Formalising Identity Management Protocols

Md Sadek Ferdous
Electronics and Computer Science
University of Southampton
Southampton, UK
Email: S.Ferdous@soton.ac.uk

Ron Poet
School of Computing Science
University of Glasgow,
Glasgow, UK
Email: Ron.Poet@glasgow.ac.uk

***Abstract*—In this paper we present the formalisation of three well-known Identity Management protocols - SAML, OpenID and OAuth. The formalisation consists of two steps: formal specification using HLPSL (High-Level Protocol Specification Language) and formal verification using a state-of-the-art verification tool for security protocols called AVISPA (Automated Validation of Internet Security Protocols and Applications). The existing formalisation initiatives using AVISPA are based on SAML and OpenID, leaving OAuth entirely, even though OAuth is one of the most widely-used Internet protocols. Furthermore, the motivation of the existing initiatives was to identify any weakness. In this paper, we have taken an opposite approach as we are keen to present how to model these protocols correctly. Moreover, our formalisation is based on a model of identity and also captures the authentication mechanism; both of these are missing in the existing works.**

***Index Terms*—Identity Management, Formalisation, SAML, OpenID, OAuth, AVISPA.**

I. Introduction

With the proliferation of the Internet and web-enabled services, a great deal of online services have been introduced which deal with extremely sensitive information. This has raised the need for providing such services only to the authenticated and authorised users and similarly, accessing such services in a secure manner. To ensure that only authenticated and authorised users can access the respective online services while maintaining security and privacy, the concept of Identity Management (IdM, in short) has been introduced which resulted in various different Identity Management Systems (IMS, in short). Shibboleth [1] based on Security Assertion Markup Language (SAML) [2], OpenID [3] and OAuth [4] are the three most widely-used IMS.

Since these protocols are used to exchange sensitive personal information, it is extremely important to analyse the security of such protocols. Analysing a security protocol is a tricky task and can be carried out in many ways, for example, by security protocol experts or by utilising automated tools. Among these two, using an automated tool is the preferred approach as there is evidence of identifying security flaws even after the protocol was carefully analysed by experts [5, 6]. A protocol analysis tool, in essence, leverages different formal methods and computational models such as mathematical logic or inductive verification to identify flaws in a security protocol. One of the widely used tools is called AVISPA (Automated Validation of Internet Security Protocols and Applications) which uses mathematical logics to analyse security properties of an Internet protocol [7, 8].

AVISPA tool internally depends on the formal modelling of a security protocol in which a formal specification is defined using a specification language called High-Level Protocol Specification Language (HLPSL). There are existing works in which AVISPA has been utilised to model SAML and OpenID and then identify flaws in different SAML and OpenID implementations. For example, Armando et. al. identified severe flaws in the SAML-based authentication used for Google Apps using AVISPA [9]. They proposed a fix and based on this fix a correct modelling of SAML in HLPSL has been published in [10]. Similarly, AVISPA tool has been utilised to find exploits in OpenID implementations [6, 11]. Unfortunately, the existing approaches have the following shortcomings:

- None of the existing models considered the inclusion of an authentication step in the respective protocol flow.
- The modelling was carried out without considering a model of digital identity. Without a model of identity, the flow is incomplete. For example, how attributes are exchanged was not included in the protocol flow.
- The security goals of the existing approach were quite narrow in the sense they did not consider the security of many constructs of the respective protocol.
- There is no analysis of the OAuth protocol using AVISPA even though OAuth is one of the most widely used Internet protocols.
- The existing formalisations of OpenID (in [6, 11]) were more focused to identify flaws in the protocol and then to propose fixes. After proposing the fixed, the correct formalisation has not been published.

In this paper we address all of the identified shortcomings by presenting a formalisation of SAML, OpenID and OAuth using a model of identity and considering the authentication step in the protocol flow. The paper is organised as follows. In Section II, we present a model of identity. A brief introduction of SAML, OpenID and OAuth is presented in Section III along with the mathematical notations for presenting different constructs of these protocols. We briefly describe the AVISPA tool in Section IV and present the formalisation in Section V. A brief discussion of our approach is presented in Section VI and finally, we conclude in Section VII.

II. Digital Identity Model

An entity, for example a user or an organisation, is a physical or logical object which has a separate distinctive existence either in a physical or logical sense [12]. To define the identity of a user, we utilise the Digital Identity Model (DIM) introduced in [13]. According to this model, the (whole) identity of a user is actually distributed in different partial identities which are valid within a security domain (context) of an enterprise (organisation). Since the partial identity of a user is only valid within a domain, it is essential to specify the domain whenever a partial identity is mentioned. Each such partial identity consists of a number of attributes and their corresponding values, valid within the domain of a particular organisation.

Let us assume that D denotes the set of domains and d defines the domain of a single organisation whereas U_d denotes the set of users, A_d denotes the set of attributes and AV_d denotes the set of values for those attributes within d. Then, we can relate users and their attributes in a domain by the following partial function:

Definition 1: Let $atEntToVal_d : A_d \times U_d \to AV_d$ be the (partial) function that for an entity and attribute returns the corresponding value of the attribute in domain d.

The function is partial as not all entities have a value for each attribute. This also makes sense in practical systems as in many such systems, users are required to provide values for a number of attributes (e.g. email, telephone number, etc.). However, there remain some optional attributes (e.g. age, postal addresses, etc.) for which users may not provide any values.

Then, we can define the partial identity of a user (u) using the following definition.

Definition 2: For a domain d, the *partial identity* of a user $u \in U_d$ within d, denoted $parIdent^u_d$, is given by the set:

$$\{(a,v)\,|a \in A_d,\ atEntToVal_d(a,u) \text{ is defined and equals } v\}.$$

If we consider that there are n valid attribute-value pairs for a user u, the partial identity of u in d can also be defined as:

$$parIdent^u_d = \{(a_1,v_1),(a_2,v_2),(a_3,v_3)...(a_n,v_n)\}.$$

The (total/whole) identity of an entity can be defined as the union of all her partial identities in all domains.

Definition 3: For an entity $u \in U$, the *identity* of u is given by the set:

$$ident^u = \bigcup\{(d, parIdent^u_d)\,|\, d \in DOMAIN \text{ such that } u \in U_d\}.$$

The concept of Identity Management (IdM) has been proposed to facilitate the management of digital identities [12]. Formally, IdM consists of technologies and policies for representing and recognising entities with their digital identities [14]. A system that is used for identity management is called an Identity Management System (IMS). Each IMS involves the following parties:

User. A user receives services from a service provider (see below).

Service Provider. A Service Provider (SP, in short) is an organisation that provides services to the users or to other SPs. It is also known as the Relying Party (RP, in short).

Identity Provider. An Identity Provider (IdP, in short) is an organisation that provides digital identities to allow clients to receive services from an SP.

One of the functionalities of an IMS is to enable users to share their partial identities between different organisations (e.g. an IdP and SP). During this process, for the sake of privacy, users usually do not share their full partial identities between two organisations. Instead, a privacy-friendly approach is to share a limited view of a user's data across organisational boundaries. Such a limited view is defined as the profile of a user. Mathematically, a profile is a subset of the partial identity of a user within a domain: $PROFILE^u_d \subseteq parIdent^u_d$. Hence, we can define the profile of a user $u \in U_d$ in domain d in the following way, where $j \leq n$:

$$PROFILE^u_d = \{(a_1,v_1),(a_2,v_2),(a_3,v_3)...(a_j,v_j)\}.$$

III. Identity Management protocols

In this section we present a brief overview of SAML, OpenID and OAuth and introduce the mathematical notations of different constructs of these protocols.

A. SAML

SAML is one of the most widely deployed federated identity management technologies [2]. It is an XML (EXtensible Markup Language)-based standard for exchanging authentication and authorisation information between different autonomous domains. It relies on a request/response protocol in which one party (an SP) requests particular identity information about a user and the other party (an IdP) responds with the information using an assertion.

A SAML protocol flow between a user, an IdP and an SP is as follows. When a user tries to access a resource/service provided by the SP, the SP forwards the user to a service called the *Discovery Service* or the *Where Are You From (WAYF) Service*, where a pre-configured list of trusted IdPs is shown to the user. The user chooses her preferred IdP and then the user is forwarded to the IdP with a SAML authentication request. The authentication request consists of an identifier and the entity ID of the SP. We denote a SAML authentication request with *AuthnReq* and model it as presented in Table I, where id_{req} denotes the identifier in each SAML request and id_{sp} denotes the identifier of the SP which is represented as the *entityID* in SAML.

The IdP authenticates the user and a SAML response containing a SAML assertion with user attributes is sent back to the SP. The assertion consists of the profile of the user as released by the IdP. We denote the assertion with *SAMLAssrtn* and model it as presented in Table I. The SAML assertion is at first digitally signed and then embedded inside a SAML response which also consists of the request identifier, the entity ID of the IdP and the entity ID of the SP. We denote a SAML response with *SAMLResp* and model it as presented in Table

TABLE I
SAML, OpenID and OAuth notations

SAML Authentication Request:	$AuthnReq = (id_{req}, id_{sp})$
SAML Assertion:	$SAMLAssrtn = (PROFILE^{u}_{idp})$
SAML Response:	$SAMLResp = (id_{req}, id_{sp}, id_{idp}, (\{SAMLAssrtn\}_{K^{-1}_{idp}}))$
OpenID Request:	$OpenIDReq = (id_{req}, id_{idp}, id_{rp}, url_{sp})$
OpenID Response:	$OpenIDResp = (id_{req}, id_{idp}, id_{rp}, \{PROFILE^{u}_{idp}\}_{K^{-1}_{idp}})$
OAuth Request:	$OAuthReq = (id_{req}, url_{sp})$
OAuth Response:	$OAuthResp = (id_{req}, id_{rs}, \{PROFILE^{u}_{rs}\}_{K_{rs}^{-1}})$

I, where id_{idp} denotes the entity ID of the IdP. Furthermore, $\{SAMLAssrtn\}_{K^{-1}_{idp}}$ represents a digitally signed assertion with the private key of IdP (K^{-1}_{idp}).

When the SP receives the response, it extracts the SAML assertion and its signature is validated using the public key of the IdP. If the signature is valid, the SP retrieves the embedded user attributes (the profile, $PROFILE^{u}_{idp}$) from the assertion and takes an authorisation decision.

To enable the protocol flow discussed above, a notion of trust needs to be established between an IdP and an SP. This is achieved by exchanging the respective metadata, an XML file in specified formats containing different required information, of the IdP and SP and then storing such metadata at the appropriate repositories. This enables each party to build up the Trust Anchor List (TAL). After this, the IdP and SP are said to be part of the same federation, the so called Circle of Trust (CoT).

B. OpenID

OpenID is a decentralised IMS which provides SSO solutions for web services over the Internet [3]. Like SAML, the involved parties in OpenID are : Users, OpenID Providers (also known as Providers) and Service Providers, known as Relying Parties (RPs) in OpenID terminology. OpenID is based on the open trust concept where every party trusts each other and hence, there is no need to establish trust like SAML.

The protocol flow in OpenID is similar to SAML. A user submits a request to access a service provided by an SP. The SP provides an HTML form where the user submits her OpenID identifier. The identifier is used to discover the OpenID provider. Next, the user is forwarded to the provider where the user is authenticated. Then, an authentication response is returned back to the SP. The SP validates the response and if validated, the user is considered authenticated at the SP. The response may also consist the profile of the user as released by the provider. In this case, the user attributes are extracted from the response and then the SP takes an authorisation decision. Like SAML, we denote the OpenID request and response with *OpenIDReq* and *OpenIDResp* respectively. They are modelled as presented in Table I, where id_{req} denotes an identifier for the request, id_{idp} denotes the OpenID endpoint of the provider where the request is sent, id_{rp} denotes the RP identifier and url_{sp} denotes the URL of the SP where the respective response is returned. Here, $\{PROFILE^{u}_{idp}\}_{K^{-1}_{idp}}$ represents a user profile consisting of user attributes and their respective values which is digitally signed with the private key of the provider.

C. OAuth

OAuth, based on the notion of open trust like OpenID, is one of the fastest growing community-based specifications that allows any user to delegate her access right in a more user-friendly and secure way [4]. OAuth protocol comprises of four different classes of entities.

Resource Owners. Resource owners (or owners in short) own and control the protected resources and are capable of granting (delegating) limited access rights to third parties for accessing protected resources. They take the role of users as in SAML and OpenID.

Clients. Clients are third party applications that can make requests to protected resources on a user's behalf.

Authorisation Servers. Authorisation servers are responsible for granting access tokens to clients.

Resource Servers. Resource servers host protected resources and are responsible of accepting requests and providing resources. In many settings, resource and authorisation servers may be the same entity.

A simplified abstract protocol flow in OAuth is as follows. Let us assume that a resource owner wants to delegate access rights to a client (an application) so that the client can access some protected resources of the owner, hosted at the resource server. The flow starts with the client asking for authorisation of the owner to access those resources. Once the request is authorised, a credential known as the *authorisation grant* is returned to the client. The client then authenticates itself to an authorisation server and requests an access token using the authorisation grant. The authorisation server validates the grant and if valid, issues an access token. Then, the client requests access to the protected resources by presenting the access token to the resource server. The resource server validates the access token and if valid, the access request is granted and the requested resource is returned to the client.

In OAuth, user attributes are also considered as resources which can be retrieved via the OAuth response. We denote an OAuth request and response with *OAuthReq* and *OAuthResp* respectively and abstractly model them as presented in Table I, where id_{req} denotes an identifier for the request, id_{rs} denotes the identifier of the resource server and url_{sp} denotes the URL of the SP where the respective response is returned to. Here, $\{PROFILE^{u}_{rs}\}_{K^{-1}_{rs}}$ represents a user profile consisting of user attributes and their respective values which is digitally signed with the private key of the resource server.

IV. AVISPA

AVISPA [7, 8] tool is utilised for formal modelling and analysing security protocols automatically to determine if certain security properties are satisfied. The structure of the AVISPA tool is illustrated in Figure 1.

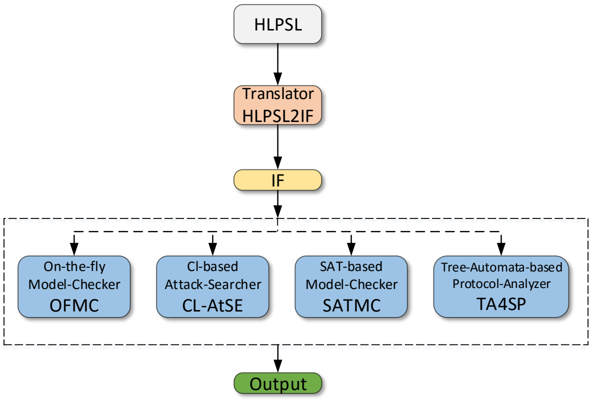

Fig. 1. AVISPA structure.

AVISPA utilises a formal specification language called High-Level Protocol Specification Language (HLPSL). The HLPSL is based on Lamport's Temporal Logics of Actions (TLA) [15] and allows anyone to specify security languages formally in a more human readable way. In HLPSL, different entities of a security protocol are modelled using roles which emulate entities. Each role defines different constructs consisting of variables that it needs to handle, agents emulating other roles in the protocol and channels exposing the communication interfaces of the role. In addition, within each role, the interactions with other roles using the defined channels are also specified. In essence, each interaction models a transition from one input state of the role to an output state. The full set of transitions defines the formal behaviour of each role of the security protocol. The security goals of the protocol are also specified in the HLPSL along with the attacker's knowledge. The AVISPA tool supports only the Dolev-Yao attacker model in which it is assumed that the intruder (attacker) has full access to all communication channels. It is also assumed that the attacker can intercept all messages sent and received over the specified channels as well as modify such messages, only if the intruder has knowledge about the cryptographic constructs and possesses the correct cryptographic credential (key).

The AVISPA tool is equipped with a translator called the *HLPSL2IF translator* which translates any security protocol model written in HLPSL into a low level protocol language called *Intermediate Format (IF)*. In summary, the AVISPA tool functions as follows. The security protocol and its goals are specified in HLPSL. This is then fed into the HLPSL2IF translator which translates the HLPSL into an IF. The IF is then fed into one of the back-ends which determines if the security goals are satisfied. Currently, four different back-ends are supported: the On-the-fly Model-Checker (OFMC), Constraint-Logic-based Attack Searcher (CL-AtSe), the SAT-based Model-Checker (SATMC) and the Tree Automata based on Automatic Approximations for the Analysis of Security Protocols (TA4SP).

In practice, protocol formalisations using AVISPA is a two step process: specification and verification.

- A protocol specification is carried out using two steps:
  - writing the protocol itself in Alice-Bob (A-B) notation, explained later and
  - specifying the protocol and its security goals in HLPSL and saving it in a file.
- Verification of the modelled protocol is carried out by executing the AVISPA tool using the HLPSL file to determine if the security goals are satisfied.

The A-B notation is merely a simplified flow of a security protocol specifying the interactions among different entities in the protocol. A protocol presented in A-B notation provides a clear illustration of the entities involved and the messages exchanged between them. An example of the A-B notation of the well-known Wide Mouth Frog protocol [16] is presented in Listing 1. Here, two entities A and B would like to setup a new session key KAB using a trusted server S. Furthermore, KAS represents a key shared between A and S whereas KBS represents a shared key between B and S. A generates and sends KAB encrypted with KAS to S and then S forwards it to B encrypted with KBS. After this, A and B will share the new session key to interact with each other.

```
1. A->S: {KAB}_KAS
2. S->B: {KAB}_KBS
```

Listing 1: Alice-Bob Notation

In the next step, the protocol is modelled using HLPSL along with a set of security goals. HLPSL supports two different types of security goals: *secrecy* and *authentication*. Here, *secrecy* refers to the goal which asserts that a certain value should be kept secret between only two entities. The format for specifying a secrecy goal is: *secrecy_of id*. The *id* represents a protocol id variable in HLPSL. The id is then embedded inside a goal fact which is written in HLPSL as: $secret(value, id, A, B)$. This represents that the goal represented using *id* asserts that the *value* should be kept secret between entities A and B. Then, the security fact is added as part of the transitions of the entity which generates the value. Finally, the secrecy goal is added at the section called *goal* inside the HLPSL file.

On the other hand, the *authentication* goal checks if an entity is correct in believing that the other entity is the intended peer of their interaction in the current session and agrees on a certain value upon reaching a certain state. The format for specifying an authentication goal is: *authentication_on id*, where *id* represents a protocol id variable in HLPSL. The

```
1. U->SP:{KU.U.SP.URI}_KSP
2. SP->U:{SP.U.IDP.{AuthnReq(ID.SP).URI}_inv(SAMLKSP)}_KU

3. U->IDP:{U.IDP.{AuthnReq(ID.SP).URI}_inv(SAMLKSP)}_KIDP
4. IDP->U:{IDP.U.Resource(LogInURI).N}_KU

5. U->IDP:{U.IDP.UName.Pass.N}_KIDP
6. IDP->U:{IDP.U.SP.{{SAMLResp(ID.IDP.SP.SAMLAssrtn)}_inv(SAMLKIDP).URI}_inv(KIDP)}_KU

7. U->SP:{U.SP.{{SAMLResp(ID.IDP.SP.SAMLAssrtn)}_inv(SAMLKIDP).URI}_inv(KU)}_KSP
8. SP->U:{{Resource(URI)}_inv(SP)}_KU
```

Listing 2: SAML protocol flow using Alice-Bob Notation

```
role user (
  U,IDP,SP : agent, KU,KSP,KIDP,SAMLKIDP : public_key,
  SSP,RSP,SIDP,RIDP          : channel(dy),
  Resource,PROFILE,SAMLAssrtn : hash_func, AuthnReq,SAMLResp
    ↪   : message)
played_by U def=
  local
   State, N,N1          : nat,
   ID,URI,UName,Pass,LogInURI          : text
   const u_idp_n,u_idp_uname,u_idp_pass: protocol_id

  init
   State := 0

  transition

   1. State=0 /\ RSP(start) =|> State':=2 /\ URI':=new() /\
   ↪   SSP({KU.U.SP.URI'}_KSP)
   2. State=2 /\ RSP({{U.IDP.AuthnReq(ID'.SP).URI}_inv(KSP)}
   ↪   _KU) =|> State':=4 /\ SIDP({U.IDP.AuthnReq(ID'.SP).
   ↪   URI}_KIDP)
   3. State=4 /\ RIDP({IDP.U.Resource(LogInURI).N'}_KU) =|>
   ↪   State':=6 /\ SIDP({U.IDP.UName.Pass.N'}_KIDP) /\
   ↪   request(U,IDP,u_idp_n,N') /\ witness(U,IDP,
   ↪   u_idp_uname,UName) /\ witness(U,IDP,u_idp_pass,Pass)
   4. State=6 /\ RIDP({IDP.{U.SP.{SAMLResp(ID'.IDP.SP.
   ↪   SAMLAssrtn(PROFILE(UName')).N1')}_SAMLKIDP}_inv(KIDP
   ↪   )}_KU) =|> State':=8 /\ SSP({{U.SP.{SAMLResp(ID'.IDP
   ↪   .SP.SAMLAssrtn(PROFILE(UName')).N1')}_inv(SAMLKIDP)}
   ↪   _inv(KU)}_KSP)
   5. State=8 /\ RSP({{Resource(URI)}_inv(KSP)}_KU) =|> State
   ↪   ':=10
end role
```

Listing 3: Modelling a User in SAML using HLPSL

goal facts related to the authentication goal are *witness* and *request*. One of this goal fact appears in the transition of one entity whereas the other goal fact appears in the transition of the other entity. Examples of these goals facts are $request(A, B, id, value)$ and $witness(B, A, id, value)$ added in the transitions of A and B respectively to check that if these entities are authenticated to each other by agreeing on the same *value* between themselves. Finally, the authentication goal is added at the section called *goal* inside the HLPSL file.

In the last step, the AVISPA tool is invoked with the HLPSL file for verification using the following command: % avispa file.hlpsl –{ofmc | satmc | clastse | ta4sp} with one of the backends.

V. Protocol formalisations

At first, the SAML protocol is illustrated in Alice-Bob notation in Listing 2. Next, we model the involved entities in SAML using HLPSL by defining each role (entity) and then specifying their interactions in Listing 3, Listing 4, Listing 5 and Listing 6.

The modelling of a user in SAML using HLPSL starts with defining some variables as presented in Listing 3. Among these variables, the agent variables represent the entities the current entity interacts with in the role. *KU*, *KIDP* and *KSP* represent the public keys of the user, IdP and the SP respectively. *SAMLKIDP* and *SAMLKSP* represent the public keys of the IdP and SP which are exchanged when the IdP and SP exchange their respective metadata and are hence are not public to the user and other entities. *SSP* represents the channel used by the user to send any message to the SP whereas *RSP* represents the receiving communication channel from the SP. Similarly, *SIDP* and *RIDP* represent the sending and receiving communication channels to and from the IdP respectively from the perspective of the user. The *protocol_id* variables (e.g. *c_idp_n*) represent the ids that are used while defining the security goals explained below. The transitions in each role essentially define the interactions of the entity with other entities. The role starts from an initial state (for the user it is 0) and upon receiving a message in a receiving channel it switches to its next state. The receiving and sending of messages for the role are analogous to the respective interactions presented in the A-B notation.

It is to be noted that HLPSL does not support secure (HTTPS) channel. Therefore, to ensure security and simulate the behaviour of a secure channel, all messages need to be encrypted with the corresponding keys before they are sent over any (insecure) channel. To achieve this, each sender encrypts every message with the public key of the receiver and then sends it over the defined channel. For example, $SSP(\{KU.U.SP.URI'\}\_KSP)$ indicates that the message consisting of the user agent U, his corresponding public key KU, the URL of the requested service *URI* and the SP user agent *SP* is encrypted with the public key of the SP (*KSP*) and then is sent over the *SSP* channel.

One important role is the *session* (Listing 6) which models a whole session for one single run of a protocol and contains the initialisation of all other roles with appropriate parameters.

```
role serviceProvider (
  U,IDP,SP : agent, KSP : public_key,
  SAMLKSP,SAMLKIDP : public_key, SND,RCV : channel(dy),
  Resource,PROFILE,SAMLAssrtn : hash_func, AuthnReq,SAMLResp : message)
played_by SP def=
  local
    State,N1 : nat,
    ID,UName : text,
    KU : public_key,
    URI : text
    const sp_idp_samlresponse,sp_idp_n1 : protocol_id

  init
    State:=1

  transition

  1. State=1 /\ RCV({KU'.U.SP.URI'}_KSP) =|>        State':=3 /\ ID':=new() /\ SND({{U.IDP.AuthnReq(ID'.SP).URI}_inv(SAMLKSP)}_KU')
  2. State=3 /\ RCV({{U.SP.{SAMLResp(ID.IDP.SP.SAMLAssrtn(PROFILE(UName')).N1')}_inv(SAMLKIDP)}_inv(KU)}_KSP) =|> State':=5 /\ SND({{Resource(URI)}_inv(KSP)}_KU) /\ request(SP,IDP,sp_idp_n1,N1') /\ request(SP,IDP,sp_idp_samlresponse,SAMLResp) /\ secret(Resource(URI),u_sp_resource,{U,SP})
end role
```

Listing 4: Modelling an SP in SAML using HLPSL

```
role identityProvider (
  U,IDP,SP : agent, KU,KIDP : public_key, SAMLKSP,SAMLKIDP : public_key,
  SND,RCV : channel(dy), Resource,PROFILE,SAMLAssrtn : hash_func,
  AuthnReq,SAMLResp : message)
played_by IDP def=

  local
    ID,UName,Pass : text,
    URI,LogInURI : text,
    State,N,N1 : nat
    const u_idp_n,sp_idp_n1,u_idp_uname,u_idp_pass, sp_idp_samlresponse : protocol_id

  init
    State:=7

  transition

  1. State=7 /\ RCV({U.IDP.AuthnReq(ID'.SP).URI'}_KIDP) =|> State':=9 /\ N' := new() /\ SND({IDP.U.Resource(LogInURI).N'}_KU) /\ witness(IDP,U,u_idp_n,N')
  2. State=9 /\ RCV({U.IDP.UName'.Pass'.N'}_KIDP) =|> State':=11 /\ N1':=new() /\ SND({IDP.{U.SP.{SAMLResp(ID.IDP.SP.SAMLAssrtn(PROFILE(UName')).N1')}_inv(SAMLKIDP)}_inv(KIDP)}_KU) /\ secret(N,u_idp_n,{U,IDP}) /\ secret(UName',u_idp_uname,{U,IDP}) /\ secret(Pass',u_idp_pass,{U,IDP}) /\ secret(N1,sp_idp_n1,{SP,IDP}) /\ secret(SAMLResp,sp_idp_samlresponse,{SP,IDP}) /\ witness(IDP,SP,sp_idp_n1,N1') /\ witness(IDP,SP,sp_idp_samlresponse,SAMLResp) /\ witness(IDP,U,u_idp_uname,UName)
end role
```

Listing 5: Modelling an IdP in SAML using HLPSL

```
role session (
  U,IDP,SP : agent,
  KU,KSP,KIDP,SAMLKIDP,SAMLKSP : public_key,
  Resource,PROFILE,SAMLAssrtn : hash_func,
  AuthnReq,SAMLResp : message)
def=
  local SUSP,RUSP,SUIDP,RUIDP : channel(dy)
   composition
   user(U,IDP,SP,KU,KSP,KIDP,SAMLKIDP,SUSP,RUSP,SUIDP,RUIDP, Resource,PROFILE,SAMLAssrtn,AuthnReq,SAMLResp)
   /\ serviceProvider(U,IDP,SP,KSP,SAMLKSP,SAMLKIDP,SUSP, RUSP,Resource,PROFILE,SAMLAssrtn,AuthnReq,SAMLResp)
   /\ identityProvider(U,IDP,SP,KU,KIDP,SAMLKSP,SAMLKIDP, SUIDP,RUIDP,Resource,PROFILE,SAMLAssrtn,AuthnReq, SAMLResp)
end role

role enviroment()
def=
  const u_sp_resource,sp_idp_samlresponse,u_idp_uname, u_idp_pass,u_idp_n,sp_idp_n1 : protocol_id,
  u,idp,sp : agent,
  ku,ksp,kidp,ki,samlkidp,samlksp         : public_key,
  resource,profile,samlassrtn : hash_func,
  authnreq,samlresp : message

  intruder_knowledge={u,sp,ksp,ki,inv(ki),idp,kidp,resource}
  composition
  session(u,idp,i,ku,ki,kidp,samlkidp,samlksp,resource, profile,samlassrtn,authnreq,samlresp)
  /\ session(u,idp,sp,ku,ksp,kidp,samlkidp,samlksp,resource, profile,samlassrtn,authnreq,samlresp)
  /\ session(i,idp,sp,ki,ksp,kidp,samlkidp,samlksp,resource, profile,samlassrtn,authnreq,samlresp)
end role

goal
secrecy_of u_sp_resource secrecy_of sp_idp_samlresponse secrecy_of u_idp_uname secrecy_of u_idp_pass secrecy_of u_idp_n secrecy_of sp_idp_n1

authentication_on u_idp_n authentication_on sp_idp_n1 authentication_on u_idp_uname authentication_on u_idp_pass authentication_on sp_idp_samlresponse
end goal

enviroment()
```

Listing 6: Modelling a session and security goals for SAML

Next, a top level role called *environment* is declared which defines three concurrent sessions. In the first and the third sessions, an intruder is assumed to impersonate an SP and a user respectively. Finally, several secrecy and authentication security goals are specified. The meaning of each security goal is elaborated in Table II and Table III.

By compiling all roles of SAML into a single HLPSL file and executing the AVISPA tool with the file and the *ofmc* backend will indicate that the modelled protocol is secure, indicating that the security goals are satisfied.

The OpenID and OAuth protocols are illustrated in Alice-Bob notation in Listing 7 and Listing 8 respectively. The cor-

TABLE II
SECRECY GOALS FOR SAML

secrecy_of	Meaning
c_sp_resource	provision of resource should be a secret between *U* and *SP*
sp_idp_samlresponse	AuthnResponse should be a secret between *SP* and *IDP*
c_idp_{uname,pass}	UName and Pass are secret between *U* and *IDP*
{c_idp_n,sp_idp_n1}	secret nonces between entities

TABLE III
AUTHENTICATION GOALS FOR SAML

authentication_on	Meaning
sp_idp_samlresponse	agreement on AuthnResponse between *SP* and *IDP*
c_idp_{uname,pass}	agreement on UName and Pass between *U* and *IDP*
{c_idp_n,sp_idp_n1}	agreement of nonces between entities

responding HLPSL files have been included in the appendix. Each specified role of each protocol essentially captures the interactions presented in the respective A-B notation. The security goals are similar to what presented for SAML. Copying the contents of each the modelled protocols in an HLPSL file and executing the AVISPA tool with the file and the *ofmc* backend will show that the modelled protocols are secure against the specified security goals. One interesting aspect to be noted for OAuth is the presence of *A* in the A-B notation (Listing 8) and the HLPSL in the appendix. Here, *A* represents a set of attributes (resources in OAuth terminology) that the RP would require to provide the requested service.

VI. Discussions

The security goals of secrecy of a certain value (or message) between two entities is satisfied only if that value/message is never exposed in plain text to other entities. Generally, such a value/message is generated by one entity and is then consumed by another entity. To ensure the secrecy of such a value/message, the generator must encrypt the value/message with the public key of the consumer. This will ensure that only the consumer having the corresponding private key can decrypt the value/message. We highlight two examples from the HLPSL representation of SAML, OpenID and OAuth. The SP encrypts the *SAMLResp* message with the public key of the IdP (*SAMLKIDP*) retrieved from the exchange metadata. This ensures that the *SAMLResp* can only be decrypted by the corresponding IdP. Similarly, in OAuth, the AS (Authorisation Server) encrypts *AccessToken* message with the public key of the RS (Resource Server). This ensures that only the RS can decrypt this message. Hence, even though, the encrypted message is received by the *APP* (application) as part of the flow, it can never decrypt the message.

On the other hand, the security goals of authenticity would require two entities must agree on a certain message/value. In the HLPSL representation of SAML, OpenID and OAuth, examples of such values or messages are username, password, SAMLResp, AuthzGrant and AccessToken.

Integrity of a certain value/message is modelled by signing it digitally with the private key of the generator which is then validated by the consumer using the public key of the generator. In HLPSL, such a digital signature is modelled using $\{value/message\}\_inv(key)$ where $inv(key)$ represents the private key of the corresponding entity.

To deter any reply attacks, each important message needs to be accompanied with fresh nonces and the secrecy and authenticity of such nonces must be satisfied. In our HLPSL representations, nonces (e.g. $N1, N2,$etc.) have been presented using natural numbers and their security has been satisfied with their respective security goals in each modelling.

VII. Conclusion

In this paper we present the formalisation of three well-known Identity Management protocols - SAML, OpenID and OAuth - using the AVISPA tool. At first, the flow of each of the protocol has been presented using A-B notation. Then, each protocol has been modelled using HLPSL. Finally, the HLPSL modelling of each protocol has been verified using the AVISPA tool and the security goals of each protocol have been satisfied, indicating the modelling of the protocols to be secure.

Our representations are the first ones to include the formalisation of username and password in the protocol. In addition, this is the first formalisation using AVISPA that is based on a model of digital identity. The inclusion of *PROFILE* introduces a powerful abstraction into the formalisation as it hides away the details of the attribute values which are passed during the protocol flow. Furthermore, none of the existing formalisations of SAML and OpenID considered the need for the extensive set of values/messages to be secret and to be authentic like ours.

In essence, these formalisations can act as the blue-prints which can be utilised to extend the formalisation of other advanced Identity Management scenarios such as *Attribute Aggregation*. An attribute aggregation mechanism allows a user to aggregate attributes from multiple providers in a single service session [17, 18]. There are different models of attribute aggregation, all of which are based on SAML. The SAML HLPSL formalisation presented here can be the basis for formalising these existing attribute aggregation models. Furthermore, the HLPSL representation can be leveraged to formalise different implementations of SAML, OpenID and OAuth to determine which implementations are secure.

References

[1] Shibboleth. https://shibboleth.net/.

[2] OASIS Standard. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. 15 March, 2005. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.

[3] OpenID Authentication 2.0 - Final. 5 December, 2007. http://openid.net/specs/openid-authentication-2_0.html.

[4] OAuth 2.0. http://oauth.net/2.

```
1. U->RP:{KU.U.RP.URI}_KRP
2. RP->U:{RP.U.IDP.{OpenIDReq(ID.IDP.RP.URI)}_inv(KRP)}_KU

3. U->IDP: {U.IDP.OpenIDReq(ID.IDP.RP.URI)}_KIDP
4. IDP->U: {IDP.U.Resource(LogInURI).N1}_KU

5. U->IDP: {U.IDP.UName.Pass.N1}_KIDP
6. IDP->U: {{IDP.U.RP.{OpenIDResp(ID.IDP.RP.PROFILE(UName)).N2}_inv(KIDP)}_inv(KRP)}_KU


7. U->RP: {{U.RP.{OpenIDResp(ID.IDP.RP.PROFILE(UName)).N2}_inv(KIDP)}_inv(KU)}_KRP
8. RP->IDP: {RP.IDP.{OpenIDResp(ID.IDP.RP.PROFILE(UName).N2)}_inv(KIDP)}_KIDP

9. IDP->RP: {{RP.IDP.CheckAuthnResponse.N3}_inv(KIDP)}_KRP
10. RP->U:{{Resource(URI)}_inv(RP)}_KU
```

Listing 7: OpenID protocol flow using Alice-Bob Notation

```
1. U->RP: {KU.U.RP.URI}_KRP
2. RP->APP: {RP.APP.AS.RS.{AuthZReq(ID.RP.AS.RS.A)}_inv(KRP)}_KAPP

3. APP->AS: {APP.AS.RS.{AuthZReq(ID.RP.AS.RS.A)}_inv(KAPP)}_KAS
4. AS->U: {AS.U.Resource(LogInURI).N1.ID}_KU
5. U->AS: {U.AS.{UName.Pass}_inv(KU).N1.ID}_KAS
6. AS->U: {AS.U.APP.{AuthZReq(ID.RP.AS.RS.A)}_inv(KAS).ID.N2}_KU

7. U->APP: {U.APP.AS.{{AuthZGrant(RP.AS.UName.A.N2.N3)}_inv(KU)}_KAS.ID}_KAPP

8. APP->AS: {APP.AS.{{AuthZGrant(RP.AS.UName.A.N2.N3)}_inv(KU)}_KAS.ID}_KAS
9. AS->APP: {AS.APP.RS.{{AccessToken(AS.RP.RS.UName.A.N4)}_inv(KAS).ID}_KRS}_KAPP

10. APP->RS: {{APP.RS.{{AccessToken(AS.RP.RS.UName.A.N4)}_inv(KAS).ID}_KRS}_inv(KAS).ID}_KRS
11. RS->APP: {RS.APP.RP.{RP.{PROFILE(A).N5}_inv(RS).ID}_KRP}_KAPP

12. APP->RP: {APP.RP.{AuthZResp({RP.{PROFILE(A).N5}_inv(KRS).ID}_KRP)}_inv(KAPP)}_KRP
13. RP->U: {{Resource(URI)}_inv(RP)}_KU
```

Listing 8: OAuth protocol flow using Alice-Bob Notation

[5] Fábrega, F. J. T., Herzog, J. C. and Guttman, J. D. *Strand spaces: why is a security protocol correct?*. In *IEEE S & P*, page 160–171, 1998. IEEE.

[6] Lindholm, A.. *Security evaluation of the OpenID protocol*. MSc. Thesis, KTH. Accessed on 1 June, 2016. 2009. https://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2009/rapporter09/lindholm_alexander_09076.pdf.

[7] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P. H., Héam, P., Kouchnarenko, O., Mantovani, J. and others. *The AVISPA tool for the automated validation of internet security protocols and applications*. In *International Conference on Computer Aided Verification*, page 281–285, 2005.

[8] AVISPA Project. Accessed on 10 April, 2016. http://www.avispa-project.org/.

[9] Armando, A., Carbone, R., Compagna, L., Cuellar, J. and Tobarra, L. *Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps*. In *the 6th ACM workshop on Formal methods in security engineering*, page 1–10, 2008. ACM.

[10] User-Contributed Protocol Specifications. Accessed on 1 July, 2016. http://www.avispa-project.org/library/user-contributed-library.html.

[11] Sun, S.. *Towards improving the usability and security of Web single sign-on systems*. PhD. Thesis, University of British Columbia. Accessed on 1 June, 2016. 2013. https://open.library.ubc.ca/cIRcle/collections/ubctheses/24/items/1.0103287.

[12] Ferdous, M.S., Jøsang, A., Singh, K. and Borgaonkar, R. *Security Usability of Petname Systems*. In *NordSec'09*, volume 5838 of *LNCS*, pages 44–59, Springer, 2009.

[13] Ferdous, M.S., Norman, G. and Poet, R. *Mathematical Modelling of Identity, Identity Management and Other Related Topics*. In the *SIN'14*, page 9–16, 2014.

[14] Jøsang, A., Al, M. and Suriadi, Z. *Usability and privacy in identity management architectures*. In *ACSW'07*, pages 143–152, 2007.

[15] Lamport, L. *The temporal logic of actions*. *ACM TOPLAS*, 16(3):872-923, 1994.

[16] Burrows, J.M, Abadi, M. and Needham, R. *A Logic of Authentication*. *Computer Systems*, 8(1):18-36, 1990.

[17] Chadwick, D.W. and Inman, G. *Attribute aggregation in federated identity management*. *Computer*, 42(5):33-40, 2009.

[18] Ferdous, M.S. and Poet, R. *Analysing Attribute Aggregation Models in Federated Identity Management*. In *SIN '13*, page 181-188, 2013. ACM.

Appendix

OpenID HLPSL

```
role user (U, IDP, RP : agent, KU, KRP, KIDP : public_key, SRP, RRP, SIDP, RIDP : channel(dy), Resource, PROFILE : hash_func,
  OpenIDReq, OpenIDResp: message)
played_by U def=
  local
    State,N,N1 : nat, ID,URI,UName,Pass,LogInURI : text const u_idp_n,u_idp_name,u_idp_passass : protocol_id
  init
    State := 0
  transition
  1. State=0 /\ RRP(start) =|> State’:=2 /\ URI’:=new() /\ SRP({KU.U.RP.URI’}_KRP)
  2. State=2 /\ RRP({{U.IDP.OpenIDReq(ID’.IDP.RP.URI)}_inv(KRP)}_KU) =|> State’:=4 /\ SIDP({U.IDP.OpenIDReq(ID’.IDP.RP.URI)}_KIDP)
  3. State=4 /\ RIDP({IDP.U.Resource(LogInURI).N’}_KU) =|> State’:=6 /\ SIDP({U.IDP.UName.Pass.N’}_KIDP) /\ request(U,IDP,u_idp_n,N’) /\ witness(U,IDP,u_idp_uname,U)
     ↪   /\ witness(U,IDP,u_idp_passass,Pass)
  4. State=6 /\ RIDP({IDP.{U.RP.{OpenIDResp(ID’.IDP.RP.PROFILE(UName’).N1’)}_inv(KIDP)}_inv(KIDP)}_KU) =|> State’:=8 /\ SRP({{U.RP.{OpenIDResp(ID’.IDP.RP. PROFILE(
     ↪   UName’).N1’)}_inv(KIDP)}_inv(KU)}_KRP)
  5. State=8 /\ RRP({{Resource(URI)}_inv(KRP)}_KU) =|> State’:=10
end role

role relyingParty (U,IDP,RP : agent, KRP,KIDP : public_key, SU,RU,SIDP,RIDP : channel(dy), Resource, PROFILE : hash_func,
  OpenIDReq, OpenIDResp: message)
played_by RP def=
  local
    State, N1, N2 : nat, ID,UName,URI : text, KU : public_key,  CheckAuthnResponse : bool
    const rp_idp_openidresponse,rp_idp_n1,rp_idp_n2,u_rp_resource,rp_idp_checkAuthnResponse                : protocol_id
  init
    State:=1
  transition
  1. State=1 /\ RU({KU’.U.RP.URI’}_KRP) =|>        State’:=3 /\ ID’:=new() /\ SU({{U.IDP.OpenIDReq(ID’.IDP.RP.URI)}_inv(KRP)}_KU’)
  2.  State=3 /\ RU({{U.RP.{OpenIDResp(ID.RP.PROFILE(UName’)).N1’}_inv(KIDP)}_inv(KRP)}_KU) =|> State’:=5 /\ SIDP({{OpenIDResp(ID.RP.PROFILE(UName’).N1’)}_inv(KIDP)}
     ↪   _KIDP) /\ request(RP,IDP,rp_idp_n1,N1’) /\ request(RP,IDP,rp_idp_openidresponse,OpenIDResp) /\ secret(OpenIDResp,rp_idp_openidresponse,{RP,IDP})
  3. State=5 /\ RIDP({{RP.IDP.CheckAuthnResponse’.N2’}_inv(KIDP)}_KRP) =|> State’:=7 /\ SU({{Resource(URI)}_inv(KRP)}_KU) /\ secret(Resource(URI),u_rp_resource,{U,RP
     ↪   })/\ request(RP,IDP,rp_idp_n2,N2’) /\ request(RP,IDP,rp_idp_checkAuthnResponse,CheckAuthnResponse) /\ secret(CheckAuthnResponse,rp_idp_checkAuthnResponse,{
     ↪   RP,IDP})
end role

role openIDProvider (U,IDP,RP : agent, KU,KIDP,KRP : public_key, SU,RU,SRP,RRP : channel(dy), Resource,PROFILE : hash_func,
  OpenIDReq, OpenIDResp: message)
played_by IDP def=
  local
    ID,UName,Pass,URI,LogInURI : text, CheckAuthnResponse :bool, State,N,N1,N2 : nat
    const u_idp_n,rp_idp_n1,rp_idp_n2,u_idp_uname,u_idp_pass,rp_idp_authnresponse,rp_idp_checkauthnresponse : protocol_id
  init
   State:=9
  transition
  1. State=9 /\ RU({U.IDP.OpenIDReq(ID’.RP).URI’}_KIDP) =|> State’:=11 /\ N’ := new() /\ SU({IDP.U.Resource(LogInURI).N’}_KU) /\ witness(IDP,U,u_idp_n,N’)
  2. State=11 /\ RU({U.IDP.UName’.Pass’.N’}_KIDP) =|> State’:=13 /\ N1’:=new() /\ SU({{U.RP.{OpenIDResp(ID.RP.PROFILE(UName’)).N1’}_inv(KIDP)}_inv(KRP)}_KU) /\
     ↪   secret(N,u_idp_n,{U,IDP}) /\ secret(N,u_idp_n,{U,IDP}) /\ secret(U,u_idp_uname,{U,IDP}) /\ secret(Pass,u_idp_pass,{U,IDP}) /\ secret(N1,rp_idp_n1,{RP,IDP})
     ↪   /\ secret(OpenIDResp,rp_idp_authnresponse,{RP,IDP}) /\ witness(IDP,RP,rp_idp_n1,N1’) /\ witness(IDP,RP,rp_idp_authnresponse,OpenIDResp) /\ witness(IDP,U,
     ↪   u_idp_uname,U) /\ witness(IDP,U,u_idp_uname,U)
  3. State=13 /\ RRP({{OpenIDResp(ID.RP.PROFILE(UName’).N1’)}_inv(KIDP)}_KIDP) =|> State’:=15 /\ N2’:=new() /\ CheckAuthnResponse’:=new() /\ SRP({{RP.IDP.
     ↪   CheckAuthnResponse’.N2’}_inv(KIDP)}_KRP) /\ witness(IDP,RP,rp_idp_checkauthnresponse,CheckAuthnResponse) /\ witness(IDP,RP,rp_idp_n2,N2’) /\ secret(N2,
     ↪   rp_idp_n2,{RP,IDP}) /\ secret(CheckAuthnResponse,rp_idp_checkauthnresponse,{RP,IDP})
end role

role session (U,IDP,RP : agent, KU,KRP,KIDP : public_key, Resource,PROFILE: hash_func, OpenIDReq, OpenIDResp: message)
def=
  local SURP,RURP,SUIDP,RUIDP,SRPIDP,RRPIDP,SIDPRP,RIDPRP: channel(dy)
  composition
    user(U,IDP,RP,KU,KRP,KIDP,SURP,RURP,SUIDP,RUIDP,Resource,PROFILE,OpenIDReq,OpenIDResp)
    /\ relyingParty(U,IDP,RP,KRP,KIDP,SURP,RURP,SRPIDP,RRPIDP,Resource,PROFILE,OpenIDReq,OpenIDResp)
    /\ openIDProvider(U,IDP,RP,KU,KIDP,KRP,SUIDP,RUIDP,SIDPRP,RIDPRP,Resource,PROFILE,OpenIDReq,OpenIDResp)
end role

role enviroment()
def=
  const u_rp_resource,rp_idp_authnresponse,rp_idp_checkauthnresponse,u_idp_uname,u_idp_pass,u_idp_n,rp_idp_n1,rp_idp_n2 : protocol_id,
  u,idp,rp : agent, ku,krp,kidp,ki : public_key, resource,profile : hash_func, openidreq,openidresp : message
  intruder_knowledge={u,rp,krp,ki,inv(ki),idp,kidp,resource}
  composition
   session(u,idp,i,ku,ki,kidp,resource,profile,openidreq,openidresp)
   /\ session(u,idp,rp,ku,krp,kidp,resource,profile,openidreq,openidresp)
   /\ session(i,idp,rp,ki,krp,kidp,resource,profile,openidreq,openidresp)
end role

goal
secrecy_of u_rp_resource secrecy_of rp_idp_authnresponse secrecy_of rp_idp_checkauthnresponse secrecy_of u_idp_uname secrecy_of u_idp_pass secrecy_of u_idp_n
   ↪   secrecy_of rp_idp_n1 secrecy_of rp_idp_n2

authentication_on u_idp_n authentication_on rp_idp_n1 authentication_on rp_idp_n2 authentication_on u_idp_uname authentication_on u_idp_pass authentication_on
   ↪   rp_idp_authnresponse authentication_on rp_idp_checkauthnresponse
end goal

enviroment()
```

OAuth HLPSL

```
role user (
  U,RP,AS,APP,RS : agent, KU,KRP,KAS,KAPP : public_key, SRP,RRP,SAS,RAS,SAPP,RAPP        : channel(dy), Resource,PROFILE,AuthZGrant : hash_func,
  AuthZReq : message)
played_by U def=
  local
    State,N1,N2,N3 : nat, ID,URI,UName,Pass,LogInURI : text, A : text set
    const u_as_n1,u_as_n3,u_as_uname,u_as_pass,u_as_authzgrant : protocol_id
  init
    State := 0
  transition
  1. State=0 /\ RRP(start) =|> State’:=2 /\ URI’:=new() /\ SRP({KU.U.RP.URI’}_KRP)
  2. State=2 /\ RAS({AS.U.Resource(LogInURI).N1’.ID’.A’}_KU) =|> State’:=4 /\ SAS({U.AS.{UName.Pass}_inv(KU).N1’.ID’}_KAS) /\ request(U,AS,u_as_n1,N1’) /\ witness(U,
     ↪   AS,u_as_uname,UName) /\ witness(U,AS,u_as_pass,Pass) /\ secret(UName,u_as_uname,{U,AS}) /\ secret(Pass,u_as_pass,{U,AS})
  3. State=4 /\ RAS({AS.U.APP.{AuthZReq(ID.RP.AS.RS.A’)}_inv(KAS).ID’.N2’}_KU) =|> State’:=6 /\ N3’:=new() /\ SAPP({U.APP.AS.{{AuthZGrant(RP.AS.UName.A’.N2’.N3’)}
     ↪   _inv(KU)}_KAS.ID’}_KAPP) /\ witness(U,AS,u_as_authzgrant,AuthZGrant) /\ witness(U,AS,u_as_n3,N3’)
  4. State=6 /\ RRP({{Resource(URI)}_inv(KRP)}_KU) =|> State’:=8
end role

role relyingParty (U,RP,AS,APP,RS : agent, KRP,KAS,KAPP,KRS : public_key, SU,RU,SAPP,RAPP : channel(dy), Resource,PROFILE : hash_func,
  AuthZReq,AuthZResp : message)
played_by RP def=
  local
    State,N5 : nat, ID,UName,URI : text, KU : public_key, A : text set
    const rs_rp_profile,rp_rs_n5,u_rp_resource,app_rp_authzresp : protocol_id
  init
    State:=1
  transition
  1. State=1 /\ RU({KU’.U.RP.URI’}_KRP) =|>        State’:=3 /\ ID’:=new() /\ A’:=new() /\ SAPP({RP.APP.AS.RS.{AuthZReq(ID.RP.AS.RS.A’)}_inv(KRP)}_KAPP)
  2. State=3 /\ RAPP({APP.RP.{AuthZResp({RP.{PROFILE(A’).N5’}_inv(RS).ID}_KRP)}_inv(KAPP)}_KRP) =|> State’:=5 /\ SU({{Resource(URI)}_inv(RP)}_KU) /\ request(RP,RS,
     ↪   rp_rs_n5,N5’) /\ request(RP,RS,rs_rp_profile,PROFILE) /\ secret(PROFILE,rs_rp_profile,{RP,RS}) /\ secret(Resource(URI),u_rp_resource,{U,RP}) /\ secret(
     ↪   AuthZResp,app_rp_authzresp,{APP,RP}) /\ request(APP,RP,app_rp_authzresp,AuthZResp)
end role

role authZServer (U,RP,AS,APP,RS : agent, KU,KRP,KAS,KAPP,KRS : public_key, SU,RU,SAPP,RAPP : channel(dy), Resource,AuthZGrant,AccessToken : hash_func,
  AuthZReq : message)
played_by AS def=
  local
    ID,UName,Pass,LogInURI : text, State,N1,N2,N3,N4 : nat, A : text set
    const u_as_n1,u_as_uname,u_as_pass,u_as_authzgrant,u_as_n3,as_rs_accesstoken,as_rs_n4        : protocol_id
  init
    State:=7
  transition
  1. State=7 /\ RAPP({APP.AS.RS.{AuthZReq(ID’.RP.AS.RS.A’)}_inv(KAPP)}_KAS) =|> State’:=9 /\ N1’ := new() /\ SU({AS.U.Resource(LogInURI).N1’.ID’.A’}_KU) /\ witness(
     ↪   AS,U,u_as_n1,N1’)
  2. State=9 /\ RU({U.AS.{UName’.Pass’}_inv(KU).N1’.ID’.A’}_KAS) =|> State’:=11 /\ N2’:=new() /\ SU({AS.U.APP.{AuthZReq(ID’.RP.AS.RS.A’)}_inv(KAS).ID.N2’}_KU) /\
     ↪   secret(N1’,u_as_n1,{U,AS}) /\ request(AS,U,u_as_uname,UName’) /\ request(AS,U,u_as_pass,Pass’)
  3. State=11 /\ RAPP({APP.AS.{{AuthZGrant(RP.AS.UName’.A’.N2’.N3’)}_inv(KU)}_KAS.ID’}_KAS) =|> State’:=13 /\ N4’:=new() /\ SAPP({AS.APP.RS.{{AccessToken(AS.RP.RS.
     ↪   UName’.A’.N4’)}_inv(KAS).ID’}_KRS}_KAPP) /\ request(AS,U,u_as_authzgrant,AuthZGrant) /\ request(AS,U,u_as_n3,N3’) /\ witness(AS,RS,as_rs_accesstoken,
     ↪   AccessToken) /\ witness(AS,RS,as_rs_n4,N4’) /\ secret(AuthZGrant,u_as_authzgrant,{U,AS})
end role

role resourceServer (U,RP,AS,APP,RS : agent, KU,KRP,KAS,KAPP,KRS : public_key, SAPP,RAPP        : channel(dy), AccessToken,PROFILE : hash_func)
played_by RP def=
  local
    State,N4,N5 : nat, ID,UName : text, A : text set const as_rs_accesstoken,rs_rp_profile,as_rs_n4,rp_rs_n5 : protocol_id
  init
    State:=15
   transition
  1. State=15 /\ RAPP({APP.RS.{{AccessToken(AS.RP.RS.UName’.A’.N4’)}_inv(KAS).ID’}_KRS}_KRS) =|>        State’:=17 /\ N5’:=new() /\ SAPP({RS.APP.RP.{RP.{PROFILE(A’).
     ↪   N5’}_inv(KRS).ID’}_KRP}_KAPP) /\ request(AS,RS,as_rs_accesstoken,AccessToken) /\ secret(AccessToken,as_rs_accesstoken,{AS,RS}) /\ witness(RS,RP,
     ↪   rs_rp_profile,PROFILE) /\ request(RS,AS,as_rs_n4,N4’) /\ witness(RS,RP,rp_rs_n5,N5’)
end role

role application (U,RP,AS,APP,RS : agent, KU,KRP,KAS,KAPP,KRS : public_key, SU,RU,SRP,RRP,SAS,RAS,SRS,RRS        : channel(dy), AccessToken,AuthZGrant,PROFILE :
   ↪   hash_func,
  AuthZReq,AuthZResp : message)
played_by APP def=
  local
    State,N2,N3,N4,N5 : nat, ID,UName : text, A : text set const app_rp_authzresp : protocol_id
  init
    State:=10
  transition
  1. State=10 /\ RRP({RP.APP.AS.RS.{AuthZReq(ID.RP.AS.RS.A’)}_inv(KRP)}_KAPP) =|>        State’:=12 /\ SAS({APP.AS.RS.{AuthZReq(ID.RP.AS.RS.A’)}_inv(KAPP)}_KAS)
  2. State=12 /\ RU({U.APP.AS.{{AuthZGrant(RP.AS.UName’.A’.N2’.N3’)}_inv(KU)}_KAS.ID’}_KAPP) =|>        State’:=14 /\ SAS({APP.AS.{{AuthZGrant(RP.AS.UName’.A’.N2’.N3
     ↪   ’)}_inv(KU)}_KAS.ID}_KAS)
  3. State=14 /\ RAS({AS.APP.RS.{{AccessToken(AS.RP.RS.UName’.A’.N4’)}_inv(KAS).ID’}_KRS}_KAPP) =|>        State’:=16 /\ SRS({APP.RS.{{AccessToken(AS.RP.RS.UName’.A’
     ↪   .N4’)}_inv(KAS).ID}_KRS}_KRS)
  4. State=16 /\ RRS({RS.APP.RP.{RP.{PROFILE(A’).N5’}_inv(RS).ID’}_KRP}_KAPP) =|>        State’:=18 /\ SRP({APP.RP.{AuthZResp({RP.{PROFILE(A’).N5’}_inv(KRS).ID’}_KRP
     ↪   )}_inv(KAPP)}_KRP) /\ witness(APP,RP,app_rp_authzresp,AuthZResp)
end role

role session (U,AS,RP,APP,RS : agent, KU,KAS,KRP,KAPP,KRS : public_key, Resource,AuthZGrant,PROFILE,AccessToken        : hash_func,
  AuthZReq,AuthZResp : message)
def=
  local SURP,RURP,SUAS,RUAS,SUAPP,RUAPP,SRPU,RRPU,SRPAPP,RRPAPP,SASU,RASU,SASAPP,RASAPP,SRSAPP,RRSAPP,SAPPU,RAPPU,SAPPRP,RAPPRP,SAPPAS,RAPPAS,SAPPRS,RAPPRS: channel(
     ↪   dy)
  composition
    user(U,RP,AS,APP,RS,KU,KRP,KAS,KAPP,SURP,RURP,SUAS,RUAS,SUAPP,RUAPP,Resource,PROFILE,AuthZGrant,AuthZReq)
    /\ relyingParty(U,RP,AS,APP,RS,KRP,KAS,KAPP,KRS,SRPU,RRPU,SRPAPP,RRPAPP,Resource,PROFILE,AuthZReq,AuthZResp)
    /\ authZServer(U,RP,AS,APP,RS,KU,KRP,KAS,KAPP,KRS,SASU,RASU,SASAPP,RRSAPP,Resource,AuthZGrant,AccessToken,AuthZReq)
        /\ resourceServer(U,RP,AS,APP,RS,KU,KRP,KAS,KAPP,KRS,SRSAPP,RRSAPP,AccessToken,PROFILE)
        /\ application(U,RP,AS,APP,RS,KU,KRP,KAS,KAPP,KRS,SAPPU,RAPPU,SAPPRP,RAPPRP,SAPPAS,RAPPAS,SAPPRS,RAPPRS,AccessToken,AuthZGrant,PROFILE,AuthZReq,AuthZResp)
end role

role enviroment()
def=
  const u_as_n1,u_as_n3,u_as_uname,u_as_pass,u_as_authzgrant,rs_rp_profile,rp_rs_n4,u_rp_resource, as_rs_accesstoken,as_rs_n4,rp_rs_n5,app_rp_authzresp: protocol_id,
  u,as,rp,rs,app: agent, ku,kas,krp,krs,kapp,ki: public_key, resource,authzgrant,profile,accesstoken : hash_func,
  authzreq,authzresp : message
  intruder_knowledge={u,rp,krp,ki,inv(ki),as,kas,rs,krs,resource}
  composition
    session(u,as,i,app,rs,ku,kas,ki,kapp,krs,resource,authzgrant,profile,accesstoken,authzreq,authzresp)
    /\ session(u,as,rp,app,rs,ku,kas,krp,kapp,krs,resource,authzgrant,profile,accesstoken,authzreq,authzresp)
end role

goal
secrecy_of rs_rp_profile secrecy_of u_rp_resource secrecy_of u_as_uname secrecy_of u_as_pass secrecy_of u_as_authzgrant secrecy_of as_rs_accesstoken secrecy_of
   ↪   app_rp_authzresp

authentication_on u_as_n1 authentication_on as_rs_n4 authentication_on rp_rs_n5 authentication_on rs_rp_profile authentication_on u_as_uname authentication_on
   ↪   u_as_pass authentication_on as_rs_accesstoken authentication_on u_as_authzgrant authentication_on u_as_n3  authentication_on app_rp_authzresp
end goal

enviroment()
```