Proving a WS-Federation Passive Requestor Profile with a Browser Model

Thomas Groß
IBM Research Division
Rüschlikon, Switzerland
tgr@zurich.ibm.com

Birgit Pfitzmann
IBM Research Division
Rüschlikon, Switzerland
bpf@zurich.ibm.com

Ahmad-Reza Sadeghi
Ruhr-University Bochum
Bochum, Germany
sadeghi@crypto.rub.de

ABSTRACT

Web-based services are an important business area. For usability and cost-effectiveness these services require users to rely only on standard browsers. A representative class of such applications, currently in the focus of many industrial players, is Federated Identity Management (FIM). In this context we are facing challenging problems: on the one hand, the security of the existing FIM protocols (including Microsoft Passport, OASIS SAML, and Liberty) is not yet based on rigorous proofs and has been challenged by several analyses. On the other hand, the existing formal security models and proof methods cannot be applied to browser-based protocols in a straightforward manner since they only consider protocol-aware principals: they assume that the involved principals behave according to the specification of the security protocol unless they are corrupted. Web browsers, in contrast, have predefined features and are unaware of the protocol they are involved in. Based on a generic framework for security proofs of browser-based protocols, we model an important FIM protocol, the WS-Federation Passive Requestor Interop profile. We rigorously prove that the protocol provides authenticity and secure channel establishment in a realistic trust scenario. This constitutes the first rigorous security proof for a browser-based identity federation protocol.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General–Security and protection; K.4.4 [Computers and Society]: Electronic Commerce–Security

General Terms: Security, Theory, Standardization, Verification

Keywords: Web service security, identity federation, single signon, web browser, WS-Federation Passive Requestor profile, WSFPI, security proof of protocols

1. INTRODUCTION

Identity federation is a new technology aiming at linking a user's otherwise distinct identities at several locations. Industry is pursuing the realization of identity federation because this technology promises to strongly reduce user management costs, e.g., the cost of password helpdesks and user registrations as well as deletions. Hence, industry developed several frameworks realizing in particular multi-party authentication protocols for the current generation of access control and user management products. A major factor for the success of identity federation protocols is that they are cost-efficient to deploy and offer an easy entry point to this technology. The optimal answer to this demand are protocols that do not require the deployment of special client software and, therefore, are *zero-footprint*. Protocols only relying on a standard web browser fulfill this requirement with bravura. Thus, browser-based protocols are a crucial cornerstone of industrial proposals and expected to be widely used in the near future.

Related Work. Several concrete and complex browser-based FIM protocols were proposed, starting with Microsoft Passport [21]. The Security Assertion Markup Language (SAML) [23] is the first open standard in this area and the basis for the Liberty Alliance Project [18]. WS-Federation [15] is a proposal linking the Web Services world and the browser world with a joint identity-federation basis for both client types, where the Passive Requestor Profile [16] focuses on the zero-footprint browser case. SAML, Liberty and WS-Federation each propose a browser-based authentication protocol that leverages HTML form POSTs to transfer data. This so-called Browser/POST protocol type aims at establishing a mutually authenticated secure channel with a user.

Research was done on several of the given proposals, where Microsoft Passport was analyzed first. The most detailed analysis of its vulnerabilities [17] also describes some limitations that apply to all browser-based protocols. Vulnerabilities were found in a SAML profile [10] and in one of the original Liberty protocols, the enabled-client protocol [27]. Browser-based identity federation was also under discussion because of its privacy design principles starting with [26] and affected by general discussions of browser-based authentication such as [7]. The research in this area, especially on WS-Federation, is complemented by the analysis of web services security protocols, beginning with [9, 3]; however, this work does not consider browser-based protocols.

So far, we only cited negative results, i.e., vulnerability analyses. Though removable security problems were repaired, it is still desirable to devise security proofs to guarantee the security of the protocols. The first approach at such a positive security result was taken for the WS-Federation Passive Requestor Interop (WSFPI), a Browser/POST profile of WS-Federation, in [11]. However, this proof is built on top-down assumptions about the participating parties, i.e., assumptions made to suit the specific proof, in particular the browser and user, and not on a detailed, reusable model of these parties.

In spite of the focus put on browser-based identity federation by industry and that the security goals of authentication and secure channel establishment are well-studied, there are still no rigorous security proofs for this area based upon a general security model.

SWS'05, November 11, 2005, Fairfax, Virginia, USA.

We believe that browser-based protocols are differing from other proposals in two ways: Firstly, they do not use traditional methods for establishing a mutually authenticated secure channel. Secondly, the standard web browser involved has special properties that are not sufficiently considered by the security protocols under consideration.

Mutually Authenticated Secure Channels. Other security protocols typically establish a mutually authenticated secure channel by performing a key exchange and using the exchanged key to secure the communication. A seminal paper was [22], later revised by [19]. The security of key-exchange protocols was studied using formal methods and proof tools as well as cryptographic methods, where tool-supported proofs mostly were based upon abstractions of cryptographic primitives [5]. Cryptographic proofs of key-exchange and authentication protocols were initiated in [2]. Modeling secure channels by a comparison to ideal secure channels, a technique that we will use for the underlying secure channels below, was introduced in [29, 25, 4]. As the traditional channel establishment is not of central interest here, we refer to the extended version of this paper [13] for a broader overview.

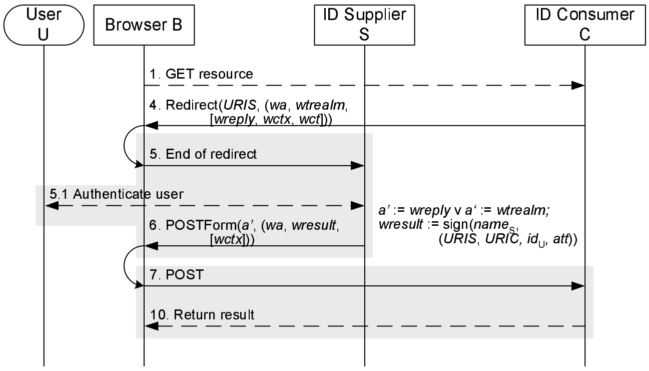
Standard browsers, however, simply do not support most of these key-exchange protocols for which security proofs exist in research. The only exception is mutual authentication through establishing SSL or TLS channels where the browser holds a client certificate. However, this is not considered truly zero-footprint, as users have to install a client certificate in their browser. Also, the client-authentication is bound to the browser itself and not to the user who is actually browsing. Therefore, browser-based protocols usually only leverage server-authenticated secure channels provided by SSL or TLS and establish client authentication by other means. Hence, browser-based protocols are different from all protocols for which prior security proofs exist.

Browser as Protocol-Unaware Principal. A web browser is a party with its own predefined behavior that impacts the security of the protocols executed across it. In usual security protocols, principals are assumed to execute precisely the security protocol under consideration (unless they are corrupted). A browser, in contrast, reacts on a number of predefined messages, adds information to responses automatically, and stores certain information such as histories in places which cannot always be assumed secure.

All this means that a detailed and rigorous browser model is a prerequisite for convincing security proofs of browser-based protocols. Such a model has been proposed in [12]. Given this model, one has to assume that a real browser does *not* perform additional actions, because it seems that for most security protocols arbitrary additional actions could destroy the security.

Our Contribution. As we have seen, there is no security proof of a browser-based FIM protocol yet that is based on a general, reusable browser model, neither with cryptographic techniques nor with formal-methods techniques. In this paper we close this gap. We chose the WSFPI protocol defined in [11] as a strict instantiation of the WS-Federation Passive Requestor Interop scenario [14]. Given the browser model, and the associated user and channel models, the main difficulty lies in the complexity of these protocol participants. Compared with typical protocol-specific principals they have considerable state and react on a large number of messages in a rather asynchronous manner, i.e., not only linearly according to a protocol flow.

Organization. We firstly give an overview of the WSFPI protocol in Section 2. After some notation in Section 3, we describe the machines of the generic browser framework that we leverage for the security proof in Section 4. We complement these protocol-unaware machines with two machines specific for WSFPI, the identity supplier and the identity consumer in Section 5. In Section 6, we describe the protocol setup, the assumptions of WSFPI and the authenticity theorem. Section 7 contains the full proof.



User U
Browser B
ID Supplier S
ID Consumer C
1. GET resource
4. Redirect(*URIS*, (*wa*, *wtrealm*, [*wreply*, *wctx*, *wct*]))
5. End of redirect
5.1 Authenticate user
6. POSTForm(*a'*, (*wa*, *wresult*, [*wctx*]))
$a' := wreply \vee a' := wtrealm;$
$wresult := \mathrm{sign}(name_S, (URIS, URIC, id_U, att))$
7. POST
10. Return result

Figure 1: WSFPI protocol with abstract parameters. Steps with uninterrupted lines are actually specified in the protocol. The gray boxes denote secure channels.

2. OVERVIEW OF WSFPI

In this section, we survey the WSFPI protocol [11], which is a strict instantiation of the interop scenario from [14]. Figure 1 shows the message flow of WSFPI when no error occurs.

Principals. Originally, WSFPI is a three-party authentication protocol, where a identity supplier S provides a security token (in other terminologies a credential or ticket) of a client to another server, called identity consumer C. However, we consider the client of the protocol as two distinguished principals, the user U and its browser B. This distinction stems from the different roles of B and U in the protocol and corresponds to the analysis of [28, 12]: a browser acts as a protocol-unaware party, whereas U supervises the browser-based protocols and knows the real user's trust relationships and identity information. Thus, browser B communicates on behalf of user U with identity supplier S and identity consumer C. Ultimately, user U wants to sign in at identity consumer C using WSFPI. The identity supplier S authenticates U and confirms its identity to identity consumer C by means of a signed SAML assertion.

Message Parameters. $URIC$ is the designated URI of C for accepting security tokens in WSFPI protocol runs. At $URIS$, the identity supplier S authenticates users of WSFPI and issues security tokens. The string id_U is the identity of a user U in the user database of S. The parameter wa names the constant protocol action and version wsignin1.0. The parameter $wtrealm$ names the security realm of C under which the WSFPI protocol is executed; it should equal $URIC$. With $wreply$, C may specify a URI to which S should redirect the browser after issuing a security token for the user. The reply address $wreply$ must be within $wtrealm$. The identity consumer C can transport information through the WSFPI protocol run by the opaque context parameter $wctx$. The security token issued by identity supplier S is contained in parameter $wresult$. The information flow of this token is crucial: if the adversary gets hold of it, the protocol security is broken.

Protocol Flow. Steps 1 and 10 show that user U is assumed to browse at identity consumer C before the protocol and to get some application-level response after the protocol, but they are not actually standardized. Steps 4-5 redirect the browser B to S, the unspecified Step 5.1 authenticates the user to S, and Steps 6-7 essentially redirect B back to C with a SAML assertion signed under the name of S in parameter $wresult$.

The figure contains all the exchanged top-level parameters with their original names. In addition, the most important elements of $wresult$ are shown. For simplicity, we have omitted the time stamps in $wresult$ and the corresponding verifications because they are not necessary for the authenticity property that we show. This may be added in the future. In both abstract messages, Redirect and POSTForm, the first parameter in the figure is the address and the second parameter the payload, here a list of the protocol parameters. Square brackets mean that parameters are optional. The "End of redirect" message gets its parameters from the Redirect message, and the POST message gets them from the payload of the message denoted POSTForm. For this the POSTForm, typically including a script, triggers the browser or user to POST the message described.

3. NOTATION

We use the same notation as [12]. For brevity we do not define the notation in full formality, but only introduce particular constructions and refer to the browser model [12] for further details.

We use a straight font for constants, including constant Sets and Types, functions, and predicates, where Types are predefined constant sets. We denote the set of URL host names, including protocol names such as "https", by URLHost, and the set of URL host and path names by URLHostPath. We write an address $adr \in \mathsf{URLHostPath}$ as a pair $(host, path)$ of a host name $host \in \mathsf{URLHost}$ and a path. The functions host and path extract these two parts from an address. By $host \subseteq host'$ we denote that $host$ lies under the address $host'$ as defined for URLs, e.g., www.ibm.com $\subseteq$ ibm.com. The type $\mathsf{ChType} := \{\mathsf{secure}, \mathsf{insecure}\}$ contains the channel types available. The function ctype, defined on URLHost and URLHostPath, extracts the security type from an address by means of the HTTP protocol declaration ("http" versus "https").

We represent our machines such as the browser model as I/O automata, in other words finite-state machines with additional variables. This is a usual basis for specifying participants in distributed protocols; the first specific use for security is in [20]. Specifically, we use the automata model from [25]. The machine model deletes *volatile* variables any final state.

Machines may have multiple fixed connections to other machines organized by means of input and output *ports*, where input ports are denoted by n? and output ports by n!. The machine model connects input and output ports n? and n! of same name n. A *clock port* $\mathsf{n}^{\triangleleft}!$ schedules the connection between simple ports n? and n! with same name n, i.e., triggers the delivery of messages previously sent through connection n.

We define the state transition function δ_{M} of a machine M using a notation analogous to UML state diagrams [24] (see Figure 2). A *transition* is an arrow from state s to s' with label $Event[Guard]//Action$, where s and s' are the source and destination states, $Event$ is a sequence of non-empty inputs to the input ports, $Guard$ is a predicate over the $Event$ and the machine's current variable allocation V. $Action$ specifies computations and outputs of the transition. We use typed inputs in $Event$s. If a machine obtains an input with another message type than specified it ignores the input. All parameters in input messages are assigned to volatile variables with the same names.

Machines may use a signature scheme and corresponding certificates. We write $m' \leftarrow \mathsf{sign}(id, m)$ for signing a message m under an identity id, and $(id, m) \leftarrow \mathsf{test}(m')$ for verifying a received message and extracting an identity id and a payload m. We denote failure by $(id, m) = (\epsilon, \epsilon)$. The functions include all necessary exchange and verification of keys and certificates. We call the function that looks up the name from a certificate name().

4. BROWSER FRAMEWORK

We briefly describe the framework for security analysis of browser-based protocols introduced in [12]. This framework models generic protocol-unaware parties such as browser and user.

Figure 3 illustrates the automata in the framework. The machines C and S depicted in a hatched pattern are specific to WSFPI and described in Section 5; the other machines are generic for the whole browser-based protocol class. The solid arrows depict direct connections between two machines through which messaages are transfered confidentially. All information flow to the adversary is handled explicitly as imperfections; specifically all ports *not* connected to an honest machine are connected to the adversary. The dashed arrows depict clock ports that schedule the corresponding connections.

Machine B described in Section 4.1 models a web browser. It involves all the capabilities of web browsers used in browser-based security protocols and includes their typical messages and their *imperfections* such as information flow to other parties. Machine U models the knowledge of a user and his or her capabilities to handle initial user authentications. It also enforces constraints about a user without which every browser-based protocol must fail, e.g., that the user only accepts certificates that are compliant with its trust relationships, or observes the channel status, especially step-downs from secure to insecure channels. The user machine U is modeled in detail in [12]; here we only stress that U stores the addresses of at least one identity supplier S trusted by U and the login information to authenticate at S.

Machine secchan models the channel types used in browser-based protocols. The user interface between between the user and browser is modelled by the ports $\mathsf{gui}_{\mathsf{U,B}}?$ and $\mathsf{gui}_{\mathsf{B,U}}!$; the ports channel_in? and channel_out! connect the channel machine to other machines.

4.1 Browser Machine B

A web browser acts as the client in the Hypertext Transfer Protocol (HTTP) [6] and renders protocol state and payloads to its user. A browser acts on behalf of one single user in a browsing session. However, the browser may display multiple windows that render different HTTP transactions in parallel. We omit the window management of the browser machine for brevity. A real browser accepts inputs from the user specifying addresses to retrieve and renders the content associated with such addresses, as well as the status of the channel to the server and the identity of the server. The browser also initiates dialogs with the user to negotiate changes of the channel state, verify a server's authentication or request for user authentication. We model the interface towards a user such that the user has the same information as in the real world. Thus, the interface not only contains content and error messages, but also information about the channel state and the address of the server. Normally the browser always displays the server's hostname. Yet this is a certified server identity only if secure channels are used.

In order to initiate an HTTP transaction, a web browser establishes a connection to a server specified by the address to access. Having established a channel, the browser issues an HTTP request to the server. Such a request specifies the resource that the browser intends to retrieve, but may also contain additional data and parameters. The server evaluates the request and issues a response using the same channel.

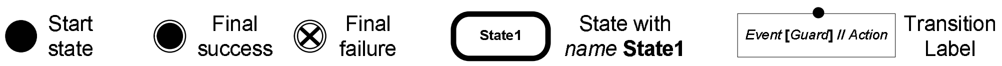


Figure 2: Key to the state diagrams

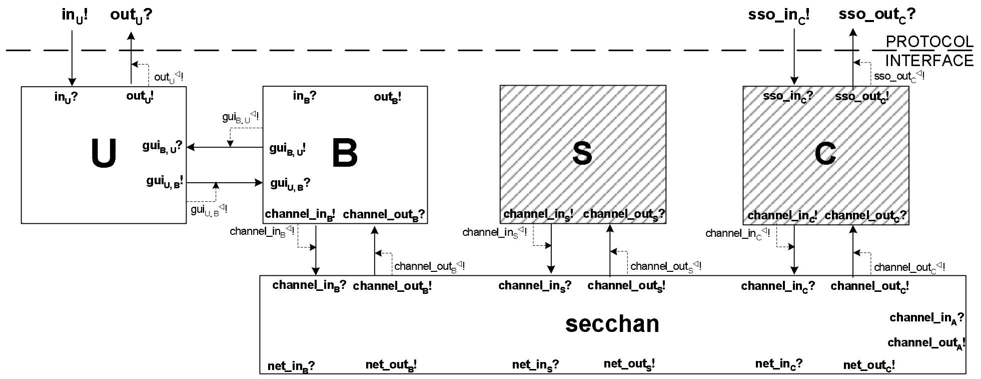


Figure 3: System architecture for browser-based protocols, here for identity federation protocols.

HTTP transactions may implement various functions. Clients can retrieve data by a GET request, and send data by a POST request. Here, the POST request is most important, as Browser/POST protocols use POST messages to transfer security tokens to servers. Servers may not only deliver content but also direct the browser to a behavior change by issuing executable scripts and error messages. We focus on HTTP responses with scripted form POST, which direct the browser to another address of the server's choice and constitute the main message of WSFPI. The framework [12] models major browser messages as abstract formats, however, we focus on the subset of message and parameter types that is used in the security-relevant core flow of WSFPI.

Handling of JavaScript. As all browser-based protocols must fail if the browser is compromised, we consider the browser as an honest party. However, real browsers are able to execute arbitrary code issued by other machines, e.g., by means of JavaScripts. As an adversary can insert JavaScripts into messages transmitted over insecure channels, this may compromise the browser behavior. The browser model [12] handles this fact by not modeling generic active content but only allowing *scripted form posts*[1], which are also used in WSFPI. We briefly discuss why this approach is reasonable. Firstly, in a real browser JavaScript may indeed be deactivated. Then the user issues the submit command itself instead leaving it to the JavaScript; thus the WSFPI protocol can still be executed. Secondly, current real browsers are already capable of filtering JavaScript commands syntactically. One may standardize a white list of scripted post commands that may pass such filters, whereas all other JavaScripts are not executed. Thirdly, specifically for WSFPI the JavaScript is issued by a trusted party over a secure channel (Step 6 in Figure 1). Therefore, more elaborate policies for browser JavaScript handling may only accept JavaScript from such parties.

Abstract Browser Messages. Browser/POST profiles mainly rely on scripted form POSTs. We shortly define the corresponding abstract message types. In WSFPI the identity supplier S issues an HTTP response to the browser B containing an HTML form with a security token enhanced by an active script that directs B to submit this form to another server. This results in an HTTP POST request issued by B to a server specified in the form.

We first consider the server's response POSTForm. An abstract message $\mathsf{POSTForm}(adr : \mathsf{URLHost}, path : \mathsf{URLPath}, query : \Sigma^*, close : \mathsf{Bool}, nocache : \mathsf{Bool})$ models a form containing a script that will POST a message whose body encodes the abstract $query$ to the address $adr/path$. The parameters $close$ and $nocache$ model the connection and cache-response directives of HTTP1.1, where $nocache = \mathsf{TRUE}$ states that an HTTP no-cache and no-store directive is explicitly set [6, Section 14.9]. In consequence of the POSTForm message the browser establishes a channel to the address adr and then sends $path$ and $query$ by a POST message over that channel. The channel type is implied by the HTTP protocol name “http” or “https” in adr. We specify such an abstract POST message as follows: $\mathsf{POST}(path : \mathsf{URLPath}, query : \Sigma^*, login : \Sigma^*, info\_leak : (\Sigma^* \times \Sigma^*)^*)$. The parameter $path$ contains the path to be retrieved. The $query$ is encoded in the body of the POST. If the POST message is issued as result of a preceding POSTForm message, then $query$ parameter of the POSTForm constitutes the body of the resulting POST message. The parameter $login$ contains the credentials of a password-based user authentication, including the account name. The parameter $info\_leak$ is a list of name-value pairs. It models that web browsers include additional data into the request and generate an information flow to the server, e.g., the preceding address in the HTTP Referer tag. We explain the details of such information flow below.

Important Browser Variables.

We briefly discuss the variables that are most relevant for Browser/POST protocols and for the proof of Browser/POST single sign-on protocols respectively.

The persistent variable $prev\_run = (ch\_type, adr, form)$ stores information about the preceding HTTP transaction of the given browser window. Its element ch_type contains the channel type, whereas adr contains the address retrieved in the preceding HTTP transaction. The element $form$ is of special interest in WSFPI, as it holds the structure of an HTML form together with hidden value fields already included in the form between two HTTP transactions. Thus, this variable holds the form

[1] Browser-based protocols use HTTP forms to transfer data from one server to another by a method called scripted form POST. The issuer of the data includes the data in hidden form fields of its HTTP response and has the browser submit the form by means of a JavaScript. Upon such a submit command the browser issues an HTTP POST containing the data to the receiving server.

transmitted in a scripted POST and is the source of information flow of confidential information in this form such as a security token. The volatile variable $form$ contains an HTML form obtained in a submit_form message and is related to the persistent variable $prev\_run.form$. $History$ and $Cache$ are two persistent variables that hold the browser's URL and page trail and may potentially flow to the adversary.

Information Flow. A browser may leak information to the underlying operating system as well as to the servers it communicates with. We discuss the impact of the browser imperfections introduced in [12] on Browser/POST protocols. When browser B issues a POST request, it includes additional information apart from the payload posted within the parameter $info\_leak$. This additional information is specified by a function leak2server of the browser variables V_{B} which especially includes the Referer, i.e., the URL accessed in the preceding HTTP transaction into the POST request, i.e., in the current model $\mathsf{leak2server}(V_{\mathsf{B}}) = prev\_run.adr$. We also consider the information that flows to the local browser cache, i.e., to the persistent variable $Cache$. The cache information flows introduce security and privacy risks to WSFPI in insecure environments such as kiosk scenarios. We capture this imperfection by letting the adversary query the browser machine for this information: upon a do_leak command at port $\mathsf{in_B}?$, the browser outputs its persistent state, including $History$ and $Cache$, at $\mathsf{out_B}!$ to the adversary. Browser/POST protocols must ensure that servers issuing scripted form POSTs correctly set a nostore cache directive to prevent information flow of the security token to the adversary.

Browser Behavior. We describe the behavior of a browser in a Browser/POST case such as WSFPI and focus on specific parts most relevant for the concrete protocol. Therefore, we start with a browser B in State Await_response depicted in Figure 4. In this state, browser B has already established a channel to a server, sent an HTTP request and now waits for an HTTP response from a server. This is the case before WSFPI Step 6 of Figure 1. In WSFPI the identity supplier issues a POSTForm message that directs B to POST a form to another server. Upon such a message, the browser enters the State Active_FormPOST. B adds the current address to the persistent variable $History$ and upon a negative $nocache$ statement the form to the persistent variable $Cache$. Then, the browser may close the channel depending on the variable $close$. The browser issues a message $(\mathsf{submit\_form}, wid, \epsilon, ra)$ to itself in order to trigger the scriped form post.

Upon a $(\mathsf{submit\_form}, wid, form\_in, ra)$ message the browser B starts a new HTTP transaction with the goal to submit the form in persistent variable $prev\_run.form$ to the address specified by adr. We depict the start of this flow in Figure 5. Firstly, B checks whether the form data provided by U in the variable $form\_in$ match the given form. Then, B checks whether a user notification is necessary because of a channel status change. In our concrete example of a WSPFI protocol run, the browser came from a secure channel in Step 6 and is about to open a new secure channel. Therefore, B won't notify the user machine and enters State Connect_confirmed. If the browser does not hold a secure channel instance to the hostname of adr, B will enter the State New_channel_required.

We focus on the case of WSFPI that B will now establish a new secure channel with the machine corresponding to the address adr. We depict this part of the state machine of B in Figure 6. Browser B requests a new secure channel from the channel machine secchan by a message $(\mathsf{new}, host, \mathsf{secure})$. Given a successful channel establishment the browser obtains a message $(\mathsf{accepted}, cid, host', sid)$ and enters State Channel_established. Now, B issues a POST message to the channel by a message $\mathsf{POST}(path, form, login, info\_leak)$ and enters again State Await_response. We depict this in Figure 4.

5. PROTOCOL MODEL

We describe the protocol-aware machines of the WFSPI identity supplier S and identity consumer C that interact with the generic machines for browser-based protocols, recall Figure 3.

5.1 Identity Supplier S

An identity supplier S vouches for a user identity authenticated by an initial password-based user authentication. We describe the machine by its variables and its behavior. For the user authentication in Step 5.1 of WSFPI (see Figure 1), this machine S uses the user authentication machine of [12], proven to provide authenticity for browser channels via password-based user authentication.

We describe the persistent variables of S here shortly, the the extended version of this paper [13] contains an aditional overview. Usually, these variables are set during the protocol setup, which we discuss in Section 6.1. Firstly, S contains a user database $MetaU_{\mathsf{S}}$, which contains pairs of user identities and login information for the initial password-based user authentication. Secondly, S maintains a database $MetaC_{\mathsf{S}}$ of identity consumers that it knows. An entry of $MetaC_{\mathsf{S}}$ consists of the URI $URIC$ where the identity consumer receives security tokens and a list att_n of names of attribute that S sends to this identity consumer. Thirdly, S has persistent variables containing its own addresses and identity data. The variable $URIS$ contains the URI of the identity supplier service at S, while $host_{\mathsf{S}}$ constrains the hostname of S. The variable sid_{S} contains the identity of S for secure channels, whereas $name_{\mathsf{S}}$ is the name under which S signs security tokens.

We now describe the state diagram of S depicted in Figure 7. We derived it from the protocol definition of WSFPI made in [11] and sketched in Figure 1. S is active in Steps 5 to 6 of Figure 1. For brevity, we start with the State Secure_channel_accepted, in which S has finished the channel establishment for a secure channel with another party (presumably a browser), because channel establishment is always equal. Let cid be the identity of this channel. Upon obtaining a message $(\mathsf{receive}, cid, \mathsf{GET}(adr, path, query, login, info\_leak))$ at $\mathsf{channel\_out_S}?$, the identity supplier tests whether the message arrived at the correct $URIS$. If so, S parses the $query$ string of the GET request and enters State Request_received. If the parameters of the request pass the security tests of WSFPI, the identity supplier enters State Requested_UAuth. These security tests mainly control whether $wtrealm$ named in the GET request belongs to an identity consumer known in $MetaC_{\mathsf{S}}$, and whether the potential second address $wreply$ lies in this realm. The identity supplier then starts the sub-protocol uauth, which is described in detail in [12]. This sub-protocol handles the user authentication that corresponds to Step 5.1 in the WSFPI protocol run of Figure 1. Upon a successful user authentication the identity supplier S obtains a message $(\mathsf{done}, cid, idu)$ at $\mathsf{uauth\_out?_S}$ with $idu \neq \epsilon$.

Given that message, S generates and signs a security token $wresult$, which contains its own $URIS$ as issuer identifier, $wtrealm$ of the corresponding identity consumer, the user identity idu obtained in the user authentication, and attributes of this user retrieved from a database. The syntax of the signed message, a SAML token, is abstracted by a function gen_token with a corresponding function parse_token for retrieving the parameters. Then the identity supplier enters State UAuth_success. There it selects the redirect address $padr$ from $wtrealm$ and $wresult$. S relies on function ctype to test if this address will result in the establishment of a secure channel. If so it issues a POSTForm message contain-

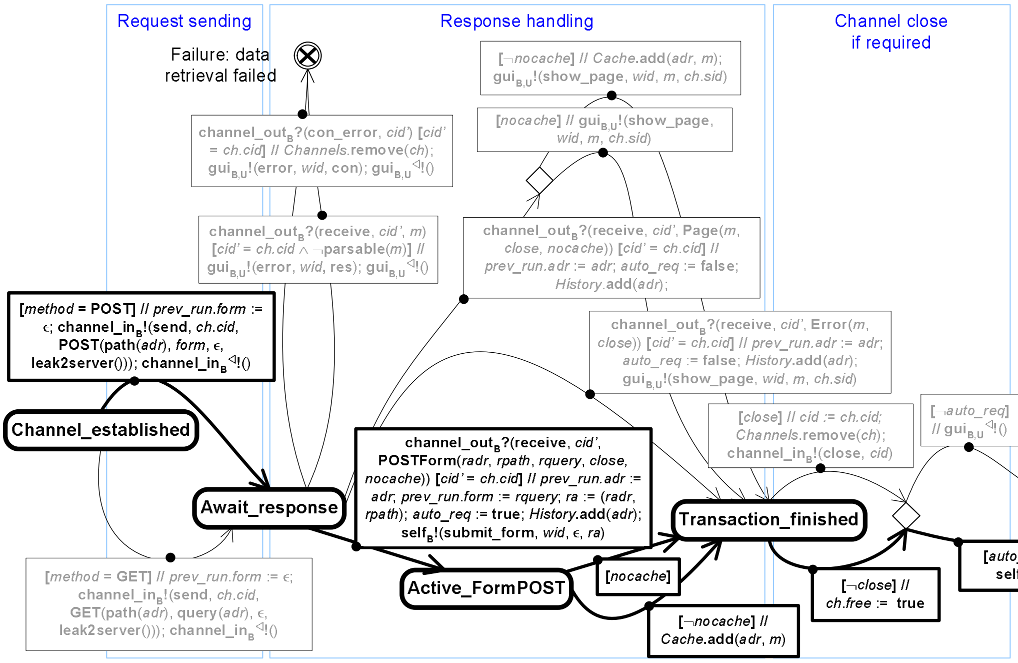

Request sending
Response handling
Channel close if required
Failure: data retrieval failed
channel_out$_B$?(con_error, cid') [cid' = ch.cid] // Channels.remove(ch); gui$_{B,U}$!(error, wid, con); gui$_{B,U}$$^{\lhd}$!()
channel_out$_B$?(receive, cid', m) [cid' = ch.cid ∧ ¬parsable(m)] // gui$_{B,U}$!(error, wid, res); gui$_{B,U}$$^{\lhd}$!()
[¬nocache] // Cache.add(adr, m); gui$_{B,U}$!(show_page, wid, m, ch.sid)
[nocache] // gui$_{B,U}$!(show_page, wid, m, ch.sid)
channel_out$_B$?(receive, cid', Page(m, close, nocache)) [cid' = ch.cid] // prev_run.adr := adr; auto_req := false; History.add(adr);
channel_out$_B$?(receive, cid', Error(m, close)) [cid' = ch.cid] // prev_run.adr := adr; auto_req := false; History.add(adr); gui$_{B,U}$!(show_page, wid, m, ch.sid)
[method = POST] // prev_run.form := ϵ; channel_in$_B$!(send, ch.cid, POST(path(adr), form, ϵ, leak2server())); channel_in$_B$$^{\lhd}$!()
Channel_established
Await_response
channel_out$_B$?(receive, cid', POSTForm(radr, rpath, rquery, close, nocache)) [cid' = ch.cid] // prev_run.adr := adr; prev_run.form := rquery; ra := (radr, rpath); auto_req := true; History.add(adr); self$_B$!(submit_form, wid, ϵ, ra)
[close] // cid := ch.cid; Channels.remove(ch); channel_in$_B$!(close, cid)
[¬auto_req] // gui$_{B,U}$$^{\lhd}$!()
Proceed: transaction successful
Transaction_finished
Active_FormPOST
[nocache]
[¬nocache] // Cache.add(adr, m)
[¬close] // ch.free := true
[auto_req] // self$_B$$^{\lhd}$!()
[method = GET] // prev_run.form := ϵ; channel_in$_B$!(send, ch.cid, GET(path(adr), query(adr), ϵ, leak2server())); channel_in$_B$$^{\lhd}$!()

Figure 4: Request handling phase of an HTTP transaction.

ing the token $wresult$ over the same secure channel with identifier cid. This message corresponds to Step 6 of the WSFPI protocol run in Figure 1.

5.2 Identity Consumer C

An identity consumer C authenticates a user at a secure channel by means of the WSFPI protocol. We designed the machine C according to the protocol definition of WSFPI [11]. It has an input port sso_in$_\mathsf{C}$? where it accepts a message (start, cid). By this message, a higher-level protocol initiates a single sign-on authentication for the channel with identifier cid. The identity consumer C outputs $(\mathsf{accepted}, cid, idu, att)$ at out-port sso_out$_\mathsf{C}$! if a single sign-on authentication finished successfully.

The persistent variables of C define the federation the identity consumer is part of. The variable $nameS$ contains the name under which C's identity supplier signs security tokens. The variable $URIS$ is the URL where the identity supplier handles runs of WSFPI and to which C therefore redirects browsers. $URIC$ is the identity consumer's own URI where it accepts security tokens from this protocol.

The behavior of C consists of two phases. In the first phase it initiates a single sign-on protocol run by redirecting a user's browser to C's identity supplier. This corresponds to Step 4 of Figure 1 and is of no particular interest here. The phase in Figure 8 handles the acceptance procedure for a security token, i.e., Step 7 of Figure 1. We start with State Secure_channel_accepted, in which the identity consumer has accepted the establishment of a secure channel with channel identifier cid. Upon a message $(\mathsf{receive}, cid, \mathsf{POST}(path, req, \epsilon, *))$ at port channel_out$_\mathsf{C}$?, the identity consumer parses the request req and enters State Request_received. Then it runs a sequence of security tests prescribed in WSFPI. It enters State Signature_valid if the signature on the security token $wresult$ is valid and under the $nameS$ of its identity supplier. Finally, the identity consumer tests whether the security token $wresult$ names its identity supplier as issuer, whether the identity consumer itself is the audience of the security token, and whether the user identity idu in the token is not empty. Only then the identity consumer outputs $(\mathsf{accepted}, cid, idu, att)$ at sso_out$_\mathsf{C}$!.

6. SECURITY OF WSFPI

In this section define the security of the WSFPI single sign-on protocol. We define the necessary assumptions and identity federation set-up. We present the authenticity theorem.

6.1 Assumptions

We need assumptions about the honesty of certain participants (including correctness of their software), about the set-up for the protocol, and about the signature scheme used. Furthermore, we have to assume that WSFPI is to some extent separated from other protocols, more precisely, that an identity supplier does not reuse its signing key for tokens in completely arbitrary other ways. In classical cryptographic protocol analyses, such separation assumptions are implicit in the definition of the protocol machines, but in a realistic web services scenario, signing keys are not necessarily assigned to only one protocol.

Definition 1. (Signature-Related Assumptions) a. The signature scheme used is secure against adaptive chosen-message attacks [8], and this property is lifted to the name-based version described here via secure certification, i.e., for every correct identity supplier S nobody except S can sign any new message m under $name_\mathsf{S}$.

b. The signature scheme is memory-less or counter-based in the sense of [1].

c. A correct S uses the secret key corresponding to $name_\mathsf{S}$ in a way that excludes cross-protocol attacks, i.e., no other application at S uses this key to sign messages that pass the verification in State Signature_valid of identity consumer C.[2]

[2] The exclusion of cross-protocol attacks does not necessarily imply the the signature key is exclusively used for WSFPI; only that no

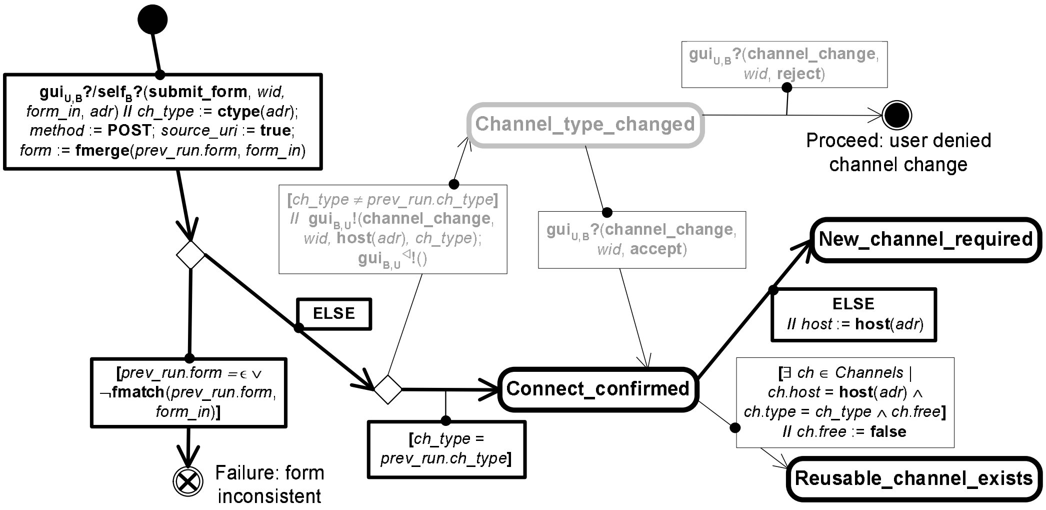

gui$_{U,B}$?/self$_B$?(submit_form, wid, form_in, adr) // ch_type := ctype(adr); method := POST; source_uri := true; form := fmerge(prev_run.form, form_in)
gui$_{U,B}$?(channel_change, wid, reject)
Channel_type_changed
Proceed: user denied channel change
[ch_type ≠ prev_run.ch_type] // gui$_{B,U}$!(channel_change, wid, host(adr), ch_type); gui$_{B,U}^{\triangleleft}$!()
gui$_{U,B}$?(channel_change, wid, accept)
New_channel_required
ELSE
ELSE // host := host(adr)
[prev_run.form = ϵ ∨ ¬fmatch(prev_run.form, form_in)]
Connect_confirmed
[∃ ch ∈ Channels | ch.host = host(adr) ∧ ch.type = ch_type ∧ ch.free] // ch.free := false
[ch_type = prev_run.ch_type]
Failure: form inconsistent
Reusable_channel_exists

Figure 5: Request initiation and local negotiation phase of B.

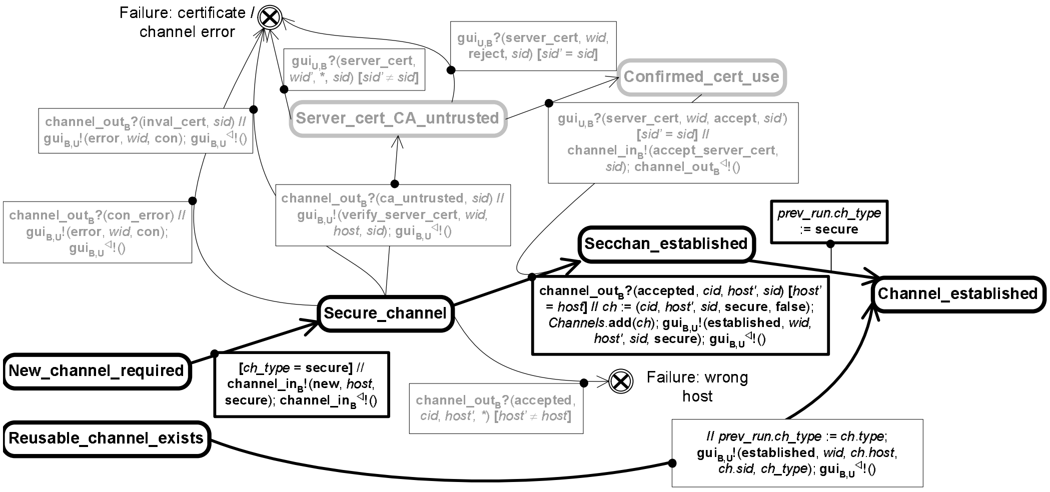

Failure: certificate / channel error
gui$_{U,B}$?(server_cert, wid, reject, sid) [sid' = sid]
gui$_{U,B}$?(server_cert, wid', *, sid) [sid' ≠ sid]
Confirmed_cert_use
channel_out$_B$?(inval_cert, sid) // gui$_{B,U}$!(error, wid, con); gui$_{B,U}^{\triangleleft}$!()
Server_cert_CA_untrusted
gui$_{U,B}$?(server_cert, wid, accept, sid') [sid' = sid] // channel_in$_B$!(accept_server_cert, sid); channel_out$_B^{\triangleleft}$!()
channel_out$_B$?(con_error) // gui$_{B,U}$!(error, wid, con); gui$_{B,U}^{\triangleleft}$!()
channel_out$_B$?(ca_untrusted, sid) // gui$_{B,U}$!(verify_server_cert, wid, host, sid); gui$_{B,U}^{\triangleleft}$!()
prev_run.ch_type := secure
Secchan_established
Channel_established
Secure_channel
channel_out$_B$?(accepted, cid, host', sid) [host' = host] // ch := (cid, host', sid, secure, false); Channels.add(ch); gui$_{B,U}$!(established, wid, host', sid, secure); gui$_{B,U}^{\triangleleft}$!()
New_channel_required
[ch_type = secure] // channel_in$_B$!(new, host, secure); channel_in$_B^{\triangleleft}$!()
Failure: wrong host
channel_out$_B$?(accepted, cid, host', *) [host' ≠ host]
Reusable_channel_exists
// prev_run.ch_type := ch.type; gui$_{B,U}$!(established, wid, ch.host, ch.sid, ch_type); gui$_{B,U}^{\triangleleft}$!()

Figure 6: Channel establishment phase of B.

d. Signatures with a fixed public key are of fixed length, the function $\mathsf{length}(\mathsf{gen\_res}(\cdot,\cdot,\cdot))$ does not leak information about its parameters beyond their lengths, and similarly for $\mathsf{length}(\mathsf{POSTForm}(\ldots))$ and $\mathsf{length}(\mathsf{POST}(\ldots))$.

We need Part b. in order to use a result from [1] that the signature scheme is then reactively secure, which essentially means that the adversary cannot guess signatures that have been made but that it has not seen. In contrast, normal signature security only considers forging signatures on messages that the signer has not signed at all.

The WSFPI protocol, like most federated identity management protocols, assumes that certain information has been exchanged in advance. In Definition 2 we formulate this as relations about the persistent variables of users, identity suppliers, and identity consumers. There is no specific protocol for the set-up, and it may occur for different participants at different times. As set-up for a particular user machine U and identity supplier S, they exchange login information $login_{\mathsf{U},\mathsf{S}} \neq \epsilon$ such that U and S are the only parties that obtain information about it. Furthermore, U must know a valid server identity sid_{S} of S so that it can verify later that it has a secure channel to S.

As set-up for a federation between an identity consumer C and its identity supplier S, the identity consumer stores the URI $URIS$ at which S handles the WSFPI protocol. It receives the certificate $certS$ that the identity supplier uses for signing security tokens in WSFPI, and sets the $nameS$ to the name from this certificate. The identity supplier stores $URIC$ of C in its table $MetaC_{\mathsf{S}}$ together with the names of attributes att_n that S transfers to C about users. Finally, we require that S and C really own their supposed URIs and server identities. In practice, this corresponds to correct certification for SSL channels; this is represented by the binding table CA of the secure channel machine. This table binds machine identifiers to certified server identities and host names of security domains. Formally, the result of the set-up is this:

Definition 2. (WSFPI Setup Assumptions) We say that a user U, an identity supplier S, and an identity consumer C fulfill the WSFPI setup assumptions (at a certain point in time) if the following holds for these machines and the channel machine secchan:

a. The user's table T_{U} of trusted servers contains an en-

other SAML assertions with WSFPI-compatible format are signed with the very same key.

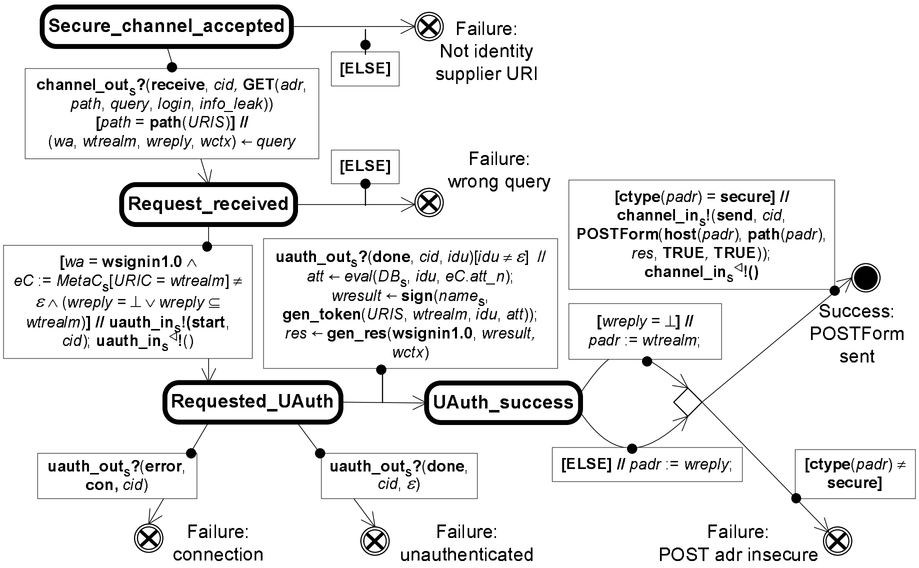

Secure_channel_accepted
[ELSE]
Failure: Not identity supplier URI
channel_out_S?(receive, cid, GET(adr, path, query, login, info_leak)) [path = path(URIS)] // (wa, wtrealm, wreply, wctx) ← query
[ELSE]
Failure: wrong query
Request_received
[wa = wsignin1.0 ∧ eC := MetaC_S[URIC = wtrealm] ≠ ε ∧ (wreply = ⊥ ∨ wreply ⊆ wtrealm)] // uauth_in_S!(start, cid); uauth_in_S^◁!()
uauth_out_S?(done, cid, idu)[idu ≠ ε] // att ← eval(DB_S, idu, eC.att_n); wresult ← sign(name_S, gen_token(URIS, wtrealm, idu, att)); res ← gen_res(wsignin1.0, wresult, wctx)
[ctype(padr) = secure] // channel_in_S!(send, cid, POSTForm(host(padr), path(padr), res, TRUE, TRUE)); channel_in_S^◁!()
[wreply = ⊥] // padr := wtrealm;
Success: POSTForm sent
Requested_UAuth
UAuth_success
[ELSE] // padr := wreply;
[ctype(padr) ≠ secure]
uauth_out_S?(error, con, cid)
uauth_out_S?(done, cid, ε)
Failure: connection
Failure: unauthenticated
Failure: POST adr insecure

Figure 7: Main part of the state machine of the identity supplier S.

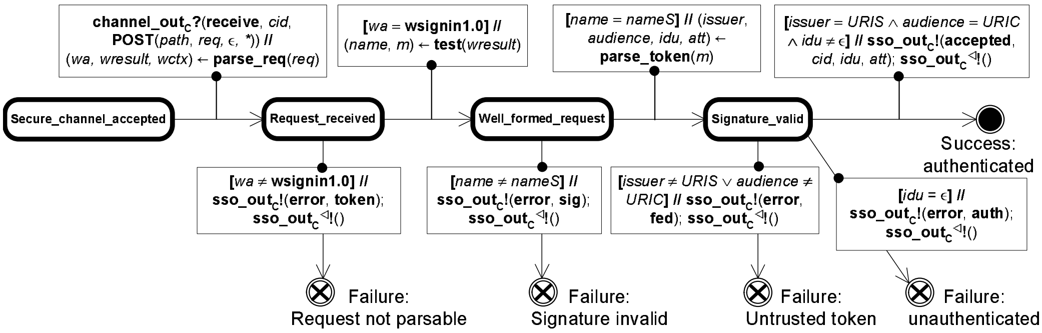

channel_out_C?(receive, cid, POST(path, req, ε, *)) // (wa, wresult, wctx) ← parse_req(req)
[wa = wsignin1.0] // (name, m) ← test(wresult)
[name = nameS] // (issuer, audience, idu, att) ← parse_token(m)
[issuer = URIS ∧ audience = URIC ∧ idu ≠ ε] // sso_out_C!(accepted, cid, idu, att); sso_out_C^◁!()
Secure_channel_accepted
Request_received
Well_formed_request
Signature_valid
Success: authenticated
[wa ≠ wsignin1.0] // sso_out_C!(error, token); sso_out_C^◁!()
[name ≠ nameS] // sso_out_C!(error, sig); sso_out_C^◁!()
[issuer ≠ URIS ∨ audience ≠ URIC] // sso_out_C!(error, fed); sso_out_C^◁!()
[idu = ε] // sso_out_C!(error, auth); sso_out_C^◁!()
Failure: Request not parsable
Failure: Signature invalid
Failure: Untrusted token
Failure: unauthenticated

Figure 8: Main part of state machine of the identity consumer C.

try $(hostS, sidS, login_{\mathsf{U},\mathsf{S}}, \{\mathsf{secure}\})$ where $hostS = host_\mathsf{S}$ and $sidS = sid_\mathsf{S}$ for the host and identity variables of S, and where the identity supplier's user table $MetaU_\mathsf{S}$ contains an entry $(id_\mathsf{U}, login_{\mathsf{U},\mathsf{S}})$. The user identity id_U is unique in $MetaU_\mathsf{S}$.

b. No other variables contain information about $login_{\mathsf{U},\mathsf{S}}$.

c. $URIS = URI_\mathsf{S} \wedge certS = cert_\mathsf{S}$ (where the left sides are variables of C and the right sides variables of S). Furthermore, $name_\mathsf{S} = \mathsf{name}(cert_\mathsf{S})$ and $nameS = \mathsf{name}(certS)$ (and thus also $nameS = name_\mathsf{S}$).

d. The table $MetaC_\mathsf{S}$ contains an entry $(URIC, att\_n)$ where $URIC$ equals the variable $URIC$ of C.

e. The address $URIC$ of C defines a *WSFPI security domain*, i.e., incoming messages at all addresses $URI' \subseteq URIC$ are treated according to the specification of C or deleted.[3]

f. The binding table CA of secchan contains an entry $(\mathsf{C}, sid_\mathsf{C}, host_\mathsf{C})$ where $\mathsf{host}(URIC) \subseteq host_\mathsf{C} \subseteq sid_\mathsf{C}$. Furthermore, $URIC$ defines a *channel security domain* of C, i.e., CA does not contain another entry $(\mathsf{C}^*, sid^*, host^*)$ with $\mathsf{C}^* \neq \mathsf{C}$ and $host^* \supseteq host$ for any $host \subseteq \mathsf{host}(URIC)$.

g. Similarly, CA contains an entry $(\mathsf{S}, sid_\mathsf{S}, host_\mathsf{S})$ where $\mathsf{host}(URIS) \subseteq host_\mathsf{S} \subseteq sid_\mathsf{S}$ and $URIS$ defines a channel security domain of S.

We call S the identity supplier of C, in formulas $\mathsf{S} = Supplier_\mathsf{C}$, under conditions c. and d.

6.2 Authenticity of Single Sign-on

The property we show is that at the successful end of a WSFPI protocol run, the identity consumer C has established a secure channel with a specific user U. This is more than entity authentication of user U, because the authentication is bound to the secure channel. Such binding is important to guarantee that a potential result in Step 10 of Figure 1 really goes to the identified user and that potential future requests in this channel come from the same user. We model the channel binding by the existence of a secure channel instance $\mathsf{SCh_{bc}}$ in the channel machine secchan as specified in [12, 13]. We present the proof in Section 7.

THEOREM 1 (AUTHENTICITY OF WSFPI). *Let a user* U*, an identity consumer* C*, its identity supplier* S *and a channel machine* secchan *fulfill the WSFPI setup assumptions (Definition 2).*

Let these machines and the user's browser B *be correct, and let* id_U *be defined as in Definition 2. Then if* C *makes an output* $(\mathsf{accepted}, cid, id_\mathsf{U}, att)$ *at* $\mathsf{sso\_out_C}!$*, there exists a secure channel instance* $\mathsf{SCh_{bc}}$ *in* secchan *with* $\mathsf{SCh_{bc}}.cid = cid \wedge \mathsf{SCh_{bc}}.state = \mathsf{established} \wedge \mathsf{SCh_{bc}}.Partner = \{\mathsf{B}, \mathsf{C}\}$*, unless*

[3] This assumption can be restricted to messages of certain valid WSFPI formats. It is needed to exclude undesired information flow from incoming unrecognized messages, because critical messages may be sent to addresses under $URIC$.

an adversary can guess $login_{\mathsf{U},\mathsf{S}}$ based on a priori knowledge of its distribution, its length, and the results of previous guessing attempts, which each exclude one potential value.

7. SECURITY PROOF

To prove Theorem 1 we use Lemmas 1 and 2, which analyze the information flow in a browser during form posting, Lemmas 3, which state properties of the channel abstraction, and Lemma 4, the user authentication lemma from [12].

PROOF. The proof starts with a successful outcome at the identity consumer C. We show that C must have received a security token $wresult$ that was signed by the identity supplier S. We proceed to show that S only produced this security token in interaction with the correct user U, and only sent it to U's browser B and one identity consumer over secure channels, without producing further information flow from the token to other parties and persistent variables. Finally, we show that this identity consumer must have been our C, in other words that the successful outcome at C is not based on a replay of the security token from a protocol run with another, potentially malicious identity consumer.

1. First we prove that if the identity consumer C outputs $(\mathsf{accepted}, cid, id_{\mathsf{U}}, att)$ at some time t^{sso}, then at some time in the past t^{sign} the identity supplier S has created and signed a security token $wresult$ that vouches for identity id_{U} in a WSFPI protocol run.

If C outputs $(\mathsf{accepted}, cid, id_{\mathsf{U}}, att)$ at time t^{sso} in the WSFPI protocol run, C must have been in State $\mathsf{Signature\_valid}$.

Therefore, identity consumer C must have received a wellformed message $(\mathsf{receive}, cid, \mathsf{POST}(path, req, \epsilon, \cdot))$ at port $\mathsf{channel\_out_C}?$ in State $\mathsf{Secure\_channel\_accepted}$, and the parsing $(wa, wresult, wctx) \leftarrow \mathsf{parse\_res}(req)$ was successful.

Furthermore, C set $(name, m) \leftarrow \mathsf{test}(wresult)$ before State $\mathsf{Wellformed\_request}$ and then validated that $name = nameS$. This implies that $name \neq \bot$ and thus that the signature test was successful.

By the setup assumptions of Definition 2, $nameS = name_{\mathsf{S}}$ is derived from the certificate $cert_{\mathsf{S}}$.

Because of the signature-related assumptions of Definition 1, S must have signed message m at some time in the past t^{sign}. By the absence of cross-protocol signature key use from the same definition, this must have happened in a WSFPI protocol run, because m is a message that passes the verification in State $\mathsf{Signature\_valid}$ of C (i.e., it is a SAML token with certain correct parameters). The only sign operation of S is in the transition to State $\mathsf{UAuth\_success}$.[4]

By the definition of identity consumer C in Section 5.2, its assignment $(issuer, audience, idu, att) \leftarrow \mathsf{parse\_token}(m)$ in State $\mathsf{Signature\_valid}$ indeed gave the value idu that it later output, i.e., $idu = id_{\mathsf{U}}$.

2. We prove next that S must have created the signed token $wresult$ upon successful execution of a user authentication sub-protocol. The channel identified as cid' by this user authentication sub-protocol is a secure channel with user U and its browser B.

The transition to State $\mathsf{UAuth\_success}$ at time t^{sign} is only executed if S obtained a message $(\mathsf{done}, cid', idu')$ at $\mathsf{uauth\_out}?_{\mathsf{S}}$ for a non-empty registered user identity idu' and a channel identifier cid'. We name the time of this output t^{uauth}.

In State $\mathsf{UAuth\_success}$, identity supplier S puts the user identity idu' obtained from the output of the user authentication into the message m within token $wresult$ with the function $\mathsf{gen\_token}$. By the correspondence of the functions $\mathsf{gen\_token}$ and $\mathsf{parse\_token}$ and by the last part of Step 1. of this proof we obtain $idu' = idu = id_{\mathsf{U}}$.

By Definition 2, the user identity id_{U} is unique in the identity supplier's metadata table $MetaU_{\mathsf{S}}$, and the only other variable with information about the corresponding login information $login_{\mathsf{U},\mathsf{S}}$ is $login_{\mathsf{U},\mathsf{S}}$ at U.

We now use the user authentication lemma, Lemma 4: That identity supplier S receives a message $(\mathsf{done}, cid', id_{\mathsf{U}})$ at t^{uauth} implies that the sub-protocol uauth was executed successfully, where cid' is the identifier of a secure channel and the partner machine at this channel is the browser B of the user U with identity id_{U}, unless an adversary can guess $login_{\mathsf{U},\mathsf{S}}$ based on a priori knowledge of its distribution, its length, and the results of previous guessing attempts, which each exclude one potential value.

Let $\mathsf{SCh_{bs}}$ be the secure channel instance in $\mathsf{secchan}$ with channel identifier cid'. Then the result in formulas is $\mathsf{SCh_{bs}}.Partner = \{\mathsf{B}, \mathsf{S}\}$ and $\mathsf{SCh_{bs}}.ch\_type = \mathsf{client\_auth\_secure}$.

3. We now prove that the identity supplier S, after producing the token $wresult$, only sends it to the given browser B in a POST-Form message pf that it inputs to $\mathsf{secchan}$, and does not produce any other information flow from $wresult$ to other parties or its own persistent variables.

When S generates the security token $wresult$ in State $\mathsf{Requested\_UAuth}$, it immediately packages it as a result $res \leftarrow \mathsf{gen\_res}(\mathsf{wsignin1.0}, wresult, wctx)$. The only reuse of $wresult$ or res before the transaction ends and thus these volatile variables disappear, is in the transition to the final state $\mathsf{Success : POSTForm\_sent}$. Here S generates a message $pf = \mathsf{POSTForm}(\mathsf{host}(padr), \mathsf{path}(padr), res, \mathsf{TRUE}, \mathsf{TRUE})$ and sends it through the channel $\mathsf{SCh_{bs}}$ designated by the identifier cid' obtained from uauth.

Within pf, the address $padr$ for later posting of the form is set to $wtrealm$ or $wreply$. In the latter case, $wreply \sqsubseteq wtrealm$ holds by a test in State $\mathsf{Request\_received}$. The test $MetaC_{\mathsf{S}}[URIC = wtrealm] \neq \epsilon$ in the same state implies that $wtrealm$ is the security domain of an identity consumer known to S. (However, we have not yet shown that it is C, nor have we assumed that other identity consumers are correct.) Furthermore, the preceding test $\mathsf{ctype}(padr) = \mathsf{secure}$ ensures that a secure channel will be used later to post the form.

4. We now consider the information flow of $wresult$ from pf in the channel machine $\mathsf{secchan}$.

By Lemma 3, the only information flow from pf that can happen in the secure channel instance $\mathsf{SCh_{bs}}$ is that pf is delivered unmodified to B, and that the adversary A learns $\mathsf{length}(pf)$.

With this information, A can at most derive $\mathsf{length}(wresult)$ as information about our protected variable by a signature-related assumption from Definition 1. By another assumption from that definition, this does not give A any information about the signature proper in $wresult$ because the length of that is known in advance.

5. We now show that the browser B only posts the form within pf over a secure channel, and that no other information flow from $wresult$ to other parties or persistent variables takes place in B.

State inspection shows that B only accepts a message pf from a channel in State $\mathsf{Await\_response}$, and then proceeds to State $\mathsf{Active\_FormPOST}$. (Channel outputs of the form $(\mathsf{receive}, \cdot, \cdot)$ are only handled in State $\mathsf{Await\_Response}$. Only two input types match the $\mathsf{POSTForm}$ parameter, and the other has a condition that

[4] As we already use idealized secure channels here, it cannot happen in the secure channel use of S either. In practice, this means that the secure channels must either use separate keys or a similar cross-protocol key use assumption must be made for them.

the message is not parsable, while pf is.)

In the remainder of this transaction of B, the only information flow from the form res in the $\mathsf{POSTForm}$ message pf is to the persistent variable $prev\_run.form$. This follows from Lemma 1, where our res is the parameter $rquery$.

At the end of this transaction, the browser initiates a new HTTP transaction where it posts the obtained form. It does this by outputting a message $(\mathsf{submit\_form}, wid, \epsilon, ra)$ to $\mathsf{self_B}!$ in State $\mathsf{Transaction\_finished}$ and clocking the corresponding clock port. Here $ra = (\mathsf{host}(padr), \mathsf{path}(padr))$.

Upon receiving such a message $(\mathsf{submit\_form}, wid, \epsilon, ra)$ at $\mathsf{self_B}?$ in State Start, the only information flow in the browser B from the initial value of $prev\_run.form$ to other parties and persistent variables (including the final value of $prev\_run.form$) is to $\mathsf{secchan}$ by a message $(\mathsf{send}, ch.cid, post)$ at $\mathsf{channel\_in_B}!$ with $post = \mathsf{POST}(\mathsf{path}(ra), prev\_run.form, \epsilon, prev\_run.adr)$, where ch is a channel variable with $ch.host = \mathsf{host}(ra)$ and $ch.type = \mathsf{secure}$. Here we applied Lemma 2; this is possible since $\mathsf{ctype}(ra) = \mathsf{ctype}(padr) = \mathsf{secure}$ by Step 3.

6. We show next that C is the channel partner of B in this POST transaction. We first show that the redirect address $padr$ is indeed an address of C, and then that consequently the secure channel instance used has the correct channel partners. Informally this means that the key and address bindings using SSL certificates and our setup assumptions about security domains are sufficient.

When producing the token $wresult$, the identity supplier S used $wtrealm$ as the $audience$ parameter. (This happened in State $\mathsf{Requested\_UAuth}$ as the second parameter of the function $\mathsf{gen\_token}$.) As C accepts the token $wresult$ at time t^{sso}, its tests in State $\mathsf{Signature\_valid}$ imply that $audience = URIC$. With Step 3. this implies $padr \subseteq wtrealm = URIC$.

The input $(\mathsf{send}, ch.cid, post)$ is only handled in $\mathsf{secchan}$ if there is a channel instance $\mathsf{SCh_{bc}}$ with $\mathsf{SCh_{bc}}.cid = ch.cid$ and in State $\mathsf{Established}$. By Lemma 3, this implies $\mathsf{SCh_{bc}}.host = \mathsf{host}(ra) = \mathsf{host}(padr)$ and $\mathsf{SCh_{bc}}.type = \mathsf{client\_auth\_secure}$.

A secure channel instance only enters State $\mathsf{Established}$ if there exists a binding $(\mathsf{R}, rid, host') \in CA$ with $\mathsf{SCh_{bc}}.host \subseteq host'$, i.e., with $\mathsf{host}(padr) \subseteq host'$. By the setup assumptions (Definition 2 f.), there exists a $binding_{\mathsf{C}} = (\mathsf{C}, sid_{\mathsf{C}}, host_{\mathsf{C}}) \in CA$ with $\mathsf{host}(URIC) \subseteq host_{\mathsf{C}}$ and thus indeed $\mathsf{host}(padr) \subseteq host_{\mathsf{C}}$. Hence this binding matches.

Still by Definition 2 f., CA contains no other binding $(\mathsf{C}^*, sid^*, host^*)$ with $\mathsf{C}^* \neq \mathsf{C}$ and $host^* \supseteq \mathsf{host}(padr)$ because $\mathsf{host}(padr)$ lies in the security domain defined by $URIC$.

Hence the secure channel instance $\mathsf{SCh_{bc}}$ uses the binding $binding_{\mathsf{C}}$, and the test in State $\mathsf{accept\_request}$ guarantees that $\mathsf{SCh_{bc}}.Partner = \{\mathsf{B}, \mathsf{C}\}$.

7. We show next that no information about $wresult$ leaks to other parties or persistent variables from the secure channel.

According to Lemma 3, the only information flow from message $post$ (recall the last part of Step 5.) is an output $(\mathsf{receive}, ch.cid, post)$ to C, and that the adversary A learns $\mathsf{length}(post)$. With the signature-related assumptions (Definition 1) this implies that A learns at most the length of the parameter $prev\_run.form = res$ within $post$ and of $wresult$ in res. As in Step 4., this yields no information about the signature proper, which is the only part whose information flow interests us.

8. As the final information flow step we show that the identity consumer C does not produce information flow from the signature in $wresult$ to other parties.

C only accepts a message $(\mathsf{receive}, ch.cid, post)$ in the transition to State $\mathsf{Request\_received}$. The part $prev\_run.form = res$ is put into the volatile variable req, which is immediately parsed into $(wa, wresult, wctx)$ and never accessed again.

In the transition to State $\mathsf{Wellformed\_request}$, the identity consumer tests the signature of $wresult$ and projects the result to $(name, m)$. Afterwards $wresult$ itself is no longer accessed. Hence there is no further information flow from the signature proper, and the fact that the signature is valid is known beforehand.

9. We have now seen that the only information flow from $wresult$ after its creation by S at time t^{sign} is via correct parties to the identity consumer, who keeps the contained signature secret. Hence the adversary has only a negligible probability of guessing this signature. Here we use the result from [1] that memory-less and counter-based signature schemes are also reactively secure, and our signature-related assumptions (Definition 1).

Hence C's acceptance of the security token $wresult$ at time t^{sso} must happen as a direct consequence of receiving $(\mathsf{receive}, ch.cid, post)$ as described in Part 8. In particular, this implies $cid = ch.cid$ (where cid is the channel identifier in C's output). Hence $\mathsf{SCh_{bc}}$ fulfills all the conditions of the theorem.

This finishes our proof. □

The full proof for the lemmas that we used is part of the long version of this paper [13]. Firstly, we present two lemmas about the information flow in a browser while receiving a $\mathsf{POSTForm}$ response from a server, and while POSTing the corresponding form in the next HTTP transaction. Secondly, we prove specific properties of our secure channel abstraction. Thirdly, we repeat the user authentication lemma from [12], which we use for the underlying user authentication at the identity supplier S. The authentication server occurring in it is called S in the original, but in our context it corresponds to a submodule used by the machine S.

LEMMA 1. *(Browser information flow from a* $\mathsf{POSTForm}$ *response) If a correct browser* B *accepts an input message* $(\mathsf{receive}, cid, \mathsf{POSTForm}(radr, rpath, rquery, close, \mathsf{TRUE}))$ *at* $\mathsf{channel\_out_B}?$*, then at most the following information flow takes place from this message to other parties or persistent variables of* B *before the transaction end, and thus the deletion of the volatile variables: (a) Direct to* B*:* $rquery$ *to the persistent variable* $prev\_run.form$*. (b) Direct to* B*:* $radr$ *and* $rpath$ *to* $\mathsf{self_B}!$*. (c) Indirect:* $close$ *to the adversary.*

LEMMA 2. *(Browser information flow while POSTing a form) Let* B *be a correct browser that accepts an input message* $(\mathsf{submit\_form}, wid, form\_in, ra)$ *at* $\mathsf{self_B}?$ *where* $\mathsf{ctype}(ra) = \mathsf{secure}$ *and* $prev\_run.form \neq \epsilon$ *(i.e., a form was indeed obtained from the partner). Then the only information flow from* $prev\_run.form$ *to other parties or persistent variables of* B *is an output* $(\mathsf{send}, ch.cid, post)$ *at* $\mathsf{channel\_in_B}!$ *with* $post = \mathsf{POST}(\mathsf{path}(ra), prev\_run.form, \epsilon, prev\_run.adr)$*, where* ch *is a channel variable with* $ch.host = \mathsf{host}(ra)$ *and* $ch.type = \mathsf{secure}$.

LEMMA 3. *(Information flow in secure channels) Let* SCh *be a channel instance of the channel machine* $\mathsf{secchan}$ *with* $\mathsf{SCh}.type = \mathsf{client\_auth\_secure}$ *and* $\mathsf{SCh}.Partners = \{\mathsf{P}, \mathsf{Q}\}$*, and let* $cid := \mathsf{SCh}.cid$*. Then if* $\mathsf{secchan}$ *obtains an input* (send, cid, m) *from* P*, the only possible information flow from* m *to other parties or* SCh*'s own persistent variables is an output* $(\mathsf{receive}, cid, m)$ *to* Q *and an output containing* $\mathsf{length}(m)$ *to the adversary* A.

Consistency of channel parameters: *If a correct browser contains a channel variable* ch *and* $\mathsf{secchan}$ *contains a channel instance* SCh *with* $\mathsf{SCh}.cid = ch.cid$*, then* $\mathsf{SCh}.ch\_type = ch.type$ *and* $\mathsf{SCh}.host = ch.host$*.*

LEMMA 4. *(User Authentication) Let a correct user machine* U *and authentication server* S' *be given that have performed setup according to Section 6.2 of [12] at some time with the user identity* id_{U}*, and let the user's browser* B *be correct. Then* S' *only outputs* $(\mathsf{done}, cid, id_{\mathsf{U}})$ *at* $\mathsf{uauth\_out}_{\mathsf{S}'}!$ *if* cid *is the identifier of a secure channel, and the partner machine at this channel is the browser* B *of the given user* U*, unless an adversary can guess* $login_{\mathsf{U},\mathsf{S}}$ *based on a priori knowledge of its distribution, its length, and the results of previous guessing attempts, which each exclude one potential value.*

8. CONCLUSION

We have devised a rigorous security proof for the WS-Federation Passive Requestor Interop protocol (WSFPI) based on a general browser model. In contrast to an earlier approach at such a security proof, we used a detailed and reusable model for user and browser instead of top-down assumptions about these parties. This proof based on a browser model is the first one of that kind for browser-based identity federation protocols. This work also constitutes the first proof of a larger real world system in the browser model. Although we shaped protocol and browser models closely to real WS-Federation Passive Requestor deployments, real environments will not always fulfill our assumption that the signature keys will exclusively be used for signing WSFPI tokens. Developers should be aware of the risks of protocol interference attacks mountable in such a scenario. As future work, we intend to analyze other browser-based FIM protocols with a similar technique. Protocols that do not rely on the POST message and use a random number as reference to security tokens, a so called artifact, are of special interest there.

9. REFERENCES

[1] M. Backes, B. Pfitzmann, and M. Waidner. Reactively secure signature schemes. *International Journal of Information Security*, 2005. To appear. Preliminary version ISC 2003, LNCS, pp. 84–95.

[2] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology: CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1994.

[3] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 198–209. ACM Press, 2004.

[4] R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. In *Advances in Cryptology: EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2002.

[5] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[6] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, June 1999.

[7] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., Aug. 2001. USENIX.

[8] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[9] A. D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *Proc. 2002 ACM Workshop on XML Security*, pages 18–29, Fairfax VA, USA, Nov. 2002.

[10] T. Groß. Security analysis of the SAML Single Sign-on Browser/Artifact profile. In *Proc. 19th Annual Computer Security Applications Conference*. IEEE, Dec. 2003.

[11] T. Groß and B. Pfitzmann. Proving a WS-Federation Passive Requestor profile. In *2004 ACM Workshop on Secure Web Services (SWS)*, Washington, DC, USA, Oct. 2004. ACM Press. To appear.

[12] T. Groß, B. Pfitzmann, and A.-R. Sadeghi. Browser model for security analysis of browser-based protocols. In *ESORICS: 10th European Symposium on Research in Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 489–508. Springer-Verlag, Berlin Germany, 2005.

[13] T. Groß, B. Pfitzmann, and A.-R. Sadeghi. Proving a WS-Federation Passive Requestor profile with a browser model. Technical Report IBM Research Report RZ 3623, IBM Research Division, July 2005.

[14] M. Hur, R. D. Johnson, A. Medvinsky, Y. Rouskov, J. Spellman, S. Weeden, and A. Nadalin. Passive Requestor Federation Interop Scenario, Version 0.4, Feb. 2004. `ftp://www6.software.ibm.com/software/developer/library/ws-fpscenario2.doc`.

[15] C. Kaler and A. Nadalin. Web Services Federation Language (WS-Federation), Version 1.0, July 2003. BEA, IBM, Microsoft, RSA Security and VeriSign, `http://www-106.ibm.com/developerworks/webservices/library/ws-fed/`.

[16] C. Kaler and A. Nadalin. WS-Federation: Passive Requestor Profile, Version 1.0, July 2003. BEA and IBM and Microsoft and RSA Security and VeriSign, `http://www-106.ibm.com/developerworks/library/ws-fedpass/`.

[17] D. P. Kormann and A. D. Rubin. Risks of the Passport single signon protocol. *Computer Networks*, 33(1–6):51–58, June 2000.

[18] Liberty Alliance Project. Liberty Phase 2 final specifications, Nov. 2003. `http://www.projectliberty.org/`.

[19] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–135, 1995.

[20] N. Lynch. I/O automaton models and proofs for shared-key communication systems. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW)*, pages 14–29, 1999.

[21] Microsoft Corporation. .NET Passport documentation, in particular Technical Overview, and SDK 2.1 Documentation, Sept. 2001.

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

[23] OASIS Standard. Security assertion markup language (SAML) V1.1, Nov. 2002. `http://www.oasis-open.org/committees/security/`.

[24] Object Management Group. Unified modeling language (UML), Mar. 2003. `http://www.omg.org/technology/documents/formal/uml.htm`.

[25] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 184–200, Oakland, CA, May 2001. IEEE Computer Society Press.

[26] B. Pfitzmann and M. Waidner. Privacy in browser-based attribute exchange. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 52–62, Washington, USA, Nov. 2002.

[27] B. Pfitzmann and M. Waidner. Analysis of Liberty single-signon with enabled clients. *IEEE Internet Computing*, 7(6):38–44, 2003.

[28] B. Pfitzmann and M. Waidner. Federated identity-management protocols — where user authentication protocols may go. In *Security Protocols—11th International Workshop*, Lecture Notes in Computer Science, Cambridge, UK, Apr. 2003. Springer-Verlag, Berlin Germany.

[29] V. Shoup. On formal models for secure key exchange. Research Report RZ 3120 (#93166), IBM Research, Apr. 1999.