemann -> opcional (passo 4 e 5 da figura 5.2)
111          % 1- pedido para uma associação
112         State':= 4 /\ SND_UR(Id.U.R.O)
113        /\ A':= new() /\ Pkr':= exp(G,A')
114        /\ SND_PR(O.R.G.P.Pkr')
115      3. State= 4  /\ RCV_PR(O.R.Pkp'.HKdr') =|>
116         State':=9 /\ Kdf':=exp(Pkp',A)
117         %/\ Kdf':= mod(I',P)
118         /\   HKdf':= Hrp(Kdf')
119          /\ equal(HKdr',HKdf') /\ request(R,O,autenticardf,HKdr')
120      % através do facto request é defenido que o agente O aceita o valor do hash e baseia-se na garantia que
121      4. State = 9 /\ RCV_UR(O.R.U.Na'.Hash(O.R.U.Na'.Kdf))=|>
122         State' := 10        /\ request(U,R,assinatura,Hash(O.R.U.Na'.Kdf))
123   end role
124
125    role discover ( D,R : agent,
126              Normaliza: text -> agent,
127                   SND_PL, RCV_PL : channel(dy)
128                   )
129            played_by D def=
130
```

```
131           local State:nat,
132                     Id: text
133
134      init State := 3
135      transition
136
137       1. State = 3 /\ RCV_PL(Id'.R) =|>
138          State':=4 /\ SND_PL(Id'.Normaliza(Id').R)
139   end role
140
141   role session(O, R, U, OH : agent,
142                K: symmetric_key,
143                Hrp: hash_func,
144                Hash : hash_func, % hash para mecanismo de autenticação opcional
145                Dados: text-> agent,
146                Id: text,
147                G,P: message)
148     def=
149     local SND_RU, SND_OU, RCV_RU, RCV_OU,
150           SND_UR, SND_OR, SND_L, RCV_UR,
151           RCV_OR, RCV_L, SND_UO, SND_PO,
152           RCV_UO, RCV_PO, SND_PL, RCV_PL : channel(dy)
153
154     composition
155           utilizador(U, R, K,Hash, Id,SND_RU, SND_OU, RCV_RU, RCV_OU)
156           /\ provIDs(O, U, K,Hash, SND_UO, SND_PO, RCV_UO, RCV_PO, Hrp)
157      /\ site(R,OH,O,SND_UR, SND_OR, SND_L, RCV_UR, RCV_OR, RCV_L,G,P,Hrp,Hash)
158      /\ discover(OH,R,Dados, SND_PL, RCV_PL)
159
160   end role
161
162   role environment()
163           def=
164                   const o,r,u,l : agent,
165                   kuo, kio, kui : symmetric_key,
166                   hrp : hash_func,
167                   h : hash_func,
168               dados:  text -> agent,
169                   id : text,
170                   g,p: message,
171                   assinatura,chave_dff,chave_sessao, autenticar , autenticardf :protocol_id
172
173                   intruder_knowledge = {o,r,u,l, kio, kui,i,g,h,hrp}
174
175                   composition
176               session(o,r,u,l, kuo, hrp,h,dados,id,g,p)
177                   /\ session(o,r,u,i, kuo, hrp,h,dados,id,g,p)
178                   /\ session(o,r,i,l, kio, hrp,h,dados,id,g,p)
179                   /\ session(o,i,u,l, kuo, hrp,h,dados,id,g,p)
180                   /\ session(i,r,u,l, kui, hrp,h,dados,id,g,p)
181   end role
182
183   goal
184   secrecy_of chave_dff
185   secrecy_of chave_sessao
186   authentication_on autenticar
187   authentication_on autenticardf
188   authentication_on assinatura
189   end goal
190
191   environment()
```

6.2.2.2 ProVerif

```
set traceDisplay = long.

free cup , cur , crp , cdr : channel.

(*tipos de dados*)
type identifier.
type provider.
type user.
type relyP.
type ident_norm.
type nonce.
type symmetric_key.

(* acordo Diffie-Hellman *)
(*não pode ser codificado com destructores porque necessita de propriedades algébricas entre os termos*)
type G.
type exponent.
type Primo.

const g: G [data].
fun exp(G,exponent): G.

equation forall x: exponent, y: exponent; exp(exp(g,x),y) = exp(exp(g,y),x).


(*hash df*)
fun h_df(G): bitstring.

(**)
fun encrypt(bitstring,nonce) : bitstring.
reduc forall x: bitstring , y:nonce; decrypt(encrypt(x,y),y) = x.

fun encryptDF(bitstring,G) : bitstring.
reduc forall x: bitstring , y:G; decryptDF(encryptDF(x,y),y) = x.

(*delvove o provider quando recebe o identificador normalizado*)
fun providerList(ident_norm) : provider [data].
(*normaliza\c c\~ao de um determinado identificador*)
fun normalize(identifier) : ident_norm.
(*concatenação - diferentes funções porque as mesmas funções
têm de ter obrigatoriamente a mesma aridade e tipos*)
fun concat(nonce,nonce): nonce.
fun concatA(provider,relyP,user,nonce,G) :bitstring.
(*funções hash *)
fun hash_funcS(nonce): symmetric_key.
fun hash_func(bitstring) : bitstring.



(* chave - mecanismo de autenticação para U-OP *)

fun senc(nonce,symmetric_key): bitstring.
reduc forall x:nonce, y:symmetric_key ; sdec(senc(x,y),y)= x.

(* Queries - segredo *)

free secretDf : bitstring [private].
free secretNc : bitstring [private].
free secretNb : bitstring [private].
query attacker(secretNc).
query attacker(secretNb).
```

```
query attacker(secretDf).

(* Queries - autenticação *)

event beginAuthparam(nonce).
event endAuthparam(nonce).

event beginSignparam(bitstring).
event endSignparam(bitstring).
event beginDFparam(G).
event endDFparam(G).

query x:nonce; event(endAuthparam(x)) ==> event(beginAuthparam(x)).
query x:bitstring; event(endSignparam(x)) ==> event(beginSignparam(x)).
query x:G; event(endDFparam(x)) ==> event(beginDFparam(x)).


(* Processos A e B (mensagens trocadas entre agentes )*)

let processUser(u:user,id:identifier,key:symmetric_key) =
    (*manda o seu identificador ao provider*)
        out(cur,id);
        (*recebe do rp o OP final e identidade de RP*)
    in(cur,op_end:provider);
    in(cur,rp:relyP);
    (*comunica com o OP correcto, envia dados nececessarios*)
    out(cup,op_end);
    out(cup,rp);
    out(cup,u);
    (*mecanismo de autenticação- aleatorio*)
    new Nb: nonce;
    out(cup,senc(Nb,key));
    out(cup,encrypt(secretNb,Nb));
    in(cup,m_nc:bitstring);
        let (Nc:nonce,K:symmetric_key)= sdec(m_nc,key) in
                event beginAuthparam(Nc);
    (*definir password para comunicação*)
    let (Ks:symmetric_key) = hash_funcS(concat(Nb,Nc)) in
    (*confirmar o NC*)
    out(cup,senc(Nc,Ks));
    (*recebe uma assertion posita do OP*)
    in(cup,assert:bitstring);
    (*reenvia tal assertion ao RP*)
    out(cur,assert).


let processDiscover() =
    (*lÊ identificador *)
    in(cdr,m:identifier);
    (*normaliza*)
    let mn= normalize(m) in
    (*calcula op*)
    let op_end=providerList(mn) in
    (*envia dado ao OP*)
    out(cdr,op_end).

let processRP(RP: relyP) =
    (*lê identificador*)
    in(cur,m:bitstring);
    (*tenta obter OP *)
    out(cdr,m);
    in(cdr,op_end:provider);
    (* estabelece uma chave com o OP*)
```

```
125      out(crp,RP);
126      new Gf : G;
127      out(crp,Gf);
128      new P: Primo;
129      out(crp,P);
130      new a : exponent;
131      let (Pkr:G)= exp(Gf,a) in
132      out(crp,Pkr);
133      in(crp,Pkp:G);
134      in(crp,hash_keyr:bitstring);
135      let (KEY:G) =exp(Pkp,a) in
136      let (h_v:bitstring) = h_df(KEY) in
137              if hash_keyr= h_v then
138              out(crp,encryptDF(secretDf,KEY));
139              event endDFparam(KEY);
140              (*envia os dados ao utilizador*)
141              out(cur,op_end);
142              out(cur,RP);
143              (*recebe a assertion posita do utilizador*)
144              in(cur,assert:bitstring);
145              (*envia assertion de modo a OP poder verificar a sua validade*)
146              out(crp,assert);
147              event endSignparam(assert).
148
149  let processOP(op:provider,key:symmetric_key) =
150      (* acordo diffie hellman*)
151      in(crp,rp:relyP);
152      in(crp,Gf:G);
153      in(crp,P:Primo);
154      in(crp,Pkr:G);
155      new b : exponent;
156      let (Pkp:G)= exp(Gf,b) in
157      let (Key_df:G) = exp(Pkr,b) in
158      event beginDFparam(Key_df);
159      out(crp,Pkp) ;
160      out(crp,h_df(Key_df)) ;
161
162      (*recebe "pedido" do utilizador para se autenticar*)
163      in(cup,op_end:provider);
164      in(cup,rp_u:relyP);
165      in(cup,id:user);
166      if rp_u=rp   then
167      if op_end=op then  (*| confirmar *)
168     (* passos 7 e 8 mecanismo de autenticação*)
169          in(cup,m_nb:bitstring);
170          new Nc : nonce; out(cup,senc(Nc,key));
171          let (Nb:nonce,K:symmetric_key)= sdec(m_nb,key) in
172           if K=key then
173                  out(cup,encrypt(secretNc,Nc));
174                    (*definir password para autenticar*)
175           let (Ks:symmetric_key) = hash_funcS(concat(Nb,Nc)) in
176            in(cup,m_ks:bitstring);
177            let Nc2 = sdec(m_ks,Ks) in
178            (*só deve prosseguir se passwords forem iguais*)
179               if Nc2=Nc then
180                   event endAuthparam(Nc);
181      (*cria assertion positiva*)
182                  new Na : nonce ;
183
184                  out(cup,hash_func(concatA(op,rp,id,Na,Key_df)));
185      (*valida assertion*)
186             in(crp,s:bitstring);
187             event beginSignparam(s).
```

```
188
189
190  (* interação de todos os processos, nomeadamente,
191   de todos os conjuntos de entidades com determinado comportamento *)
192  process
193      new op : provider;
194      new rp : relyP;
195      new u: user;
196      new id : identifier;
197      new key:symmetric_key; (*chave simétrica usada na autenticação entre utilizador e OP*)
198    ((!processUser(u,id,key))|(!processDiscover())|(!processRP(rp))|(!processOP(op,key)))
199
```

6.3 Ataques detectados pelas ferramentas - OpenID

6.3.1 AVISPA

Ataque determinado pela ferramenta AVISPA no primeiro cenário (similar ao segundo cenário).

```
1   ATTACK TRACE
2    i -> (u,23):  start
3    (u,23) -> i:  id.u.r
4
5    i -> (r,5):  start.Id(15).U(15).r
6    (r,5) -> i:  Id(15).l.r
7
8    i -> (r,5):  Id(15).o.r
9    (r,5) -> i:  Id(15).o.r
10
11   i -> (r,5):  start.o.r.U(15).Na(17).X(17)
12   (r,5) -> i:  o.r.U(15).Na(17).X(17)
13                & Request(r,U(15),assinatura,X(17));
```

6.3.2 ProVerif

Ataque determinado pela ferramenta ProVerif no primeiro cenário (similar ao segundo cenário).

```
1
2   Completing...
3   Starting query event(endSignparam(x_18)) ==> event(beginSignparam(x_18))
4   goal reachable: attacker(x_184) -> end(endSignparam(x_184))
5   1. The attacker has some term m_188.
6   attacker(m_188).
7
8   2. The attacker has some term op_end_187.
9   attacker(op_end_187).
10
11  3. We assume as hypothesis that
12  attacker(x_190).
13
14  4. The message m_188 that the attacker may have by 1 may be received at input {29}.
15  The message op_end_187 that the attacker may have by 2 may be received at input {31}.
16  The message x_190 that the attacker may have by 3 may be received at input {34}.
17  So event endSignparam(x_190) may be executed at {36}.
18  end(endSignparam(x_190)).
19  (...)
20  The event endSignparam(a) is executed.
21  A trace has been found.
22  RESULT event(endSignparam(x_18)) ==> event(beginSignparam(x_18)) is false.
```

Bibliografia

[1] Microsoft passport network privacy statement, 2005. Disponível em `https://accountservices.passport.net/PPPrivacyStatement.srf`.

[2] Openid authentication 2.0, 2007. Disponível em `http://www.openid.pt/http://openid.net/specs/openid-authentication-2_0.html`.

[3] Openid authentication 2.0, 2007. Disponível em `http://openid.net/specs/openid-authentication-2_0.html#sign_algos`.

[4] Yadis protocol, 2007. Disponível em `http://yadis.org/wiki/What_is_Yadis`.

[5] Mobipassword 2.53 access management system, 2010. Disponível em `http://www.mobipassword.com/products.php?detailed_info`.

[6] Openid portugal, 2010. Disponível em `http://www.openid.pt/`.

[7] Hugo Andrés López. Frameworks for the Analysis of Security Protocols, A Survey, 2006.

[8] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*. Springer, 2005. Disponível em `http://www.avispa-project.org/publications.html`.

[9] David Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, December 2004.

[10] Mathieu Baudet. Random polynomial-time attacks and Dolev-Yao models. *Journal of Automata, Languages and Combinatorics*, 11(1):7–21, 2006.

[11] Bruno Blanchet. An Automatic Security Protocol Verifier based on Resolution Theorem Proving. (July), 2005.

[12] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October–December 2008. Special issue IEEE Symposium on Security and Privacy 2006.Disponível em http://doi.ieeecomputersociety.org/10.1109/TDSC.2007.1005.

[13] Bruno Blanchet. CryptoVerif Computationally Sound , Automatic Cryptographic Protocol Verifier User Manual, 2011. Disponível em `http://www.cryptoverif.ens.fr/`.

[14] Bruno Blanchet and Ben Smyth. Automatic Cryptographic Protocol Verifier , User Manual and Tutorial, 2011. Disponível em `http://www.proverif.ens.fr/`.

[15] Jan Cederquist and Mohammad Torabi Dashti. Complexity of fairness constraints for the dolev-yao attacker model. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1502–1509, New York, NY, USA, 2011. ACM.

[16] Max Charas. *Security in OpenID*. PhD thesis, Royal Institute of Technology, 2009.

[17] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A high level protocol specification language for industrial security-sensitive protocols. In *Austrian Computer Society*, pages 193–205, 2004.

[18] Jonathan Chinitz. Single Sign-On: Is It Really Possible? *Information Systems Security*, 9(3):1–14, July 2000.

[19] Jan De Clercq. Single sign-on architectures. In *Proceedings of the International Conference on Infrastructure Security*, InfraSec '02, pages 40–58, London, UK, UK, 2002. Springer-Verlag.

[20] Hubert Comon and Vitaly Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not ? *Journal of telecommunications and information tecnology*, 2001.

[21] Steve Kremer and Mark D. Ryan. Analysis of an electronic voting protocol in the applied pi-calculus. In Mooly Sagiv, editor, *Programming Languages and Systems — Proceedings of the 14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 186–200, Edinburgh, Scotland, UK, April 2005. Springer.

[22] Alexander Lindholm. *Security Evaluation of the OpenID Protocol* . PhD thesis, Royal Institute of Technology, 2009.

[23] Bruno Montalto and Carlos Caleiro. Modeling and Reasoning about an Attacker with Cryptanalytical Capabilities. *Electronic Notes in Theoretical Computer Science*, 253(3):143–165, November 2009.

[24] Rodrigo Borges Nogueira. *Verificação Formal de Protocolos Criptográficos , O Caso dos Protocolos em Cascata*. PhD thesis, Universidade de Brasília, 2008.

[25] Andreas Pashalidis and Chris J Mitchell. A Taxonomy of Single Sign-On Systems A Taxonomy of SSO Systems. pages 249–264, 2003.

[26] Andreas Pashalidis and Chris J. Mitchell. A taxonomy of single sign-on systems. In *Information Security and Privacy, 8th Australasian Conference, ACISP 2003*, pages 249–264. Springer-Verlag, 2003.

[27] Daniel Plasto and Avispa Team. HLPSL Tutorial , 2005.

[28] Daniel J Plasto. *Automated Analysis of Industrial Scale Security Protocols*. PhD thesis, Bond Univeristy.

[29] Mark Priestly. The logic of correctness in software engineering. In *CAiSE Workshops (2)*, pages 463–473, 2005.

[30] Ivo Seeba. CryptoVerif - the tool of crypto analysis, 2010.

[31] Laura Takkinen. Analysing Security Protocols with AVISPA.

[32] AVISPA Team. AVISPA v1 . 0 User Manual, 2006. Available at `http://www.avispa-project.org/`.

[33] Tomas Flanagan Tom Coffey, Reiner Dojen *. Formal verification: an imperative step in the design of security protocols*1. *Computer Networks*, 43(5):17, 2003.

[34] Luca Viganò. Automated Security Protocol Analysis With the AVISPA Tool. *Electronic Notes in Theoretical Computer Science*, 155:61–86, May 2006.