Browser Model for Security Analysis of Browser-Based Protocols

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

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

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

Abstract

Currently, many industrial initiatives focus on web-based applications. In this context an important requirement is that the user should only rely on a standard web browser. Hence the underlying security services also rely solely on a browser for interaction with the user. Browser-based identity federation is a prominent example of such a protocol. Unfortunately, very little is still known about the security of browser-based protocols, and they seem at least as error-prone as standard security protocols. In particular, standard web browsers have limited cryptographic capabilities and thus new protocols are used. Furthermore, these protocols require certain care by the user in person, which must be modeled. In addition, browsers, unlike normal protocol principals, cannot be assumed to do nothing but execute the given security protocol.

In this paper, we lay the theoretical basis for the rigorous analysis and security proofs of browser-based security protocols. We formally model web browsers, secure browser channels, and the security-relevant browsing behavior of a user as automata. As a first rigorous security proof of a browser-based protocol we prove the security of password-based user authentication in our model. This is not only the most common stand-alone type of browser authentication, but also a fundamental building block for more complex protocols like identity federation.

1 Introduction

Web-based services have received increasing attention in the last years. The idea is simple: users should be able to send their requests for desired services using a browser, which offers a set of basic functionalities, and receive and view the results at the browser. This allows easy deployment of applications at low cost and without specific user education. Services can be offered by one service provider or several affiliated enterprises. The requirement on such services not to need any special client software is also called *zero-footprint*. The underlying security services must also be zero-footprint, i.e., only a browser is used for user authentication, and, if desired, for retaining a secure channel with the user, requesting additional security relevant attributes about the users, and potential confirmation by third parties of the authentication of the authentication or attribute information.

The typical approach in other security protocols is to perform a key exchange, based on local master keys, master keys shared with a third party, or public-key certificates, and to subsequently use the exchanged key to secure the communication. A large body of literature on such protocols exist. A seminal paper was [29], although a vulnerability in one of the original protocols was later found in [22]. Tool-supported proofs were initiated in [26, 18, 25], based on abstractions of cryptographic primitives introduced in [8]. Recent tool-supported proofs concentrated on using existing general-purpose model checkers, first in [23, 27, 6], and theorem provers, first in [9, 30]. Cryptographic proofs of key-exchange and authentication protocols were initiated in [1]. Cryptography also added interesting additional properties to pure authentication, e.g., see [20]. 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 [40, 34, 3]. Analyses specifically for SSL and TLS, and thus close to an underlying mechanism used in browsers, were made in [42, 28, 31, 21].

However, standard browsers simply do not execute most of these protocols. The only exception that would include user authentication would be to use SSL or TLS channels with client certificates for 2-party authentication. However, this is not considered truly zero-footprint because the users would have to obtain the certificates, and because it would not allow a user to easily use different browsers at different times. Thus it is very rarely used, and not used at all as a basis in larger browser-based security protocols. Hence browser-based protocols are different from all protocols for which prior security proofs exist.

A prominent example of browser-based security protocols is identity federation, which aims at linking a user's (otherwise) distinct identities at several locations. The advantage of such systems is that the involved organizations can reduce user management costs, such as the cost of password helpdesks and user registration and deletion. In particular in this area, concrete and complex browser-based security protocols were proposed, e.g., Microsoft's Passport [5], the Security Assertion Markup Language (SAML) standardized by OASIS [41], the Shibboleth project for university identity federation [4], the Liberty Alliance project [38], and WS-Federation [16, 17]. Several papers discussed vulnerabilities of such protocols, in particular for Passport [19], the Liberty enabled-client protocol [37], and a SAML profile [13]. Others discussed privacy design principles and details [36, 32, 33]. Basic browser-based authentication without federated identity management is discussed in [11]. As far as the vulnerabilities found were removable security problems (in contrast to fundamental limitations of the browser-based protocol class or matters of taste like privacy), they were removed in the next version of the protocols. However, past experience in protocol design has shown that incorporating countermeasures against known attacks does not guarantee to eliminate all vulnerabilities. Hence it is desirable to devise security proofs.

It is not trivial to apply previous security proof techniques, both cryptographic techniques and formal-methods techniques, to browser-based protocols. The primary reason is that a browser represents a new party with its own, predefined behavior that has impact on 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, e.g., in an Internet kiosk. For instance, one of the SAML problems found in [13] is based on the HTTP Referer tag, i.e., a browser feature that is not mentioned at all on the level of the SAML protocol. Another usual issue is that browser-based protocols use a multitude of names for a principal, while other protocols typically assume a one-to-one mapping; for instance, there are URL addresses, identities used in SSL certificates, and identities used in higher protocols, and it is easy to forget some name comparisons in protocols and thus to enable man-in-the-middle attacks. All this means that a detailed and rigorous browser model is a prerequisite for convincing security proofs of browser-based protocols, and no such model exists so far. For the resulting 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. Hence it is not enough to make a minimal model covering the few messages and parameters explicitly used by security protocols, but one has to get as close as possible to real browsers.

Another aspect is that due to the limited capabilities of browsers, the user at the browser is an active participant and certain assumptions must be made about the user as well, e.g., that the user verifies that a secure channel to a trusted server is used before entering an important password.

In this paper, we lay the theoretical basis for research in this area by modeling the major building blocks for browser-based protocols. We present a rigorous and abstract model for a standard web browser as a principal for browser-based protocols. While our model is still extensible – in particular we do not model cookies and scripting but assume a browser with these features turned off – we believe that we have captured the major explicit and implicit browser features that play a role in typical browser-based protocols. In addition, we model the security-relevant browsing behavior of a user, i.e., a machine that implements the explicit constraints on a user that are needed for protocol proofs, but still allows arbitrary behavior apart from that. Furthermore, we model browser channels in order to capture, in particular, the naming issues across multiple protocol layers.

As a first security lemma for a browser-based protocol in our model, we focus on the security of the initial authentication of the user behind a browser by a password. Initial user authentication is an integral part of all browser-based protocols, and passwords are the standard technique used in the zero-footprint scenario.

A first step in the direction of proofs of browser-based protocols was taken in [14]. There, however, we only modeled exactly those parts of the user and browser behavior that we concretely needed for the protocol, and made assumptions that other things would not happen, where the assumptions were made top-down for the needs of the protocol rather than bottom-up from a browser and user model. In this paper, we lay the bottom-up groundwork for such assumptions.

Another related area is the analysis of web services security protocols [12, 2], because in standardization web-based protocols and web services protocols are closely related, e.g., in WS-Federation or in the use of SAML tokens for web services protocols. The techniques developed there are very useful for security proofs for real standards. However, so far, they do not consider browser-based protocols, and hence do not affect the novelty of our browser and user models and of proofs based on them.

2 Notation

General Notation. We use a straight font for constants, including constant Sets and Types, functions, and predicates, where Types are predefined constant sets. We use italics for *variables* and variable *Sets*. Let Σ be an alphabet without the symbols { "ϵ" , "!", "?", "$^\triangleleft$" , "[", "]", "//" }; then Σ^* is the set of strings over Σ where ϵ denotes the empty string and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$. For a set S, $\mathcal{P}(S)$ denotes the powerset of S and S^* the set of finite sequences over S. We define $S.\mathsf{add}\ (x)$ as $S := S \cup \{x\}$ and $S.\mathsf{remove}\ (x)$ as $S := S \setminus \{x\}$. Assignment by possibly probabilistic functions is written as $\leftarrow$. Assignment of a value to a tuple of variables means making correspondingly many projections; if one of these fails the entire assignment fails. 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 type $\mathsf{ChType} := \{\mathsf{secure}, \mathsf{insecure}\}$ contains the channel types available.

Automata. 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 very usual basis for specifying participants in distributed protocols; the first specific use for security is in [24]. Specifically we use the automata model proposed in [35], which has a well-defined realization by probabilistic interactive Turing machines and is therefore linked to more detailed cryptographic considerations where those become necessary in multi-layer proofs. In the following we give a brief overview of this machine model (see also Figure 2). Machines may have multiple fixed connections to other machines organized by means of *ports*. We define a *simple port* for message transmission as $p = (n, d) \in \Sigma^+ \times \{!, ?\}$ where n indicates the port *name* and d the port *direction*. Ports are uni-directional, where $d = !$ denotes output and $d = ?$ input. The machine model connects simple ports $\mathsf{n}?$ and $\mathsf{n}!$ of same name n and opposite direction d; these are called *complement* ports. We call ports without such a complement *free* ports. We define a *clock port* $p = (n,^\triangleleft, d) \in \Sigma^+ \times \{^\triangleleft\} \times \{!, ?\}$ as a port that schedules the connection between simple ports $\mathsf{n}?$ and $\mathsf{n}!$ with same name n, or is free itself if this connection does not exist.

A *machine* M is defined as a tuple $\mathsf{M} = (name_\mathsf{M}, Ports_\mathsf{M}, Vars_\mathsf{M}, States_\mathsf{M}, \delta_\mathsf{M}, Ini_\mathsf{M}, Fin_\mathsf{M})$ of a name $name_\mathsf{M} \in \Sigma^+$, a finite sequence $Ports_\mathsf{M}$ of ports, a finite sequence $Vars_\mathsf{M}$ of local variables a

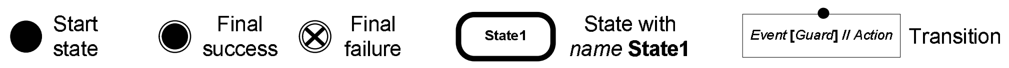

Figure 1: Key to the state diagrams

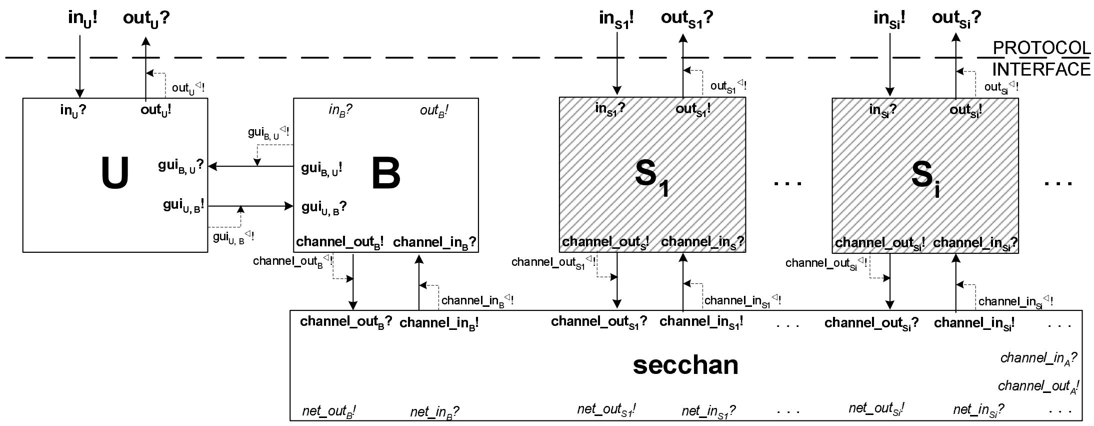

Figure 2: System architecture for browser-based protocols with a browser B, a user U, servers S_i, secchan and their interfaces.

set $States_{\mathsf{M}} \subseteq \Sigma^*$ of major states, a probabilistic state-transition function δ_{M}, and sets $Ini_{\mathsf{M}}, Fin_{\mathsf{M}} \subseteq States_{\mathsf{M}}$ of initial and final states. The inputs are tuples $I = (I_i)_{i=1,\dots,|\mathsf{in}(Ports_{\mathsf{M}})|}$, where $I_i \in \Sigma^*$ is the input for the i-th in-port, $\mathsf{in}(Ports_{\mathsf{M}})$ is the input ports of the machine and $|\mathsf{in}(Ports_{\mathsf{M}})|$ denotes the number of the input ports. Analogously, the outputs are tuples $O = (O_i)_{i=1,\dots,|\mathsf{out}(Ports_{\mathsf{M}})|}$. The empty word, ϵ, denotes "no in- or output", respectively. The value assignments of local variables are tuples $V = (V_i)_{i=1,\dots,|Vars_{\mathsf{M}}|}$, where $V_i \in \Sigma^*$ is the value for the i-th variable in the sequence $Vars_{\mathsf{M}}$.

We define the state transition function δ_{M} of a machine M using a notation analogous to UML state diagrams [15] (see Figure 1). We define a *transition* of such a state diagram as an arrow from state s to s' with label $Event[Guard]//Action$, where the components are defined as follows: 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 V which is the machine's current variable allocation. $Action$ specifies computations and outputs of the transition.

System Overview for Browser-based Protocols. Figure 2 gives an overview of the automata in our model. We call the generic browser machine B, the user machine that implements the minimum assumptions on secure user behavior U, and the machine that models the behavior of secure channels as implemented within HTTPS (see [39]) by secchan. To analyze and prove browser-based security protocols one complements these general-purpose machines with one or more server machines, here denoted by S_i, that jointly execute the browser-based protocol. Furthermore, one configures the user machine U with a suitable initial trust relationship and knowledge about trusted parties.

3 Ideal Web Browser B

In this section, we introduce the functionality of real web browsers, and we describe the feature set that we model and its benefit for the rigorous security analysis of browser-based security protocols.

A web browser acts as the client in transactions of the Hypertext Transfer Protocol (HTTP) [10] 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. The browser model represents a window by a window identifier in the communication between U and B. We therefore allow a user machine to distinguish multiple windows; however, the window management of the browser machine is omitted for brevity. A real browser accepts inputs from the user specifying addresses to retrieve and render 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, a certified server identity only if secure channels are used.

HTTP is a client-server protocol positioned in the application layer of the TCP/IP protocol stack with variable underlying transport protocols. In order to initiate an HTTP transaction, a web browser establishes a connection to a server specified by the address to access and may leverage multiple types of transport protocols underlying the HTTP transaction, e.g., TCP/IP, SSL 3.0 or TLS1.0 [7]. 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. We call such an interaction an HTTP transaction. In principle, browsers do not need to hold state beyond such a single transaction, however, a real web browser builds, e.g., local cache and browsing history and lets a transaction influence the subsequent one. Our browser model reflects this behavior of real browsers.

HTTP transactions may implement various functions. Clients such as real browsers may use a transaction to retrieve data by a GET request, and send data by a POST request. Servers may not only deliver content but also direct the browser to a behavior change by issuing executable scripts and error messages. Most prominent examples are HTTP responses with scripted form POST and redirect messages, which direct the browser to another address of the server's choice. The server may also issue an HTTP response that requests a user's authentication by means of a username-password pair We model browser messages as abstract formats and abstract from parsing bitstrings. We focus on the subset of message and parameter types of HTTP that is actively used in browser-based protocols or may have impact on their security. Thus, for modeling a standard browser-based protocol like SAML [41], Liberty [38] or WS-Federation [16] the model provides the right set of functionality without overwhelming with too many options.

An important aspect of real web browsers is that they do more than browser-based protocols intend. Most prominent is the problem of information flow. On the one hand, information flows through the HTTP requests to the server, e.g., by means of the HTTP Referer tag. Also, features such as history, cache or password storage have data flow to the underlying operating system. We explicitly model this property of real browsers in order to allow information flow analysis. We dedicate Section 3.4 to this important part of the browser specification.

Supported by a rudimentary trust model, a web browser can establish secure channels to servers by leveraging the SSL3.0 or TLS protocol. We model the establishment of insecure and secure channels and the corresponding key management by the channel machine secchan. The user's log off from the browser session that removes browsers state from the machine's memory is also security-relevant.

Further, we do not consider cookies as many browser-based protocols do not use them directly.

In Section 3.1 we define the interface consisting of ports $Ports_{\mathsf{B}}$ and possible events. Section 3.3 specifies static elements of B such as the set of local variables $Vars_{\mathsf{B}}$, whereas we explain algorithms and predicates in Section B. Specific abstract browser messages is the subject of Section 3.2 while Section 3.4 considers the imperfections of the browsers. Section 3.5 describes the behavior model of B consisting of the state sets $States_{\mathsf{B}}$, Ini_{B}, and Fin_{B} as well as the transition function δ_{B}.

3.1 Interface of Browser Machine B

We refine the interface of B depicted in Figure 2 by defining the exact messages types transferred over the ports of B. We list them all in Table 5 of Appendix Section B. Here we only explain them as far as it is useful to understand the upcoming state diagram.

The ports $\mathsf{gui}_{\mathsf{U,B}}?$ and $\mathsf{gui}_{\mathsf{B,U}}!$ model the browser's user interface and connect B to a user machine

U. The messages enter_address, trigger_address and submit_form issue a request for an address to B. The enter_address message represents an input in the browser's address field, whereas trigger_address models clicking a link. The message submit_form defines the submission of HTML forms. The messages established and error inform the user of the channel status. The message channel_change notifies the user of a change of the security level of the transport channel. The remaining messages organize a certificate verification dialog with the user. The browser uses request_uauth and authenticate in the password-based user authentication dialog. The security of browser-based protocols builds upon the browser machine reliably presenting secure channels and the server identity to their users. Thus, if an HTTP transaction uses a secure channel, B includes the channel's server identity in each message to U. The user machine U confirms the server identity in each message to B explicitly.[1]

The input and output ports $\mathsf{channel\_in_B}?$ and $\mathsf{channel\_out_B}!$ connect the browser to the underlying channel abstraction secchan. We introduce the ports $\mathsf{self_B}!$ and $\mathsf{self_B}?$ to reduce complexity of the state diagrams. By means of these ports, we allow the browser to delegate trigger_address and submit_form commands to itself on the same command path as the user inputs. The ports $\mathsf{in_B}?$ and $\mathsf{out_B}!$ model information flow to the OS and may be connected to a higher protocol layer or the adversary. We discuss in Section 3.4 how B explicitly leaks information about its state. Loosely speaking, upon an input do_leak on $\mathsf{in_B}?$ the browser outputs its full persistent state to $\mathsf{out_B}!$.

3.2 Specific Abstract Browser Messages

Correct browsers only send messages well-formed according to HTTP and only accept messages parsable according to HTTP. We model this property by refining the messages the browsers communicate at the interface to the channel machine secchan.

The abstract message $\mathsf{GET}(adr : \mathsf{URLHost}, path : \mathsf{URLPath}, query : \Sigma^*, login : \Sigma^*, info\_leak : (\Sigma^* \times \Sigma^*)^*)$ models an HTTP GET request. The parameters adr and $path$ contain the address to be retrieved. The $query$ is encoded in the query string of the URL. 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 discuss information a browser may leak to other parties through this parameter in Section 3.4. We define $\mathsf{POST}(adr : \mathsf{URLHost}, path : \mathsf{URLPath}, query : \Sigma^*, login : \Sigma^*, info\_leak : (\Sigma^* \times \Sigma^*)^*)$ for HTTP POST requests analogously.

We also define abstract messages to model HTTP responses a browser receives including authentication queries, redirects and forms for scripted POSTs. The abstract messages $\mathsf{Page}(m : \Sigma^*, close : \mathsf{Bool}, nocache : \mathsf{Bool})$ and $\mathsf{Error}(m : \Sigma^*, close : \mathsf{Bool})$ model HTTP 200 OK responses and HTTP 40x error responses, both containing a page m as payload and a flag $close$ that directs the browser to close the underlying channel or keep it alive for further HTTP transactions. This parameter models the Connection header of HTTP1.1 [10] and its token close.The parameter $nocache$ of page models the cache-response directive nostore of HTTP1.1, which forces a browser not to store any part of either this response or the request that elicited it. The abstract message $\mathsf{Redirect}(adr : \mathsf{URLHost}, path : \mathsf{URLPath}, query : \Sigma^*, close : \mathsf{Bool})$ models a redirect (HTTP 302 or 303) to $adr/path?querystring$, where $querystring$ is an encoding of the abstract $query$. Similarly, $\mathsf{POSTForm}(adr : \mathsf{URLHost}, path : \mathsf{URLPath}, query : \Sigma^*, close : \mathsf{Bool}, nocache : \mathsf{Bool})$ models a form containing a script that will POST a message whose body encodes the abstract $query$ to the address $adr/path$[2]. The parameters $close$ and $nocache$ model the connection and cache-response directives of HTTP1.1. In consequence of both messages the browser establishes a channel to the address

[1] We only model that the user sees the server identity, not a channel identifier, because he or she will not notice if a channel is interrupted. Usually, however, a user can distinguish different channels with one partner by different windows.

[2] Abstract message POSTForm ("https://www.somedomain.org", "/login/user", "login=nobody,nobody_pwd", FALSE, TRUE) describes a POST to address "https://www.somedomain.org/login/user" over a secure channel which transfers the query "login=nobody,nobody_pwd". The command directs the browser to keep the channel alive and not to store the message in the browser cache.

Name	Domain	Description	Init.
$prev\_run$	$\mathsf{ChType} \times \mathsf{URLHostPath} \times \Sigma^*$	Data of preceding HTTP transaction	undef
wid	Σ^*	Identifier of the browser window	undef
$Channels$	$\mathsf{Channel}^*$	Set of all channels the browser holds	$\emptyset$
$UAuth$	$\Sigma^* \times \Sigma^* \times \mathsf{URLHost}$	Set of login data stored by B	$\emptyset$
$History$	$\mathsf{URLHostPath}^*$	History of addresses successfully retrieved	$\emptyset$
$Cache$	$(\mathsf{URLHostPath} \times \Sigma^*)^*$	Cache of all pages retrieved	$\emptyset$

Table 1: Persistent local variables of the browser machine B.

adr and then sends $path$ and $query$ over that channel. The channel type is implied by the HTTP protocol name "http" or "https" in adr. The abstract message Authenticate() queries the browser for a user authentication and triggers the browser for this purpose.

3.3 Static Model of Browser Machine B

We now define the browser's local variables $Vars_{\mathsf{B}}$. The variable space is not only a major prerequisite of the definition of the transition function, it is also the source of all information flow of B.

We distinguish *volatile* and *persistent* variables. Persistent variables hold the browser's long-term state and are only deleted by the log_off command, whereas volatile variables are deleted in every final state of an HTTP transition. We start by describing the volatile variables, which we also summarize in Table 6 of Appendix Section B.

At the beginning of an HTTP transaction a browser is directed to retrieve an address, either through a user input or derived from the previous HTTP transaction. The variable adr contains the address to be retrieved and implies the value of $host$ and ch_type. The value of ch_type specifies the type of the channel the browser establishes to the server with hostname $host$. The variables $source\_uri$ and $method$ specify the properties of the HTTP request, where the $source\_uri$ states that the entity issuing the request for adr has a URI on its own. This variable implies whether a Referer Tag is included in the request and therefore implies an information flow to the server. The variable $method$ contains the HTTP method to be used in this HTTP transition. The variable $form\_in$ contains additional input provided by the user. The variable $ch \in \mathsf{Channel}$ contains the browser's local representation of a channel established to a server, i.e., the data the browser has acquired about the channel. An element of Channel is a tuple $(cid, host, sid, type, free)$ from the domain $\mathbb{N} \times \mathsf{URLHost} \times \Sigma^* \times \mathsf{ChType} \times \mathsf{Bool}$. The variables $host$ and sid describe the server to which the browser channel is connected, where $host$ contains the hostname of the server whereas sid names the server's identity in a secure channel, and is ϵ in an insecure channel. The variable $type$ represents the channel type (secure or insecure). The variable $free$ is used to organize the reuse of existing channels and flags that the channel is currently not associated to a HTTP transaction. The variable m contains the payload of an HTTP response, whereas variable $store$ flags the user's decision whether to store login data in the browser's state.

We describe the persistent variables in Table 1. We first consider variables that are associated with every single window. Firstly, the variable wid names the identifier of the browser window. Secondly, the tuple $prev\_run$ contains data about the preceding HTTP transaction in this window. Its element ch_type contains the channel type, whereas adr contains the address retrieved in the preceding HTTP transaction. The element $form$ contains the structure of an HTML form together with hidden value fields already included in the form. The other persistent variables are global for B. The set $Channels$ contains representations of type Channel for all channels the browser has established. The following variables are important for the information flow analysis of browser-based protocols. The set $UAuth$ contains a user's login information the user decided to store in the browser's state. The persistent variable $History$ is a finite sequence of addresses successfully retrieved by the browser. The variable $Cache$ models the local browser cache. It is a finite sequence of pairs of addresses and page contents retrieved from these addresses.

3.4 Imperfections

Even correct browsers produce information flow to (i) communication partners and (ii) the underlying operating system. As this information flow may impose vulnerabilities to browser-based security protocols, we model it as explicit imperfection in the browser model.

A web browser leaks information about previous transactions and its user to communication partners in HTTP transactions. A real browser includes such information into HTTP header tags such as Referer, From or Accept_Language. We model this behavior by having the browser machine include such data in the $info\_leak$ parameter of the abstract HTTP request messages defined in Section 3.2. Thus, the browser leaks this data into the communication with each GET or POST request. We generate the list of data that flow to a server by means of the function leak2server(). This function takes the browser's current variable assignment V as argument and computes a list of name and value pairs of information to be disclosed. Information about the user may flow to the server by, e.g., the tags Accept_Language and From, however the persistent state of our browser model does not contain data that flows into these tags. Therefore, the default implementation of leak2server only includes the Referer tag into the request and therefore generates an information flow of the preceding address to the server if the request was issued by an entity with URI: $\mathsf{leak2server}(V) = (\mathsf{Referer}, V.prev\_run.adr)$ if $V.source\_uri$, else ϵ.

Real web browsers also store information on a user's machine. Most prominent examples are the local browser cache, the browsing history and the cookie storage. This behavior of a standard web browser introduces security and privacy risks especially in kiosk scenarios. We explicitly model the local cache as a persistent variable $Cache$ and the history as a variable $History$. We do not model cookies. We model the information flow introduced by the browser's behavior with the output $\mathsf{leak}(info)$ at the browser's port $\mathsf{out_B}!$ and the input do_leak at port $\mathsf{in_B}?$. These ports are free by default, such that the adversary can connect to them or they may be a specified ports of the interface to a higher protocol. This allows for flexibility in the information flow policy. Upon the do_leak command on port $\mathsf{in_B}?$, the browser outputs all its persistent state, involving $History$, local $Cache$, data about the channels opened $Channels$ and the user authentication data stored $UAuth$. Browser B generates a leak message with the string representation of this information as argument and sends it at $\mathsf{out_B}!$.

3.5 Behavior Model

For formal security proofs of browser-based protocols, one needs a precise definition of the browser's state transition function δ_B. We choose state diagrams as concise and efficient definition method for δ_B. This method allows for graph analysis of command and information flow of B which eases security proofs. A browser handles several classes of user actions asynchronously, such as enter_address, trigger_address or log_off. Upon enter_address, trigger_address or submit_form the window with the corresponding window identifier wid starts a new HTTP transaction. If there exists an ongoing HTTP transaction, that one's state flow is exited. Upon a log_off command at port $\mathsf{gui_{U,B}}?$, the browser B exits all state diagrams of HTTP transactions and starts a log-off flow, which closes all channels and deletes the browser state. Figures 3 and 4 depict the state diagram of a single HTTP transaction, i.e., an HTTP request-response pair. The start state typically corresponds to the inactive state of the browser window where the user views a page; the transaction is then triggered by the user selecting a link or directly inputting a new URL. The start state can also correspond to a user filling a form or to the middle of a redirect.

We first describe two phases that complement the channel establishment of B. The browser begins with a *local negotiation* where the browser notifies its user U about the establishment of a new channel if it is of a different type than the previous one, e.g., insecure HTTP after secure HTTPS. If the user consents to the channel change in State Channel_type_changed, the browser procures a suitable channel. We allow the browser to reuse free channels, i.e., opened yet not associated to an ongoing HTTP transaction, with the correct host and security level (*channel reuse*).

We depict the channel establishment in Figure 3. The *channel establishment* distinguishes the Secure_channel and Insecure_channel. Establishing an insecure channel is straightforward. Establishing

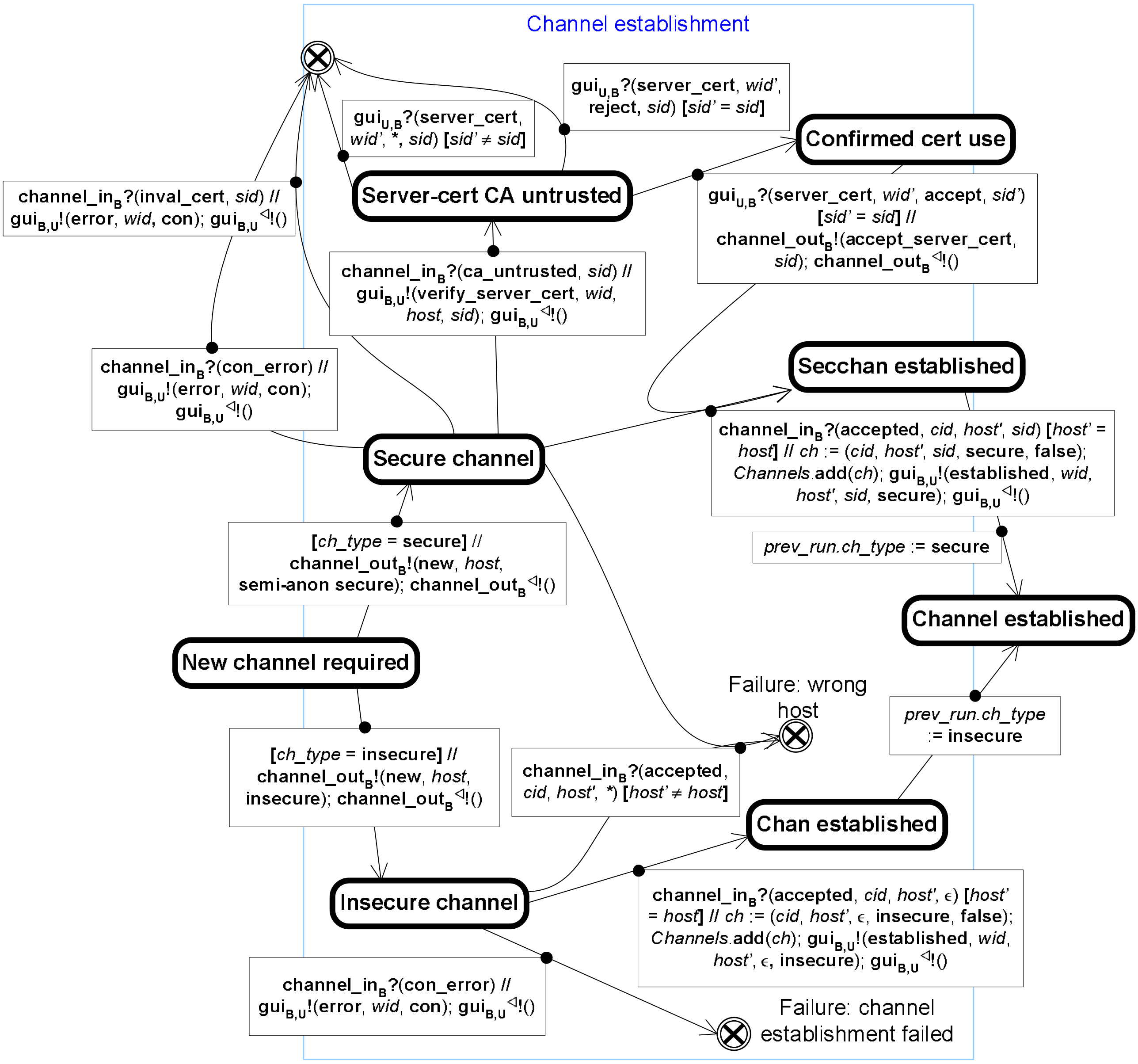

Figure 3: Channel establishment phase of an HTTP transaction. B establishes a new insecure channel in State Insecure_channel, a secure channel in State Secure_channel, or reuses a free channel in State Reusable_channel_exists.

a secure channel involves a certificate check and potential user interaction if the browser is in doubt of a certificate.

Figure 4 starts at the state Channel_established from the Figure 3 and handles HTTP requests and responses. In the *Request sending*, B issues an HTTP request as a GET or POST according to the $method$ of the initial user input and enters the state Await_response expecting a HTTP response from the server. Next B enters the *Response handling* of different abstract response types: a normal answer Page, an Error, a Redirect, scripted POST FormPOST, or an authentication request Authenticate. An Authenticate response leads to a user interaction over ports $\mathsf{gui}_{\mathsf{U,B}}?$ and $\mathsf{gui}_{\mathsf{B,U}}!$ in State Authentication_Request and finally to a resending of the HTTP request with the login information from the user. The response types Redirect and FormPOST specify an address the browser will send an HTTP request to in the following HTTP transaction. This next HTTP request is treated by the next iteration of the entire state-transition diagram, but to set up for it the browser sends a message to its own port $\mathsf{self}_{\mathsf{B}}?$, with the format accepted in the start state.

4 Ideal User Browsing Behavior U

In browser-based protocols, the browser's user has an important role. As we have seen, the browser is a state-less device that only provides a rudimental trust management. However, in higher protocols one needs to store information beyond a single transaction and have a stronger trust model. Also, the user controls most of the browser behavior and has the final say about the browser's actions. Therefore, without a user fulfilling certain tasks and properties all browser-based protocols must fail. Thus, we consider a user as active protocol participant and model it by an in general transparent machine, which, however, enforces the requirements for browser-based protocols. Firstly, the user stores data of the protocols it is involved in. Such data may be addresses of trusted servers or identity information. The user knows its trust relationship to other parties in the browser-based protocols. The machine U stores this data in its state. As the web browser is not aware of any higher-level protocol, the user acts as a supervisor of the protocol flow the browser is involved in. It checks certificates, observes the status of secure channels and logs off from the browser in error cases. Also, the user engages in the user authentication with a server and performs the crucial verification of the server's identity. The machine acts autonomically upon browser dialogs concerning these tasks.

As depicted in Figure 2, the machine U works as proxy between browser machine and the protocol interface. It forwards communication from the protocol interface to B and the browser's pages back. With the message compromised it indicates that the browser behaved contrary to the expectations of U and that U aborted the interaction with it. The user machine is connected to the browser through the user interface ports $\mathsf{gui}_{\mathsf{U,B}}!$ and $\mathsf{gui}_{\mathsf{B,U}}?$, which we described in Section 3.1 in detail.

The user machine contains confidential metadata in its state. Therefore, $Vars_{\mathsf{U}}$ is an important initial point for information flow analysis. As in Section 3.3, we distinguish persistent and volatile variables. We use similar names as in the browser machine for the volatile variables: the address adr and channel type ch_type refer to the address the browser established a channel to. For secure channels the server identity sid additionally contains the identity according to the server's certificate. The variable P refers to instances of the type Server, e.g. to servers U trusts. A Server is a tuple $(host, sid, login, sec)$ of domain $\mathsf{URLHost} \times \Sigma^* \times \Sigma^* \times \mathcal{P}(\mathsf{ChType})$ where $host$ contains the server's hostname, sid its identity in a secure channel and $login$ the login information for user U. The set sec contains the channel types allowed for a user authentication with that server. In Table 2, we introduce the persistent variables of user machine U, which we use to model the trust relationships of U. The set T_{U} contains all instances of type Server to which machine U has a trust relationship. The pairs $(host, sid)$ within this table must be unique. The set $uauth\_sec$ models the general policy of U for allowed channel types for user authentication and contains the channel types that are acceptable.

We define the state transition function δ_{U} by the state diagram in Figure 5. The Start state of this machine models a user being idle, waiting for an input from the higher protocol layer which address

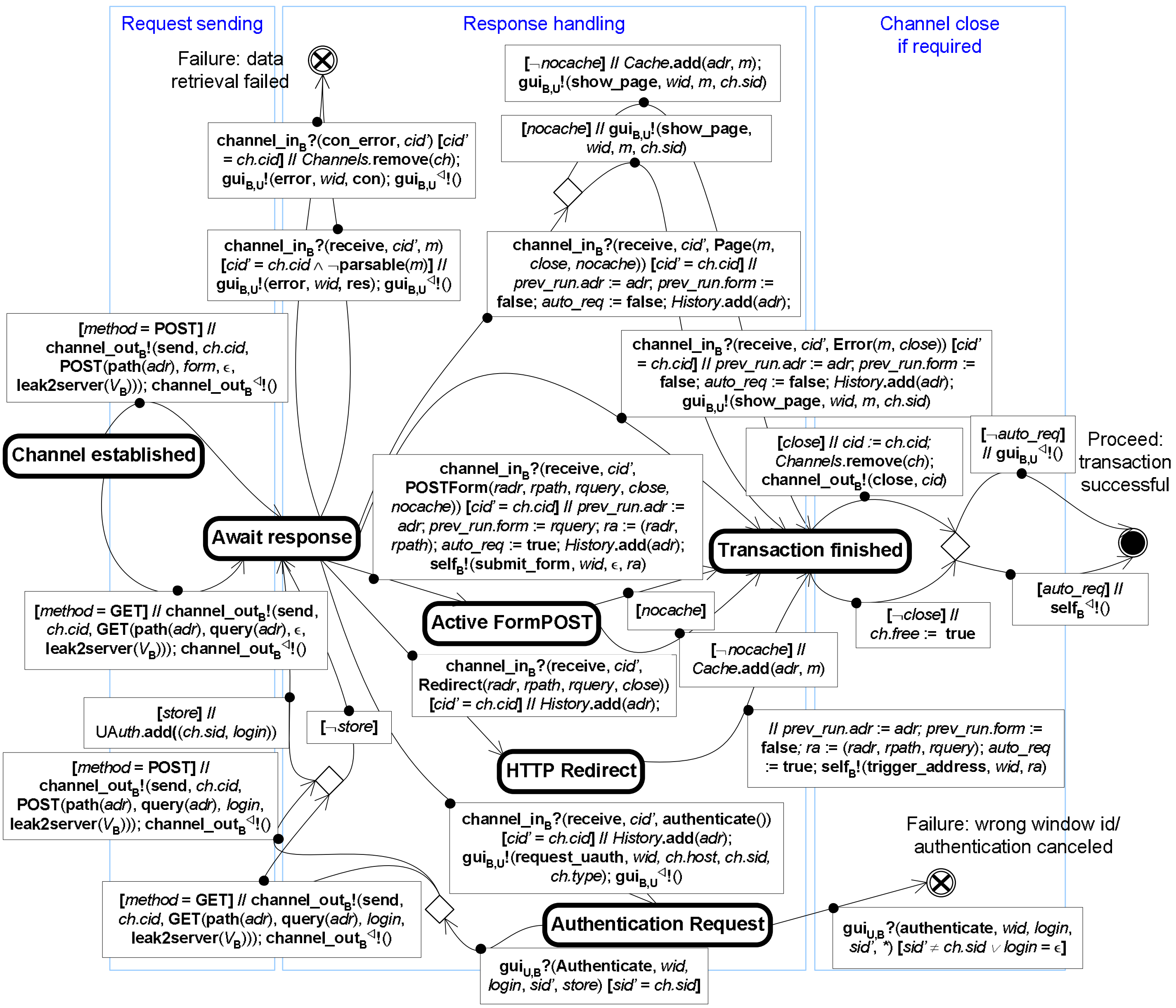

Figure 4: Request handling phase of an HTTP transaction. B issues a HTTP request in State Channel_established and handles the server's response in State Await_response. Responses Redirect and POSTForm have B request another address in the following execution of the state diagram.

Name	Domain	Description	Init.
T_U	Server*	Set of all servers trusted by U.	$\emptyset$
W_U		Identifiers of windows opened by U	$\emptyset$
$uauth\_sec$	ChType	Channel types generally allowed for uauth	{secure }

Table 2: Persistent local variables of the user machine U.

adr to retrieve or for browser events. After having issued an address request to the browser, the machine observes the browser's behavior and reacts to events generated by B. The state machine models *Transparent Behavior* on the left side (around State Honest_user_event), where it only forwards messages between protocol interface and browser. This transparent part handles the messages enter_address, trigger_address, submit_form, and show_page. The user tracks channel status changes and channels established in the States Channel_status_changed and Channelestablished, verifies certificates B doubts in State Cert_verify_requested, and forwards errors and pages to the protocol interface. The user also handles the user authentication process in State Authentication_request.

5 Channel Machine secchan

Our browser model comes with a channel abstraction secchan for secure and insecure browser channels. For space reasons, we describe this machine only partially here.

As shown in Figure 2, each machine M with network access has two ports $\mathsf{channel\_out}_\mathsf{M}!$ and $\mathsf{channel\_in}_\mathsf{M}?$ to connect to secchan. The channel machine connects to the adversary by two means. The ports $\mathsf{net\_out}_\mathsf{M}!$ and $\mathsf{net\_in}_\mathsf{M}?$ are for insecure channels and not needed here. For modeling the imperfections of secure channels, we connect the ports $\mathsf{channel\_out}_\mathsf{A}!$ and $\mathsf{channel\_in}_\mathsf{A}?$ to the adversary. The adversary also controls the network scheduling and decides which messages are delivered.

The machine secchan chooses channel identifiers uniquely and keeps track of channels. To model insecure DNS, it queries the adversary for ports corresponding to hostnames. The machine secchan has a table CA of tuples $binding = (port, sid, host) \in \Sigma^+ \times \Sigma^* \times \mathsf{URLHost}$ linking a certified identity sid and the base hostname $host$ of the corresponding security domain to the port index of a communication partner of secchan. The setup of this table enforces that the identities sid are unique. For handling a concrete channel instance, secchan dispatches the communication to a sub-machine. Such an instance contains the channel identifier cid, the port indices of the $initiator$ and $responder$, the server's actual address $host$, the server's identity rid and the security level, here secure. We depict the most important steps of a secure channel instance in Figure 6 and discuss the establishment of a secure channel in the following.

Clients initiate secure channels to an address $host$ by the command new with the parameter ch_type = secure. Then secchan queries the adversary for the recipient port index R corresponding to $host$, chooses a unique channel identifier cid, and dispatches to a sub-machine for a secure channel (Figure 6) with R, $host$, and cid as parameters. This sub-machine handles further communication. First it notifies the server with the channel identifier cid. The server may accept the channel and identify itself under an identity $rid \in \mathsf{URLHost}$ The secure channel instance verifies the server identity in State accept_request: It tests whether it has a tuple $(\mathsf{S}, rid, host') \in CA$ such that the current address $host$ lies under the base address $host'$, denoted by "$\in$". If yes, it notifies the client that the channel was accepted. From now on, the port indices of client and server are non-ambiguously bound to the unique channel identifier cid. Thus client and server are the fixed *channel partners* of this channel. Both partners may send messages referring to cid.

6 Security of User Authentication

In this section, we present the first protocol proof based on a detailed browser model: We show the security of typical password-based user authentication by one server. Such user authentication is an important building block for most other security protocols based on browsers, e.g., in federated identity management.

6.1 Authentication Server

The overall system is a special case of the architecture shown in Figure 2. We consider the definition of one server S; of course there can be several such servers and also servers of different types interacting with the same browsers and users. We only rename the free ports of this server from $\mathsf{in}_\mathsf{S}?$ and $\mathsf{out}_\mathsf{S}!$

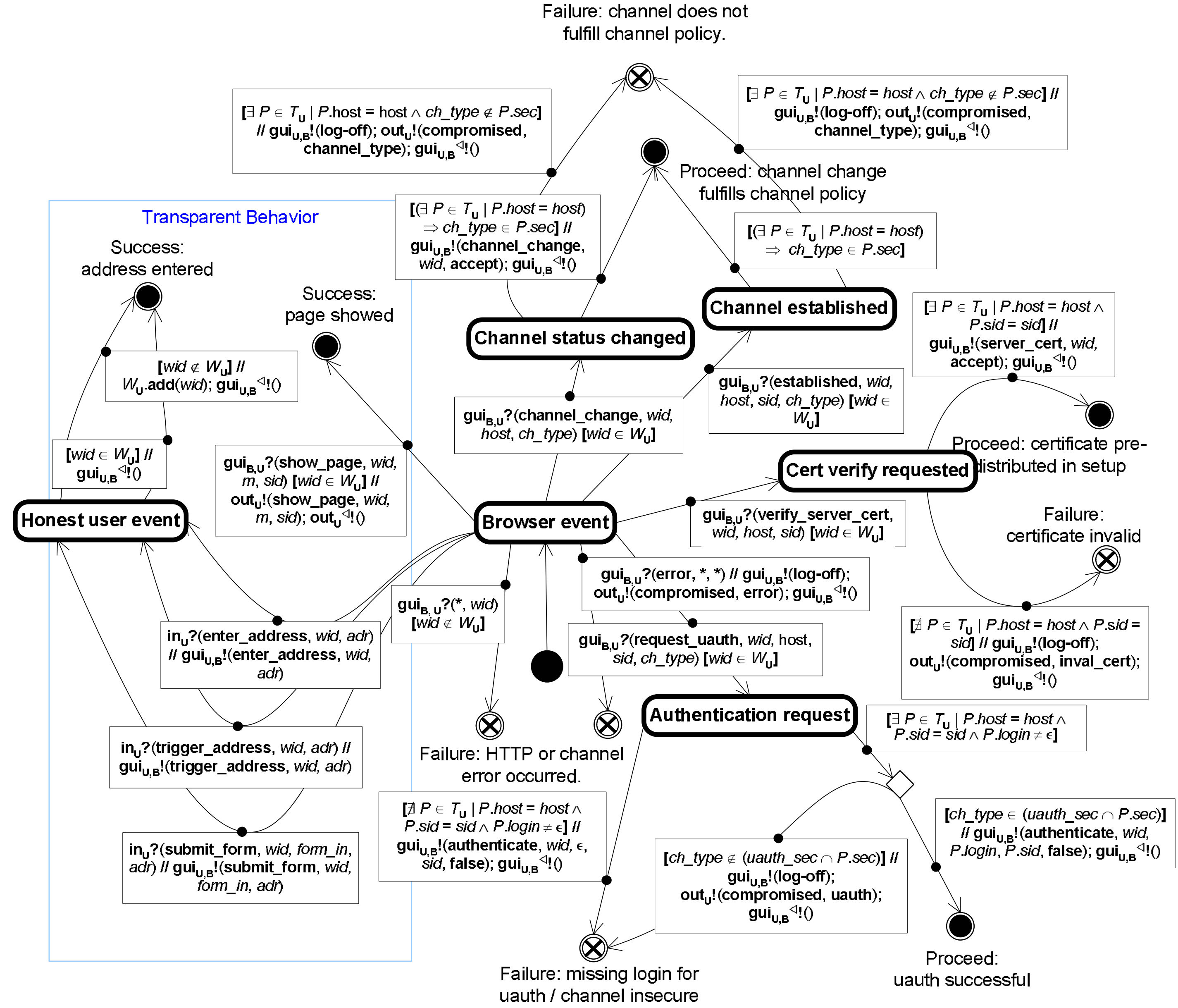

Figure 5: The ideal user browsing behavior

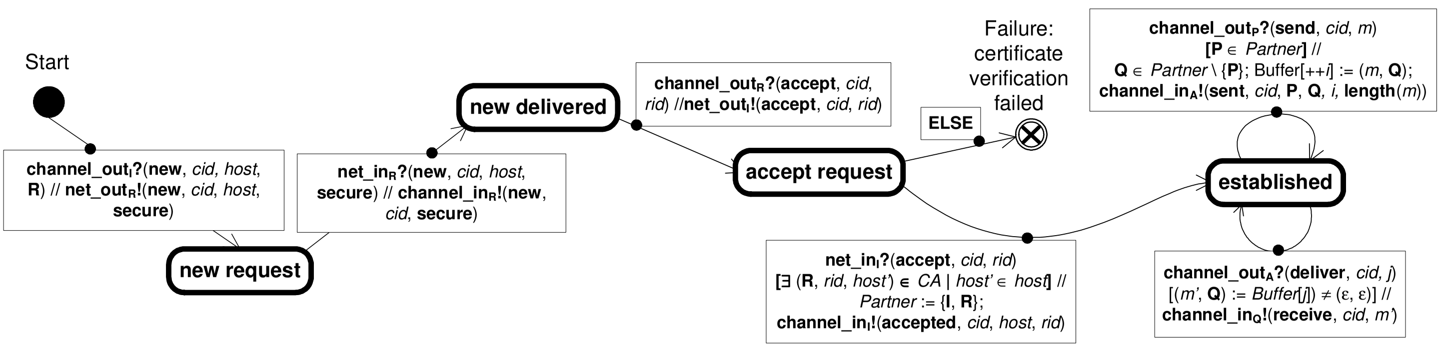

Figure 6: State diagram of a single instance of a secure channel

Port	Type	Parameters	Description
$\mathsf{uauth\_in_S?}$			Input to authentication server S
	start	$cid : \Sigma^*$	Start authentication of channel cid
$\mathsf{uauth\_out_S!}$			Output of authentication server S
	done	$cid : \Sigma^*$, $idu : \Sigma^*$	Authentication for channel cid finished with identity idu, where ϵ means failure.

Table 3: Protocol in- and outputs of the authentication server S

Name	Domain	Description	Init.
$host_S$	URLHost	Hostname of this server	See setup
sid_S	Σ^*	Identity of this server for secure channels	See setup
$MetaU_S$	$\mathcal{P}(\Sigma^+ \times \Sigma^*)$	Pairs of known user identities and login information.	$\emptyset$

Table 4: Persistent local variables of the authentication server S

into $\mathsf{uauth\_in_S?}$ and $\mathsf{uauth\_out_S!}$ to indicate that it offers a user authentication service. Further, we specialize the architecture by allowing the adversary full access to the browser's cache and history, i.e., we show that user authentication (in contrast to some other protocols) is not vulnerable to such attacks. This means that the adversary connects to all free ports in Figure 2 that are not defined to belong to the protocol interface.

The inputs at the ports that S does not share with a prior machine and its persistent variables are shown in Tables 3 and 4. We refer to the two parts of an entry e in the user metadata table $MetaU_S$ as $e.id$ and $e.login$. We require that both id and $login$ are unique within the table $MetaU_S$ of a correct server S at all times.

The state machine for one authentication protocol run of server S is shown in Figure 7. The server user (typically a higher protocol) starts authentication for some channel with identity cid. The server sends an authentication request over the channel cid. Upon receipt of an authentication message, it looks up whether the included login information is present in its user metadata. If yes, it outputs the corresponding identity as the main part of the authentication result, else ϵ.

6.2 Setup Assumptions

As set-up for a particular user machine U and authentication server S, they exchange login information $login_{U,S} \neq \epsilon$ such that U and S are the only parties that obtain information about it. Further, U must know a valid certificate identity of S so that it can verify later that it has a secure channel to S. Formally, the result of the set-up is this: The set T_U of U's trusted servers contains an entry $server_S = (host_S, sid_S, login_{U,S}, \{\mathsf{secure}\})$ with the same variables $host_S$ and sid_S as in S. The server's set $MetaU_S$ contains a pair $(id_U, login_{U,S})$, where the user's identity id_U is freely chosen by S. The binding table CA of the secure channel abstraction secchan contains a triple $binding_S = (\mathsf{S}, sid_S, host_S)$ where $host_S$ defines a security domain of S. No other variables contain

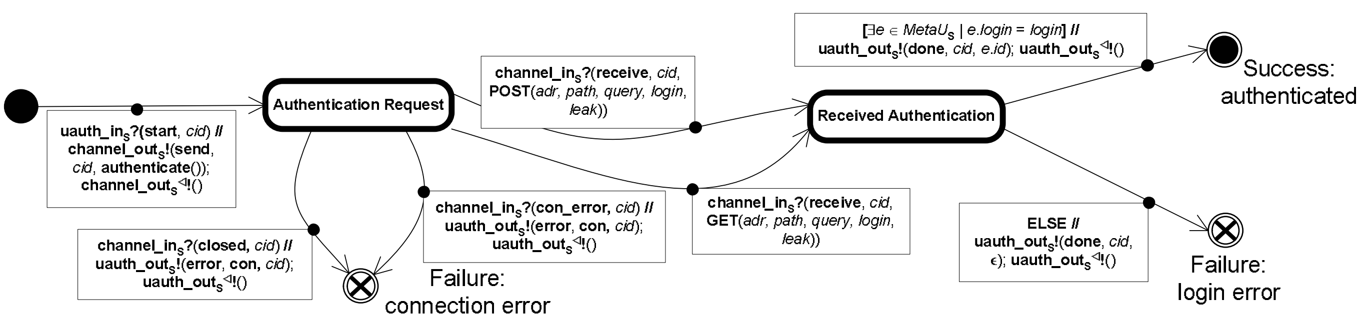

Figure 7: State machine of the user authentication server S.

information about $login_{\mathsf{U,S}}$.

6.3 Security of User Authentication

We now show that user authentication, as defined by the general user machine U and the specific authentication server S, is secure. Note that we are making a relatively strong statement: We have not required that S only makes its requests on secure channels, nor that the user correctly logs out of browser sessions or otherwise protects caches and histories. Extended protocols, e.g., the continued secure use of the channel for which the authentication is made, may need additional assumptions.

Lemma 6.1 *(User Authentication) Let a correct user machine* U *and authentication server* S *be given that have performed setup according to Section 6.2 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 identity 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* $login_{\mathsf{U,S}}$ *based on a priori knowledge of its distribution, its length, and the results of previous guessing attempts, which each exclude one potential value.* □

7 Conclusion

In prior art, browser-based protocols only came with vulnerability analyses and informal security considerations. However, those methods do not guarantee the protocols' security and do not meet the requirements of industry embracing browser-based protocols in complex scenarios. We designed the first model for the rigorous security analysis of browser-based protocols. Our model encompasses generic machines for browsers, user browsing behavior and channel abstraction that allow precise protocol proofs. We have also proven the security of the initial password-based user authentication, a very common protocol on its own and a key ingredient of browser-based protocols. In future work, we will use this model to analyze and prove the security of POST- and artifact-based protocols in the prominent area of identity federation.

References

[1] Mihir Bellare and Phillip 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.

[2] Karthikeyan Bhargavan, Cédric Fournet, and Andrew 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.

[3] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels (extended abstract). In *Advances in Cryptology: EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2002. Extended version in IACR Cryptology ePrint Archive 2002/059, `http://eprint.iacr.org/`.

[4] Scott Cantor and Marlena Erdos. Shibboleth-architecture draft v05, May 2002. `http://shibboleth.internet2.edu/docs/draft-internet2-shibboleth-arch-v0%5.pdf`.

[5] Microsoft Corporation. .NET Passport documentation, in particular Technical Overview, and SDK 2.1 Documentation (started 1999), September 2001.

[6] Zhe Dang and Richard Kemmerer. Using the ASTRAL model checker for cryptographic protocol analysis. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. `http://dimacs.rutgers.edu/Workshops/Security/`.

[7] Tim Dierks and Christopher Allen. RFC 2246: The TLS protocol, January 1999. Status: Standards Track.

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

[9] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Proc. International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 1997.

[10] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, June 1999. Status: Standards Track.

[11] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001. USENIX. An extended version is available as MIT-LCS-TR-818.

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

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

[14] Thomas Groß and Birgit Pfitzmann. Proving a WS-Federation Passive Requestor profile. In *2004 ACM Workshop on Secure Web Services (SWS)*, Washington, DC, USA, October 2004. ACM Press.

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

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

[17] Chris Kaler and Anthony Nadalin (ed.). 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/`.

[18] Richard A. Kemmerer. Using formal verification techniques to analyze encryption protocols. In *Proc. 1987 IEEE Symp. on Security and Privacy*, pages 134–138, Oakland, California, April 1987. IEEE.

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

[20] Hugo Krawczyk. SKEME: A versatile secure key exchange mechanism for the Internet. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '96)*, pages 114–127, San Diego, California, February 1996. Internet Society.

[21] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?). In *CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2001.

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

[23] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[24] Nancy 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.

[25] Catherine Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.

[26] Jonathan K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.

[27] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proc. 18th IEEE Symposium on Security & Privacy*, pages 141–151, 1997.

[28] John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state analysis of SSL 3.0 and related protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997. `http://dimacs.rutgers.edu/Workshops/Security/`.

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

[30] Lawrence Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.

[31] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.

[32] Birgit Pfitzmann. Privacy in enterprise identity federation - policies for Liberty single signon. In *3rd International Workshop on Privacy Enhancing Technologies (PET 2003)*, volume 2760 of *Lecture Notes in Computer Science*, pages 189–204, Berlin, March 2003. Springer-Verlag, Berlin Germany.

[33] Birgit Pfitzmann. Privacy in enterprise identity federation - policies for Liberty 2 single signon. *Elsevier Information Security Technical Report (ISTR)*, 9(1):45–58, 2004. `http://www.sciencedirect.com/science/journal/13634127`.

[34] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security & Privacy*, pages 184–200, 2001. Extended version of the model (with Michael Backes) IACR Cryptology ePrint Archive 2004/082, `http://eprint.iacr.org/`.

[35] Birgit Pfitzmann and Michael 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.

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

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

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

[39] E. Rescorla. Internet RFC 2818: HTTP over TLS, May 2000.

[40] Victor Shoup. On formal models for secure key exchange. Research Report RZ 3120 (#93166), IBM Research, April 1999. Version 4, November 1999, available from `http://www.shoup.net/papers/`.

[41] OASIS Standard. Security assertion markup language (SAML), November 2002.

[42] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.

A Proof of User Authentication

We now prove Lemma 6.1.

Proof. The proof begins with an information-flow analysis about the login information $login_{\mathsf{U,S}}$ that U and S shared in the setup. This analysis shows that no parties except U, S, the browser B and the channel abstraction secchan learn the login information, and that the login information never flows into other persistent variables than the original entries in T_{U} and $MetaU_{\mathsf{S}}$. The analysis further shows that the login information only passes volatile variables and interface messages as one would expect by the intended login protocol. Finally, we use this knowledge to show that S only outputs the authentication acceptance message $(\mathsf{done}, cid, id_{\mathsf{U}})$ under the conditions claimed in the lemma.

We now carry out this proof in detail. Even more rigorously, what we show in the information-flow analysis are invariants about the variables that can hold login information and the messages that the participants send with this login information.

Information flow in the user machine U. Initially, the only persistent variable in U that contains information about $login_{\mathsf{U,S}}$ is the entry $server_{\mathsf{S}} = (host_{\mathsf{S}}, sid_{\mathsf{S}}, login_{\mathsf{U,S}}, \{\mathsf{secure}\})$ in T_{U}. In the state diagram, the login information therefore occurs only as a variable $P.login$ where P is a server variable. The only read access to this variable is in State Authentication_Request.

For our case $P = server_{\mathsf{S}}$, the verification $P.login \neq \epsilon$ made in this state is always successful. Thus it does not cause any indirect information flow.

Direct information flow in this state occurs if all checks were successful and the user enters $login_{\mathsf{U,S}}$ into the browser as a message $m = (\mathsf{authenticate}, wid, login_{\mathsf{U,S}}, P.sid, \mathsf{false})$ at $\mathsf{gui}_{\mathsf{B,U}}!$. The user machine U reached this state upon receiving a login request $m_0 = (\mathsf{request\_uauth}, wid, host, sid, ch\_type)$ at $\mathsf{gui}_{\mathsf{B,U}}?$ with $host = P.host$ and $sid = P.sid$. The tests before entering $login_{\mathsf{U,S}}$ ensure that $ch\_type = \mathsf{secure}$, because we have $P.sec = \{\mathsf{secure}\}$ for $P = server_{\mathsf{S}}$.

After these actions, U enters a final state of the protocol run defining the volatile variables. Hence it sets the volatile variable $login$ to ϵ and further information flow is prevented.

Information flow in the browser B**.** This information flow is based on the login input m from the user U. A message of the form of m is only accepted in the browser state Authentication_Request (bottom of Figure 4). The browser does not add an entry (which would contain the login information) to the persistent variable $UAuth$ because for our specific message m, we have $store = \mathsf{false}$.

The value $login_{\mathsf{U,S}}$ from the input is assigned to the local volatile variable $login$. The only information flow from this variable in this protocol run is that the browser sends it to the server in a message $m_1 = (\mathsf{send}, ch.cid, \mathsf{GET}(adr.path, adr.query, login, \mathsf{leak2server}(V_{\mathsf{B}})))$ at $\mathsf{channel\_out}_{\mathsf{B}}!$. It only does this after verifying that the currently used channel ch fulfills $ch.sid = P.sid$, i.e., the recipient of this channel has the identity desired in the user input m. This implies $ch.sid = sid_{\mathsf{S}}$.

The volatile variable $login$ is set to ϵ in the final states of this protocol run, so that no further information flow from it is possible.

Information flow in secchan**.** Upon receiving the message m_1 (accepted only in state established), and because the channel with identity $ch.cid$ is secure, the channel machine secchan only outputs $m_2 = \mathsf{channel!}(\mathsf{sent}, ch.cid, \mathsf{B}, \mathsf{S}, \mathsf{length}(m_1))$ to the adversary.

If the adversary schedules the message, then the message $m_3 = (\mathsf{receive}, ch.cid, \mathsf{GET}(adr.path, adr.query, login, \mathsf{leak2server}(V_{\mathsf{B}})))$ is delivered to the channel partner of $ch.cid$. We know that B verified $ch.sid = sid_{\mathsf{S}}$ and that secchan verified during the establishment of the channel with identity $ch.cid$ that there exists a binding between sid_{S} and port S. As secchan enforces that server identities are unique, only our server S controls the server identity sid_{S}. Thus, S is indeed the channel partner of $ch.cid$ and unique recipient of m_3.

Information flow in S**.** The server S contains $login_{\mathsf{U,S}}$ in a persistent variable and further receives it in a message m_3 via a secure channel.

Upon receiving $login_{\mathsf{U,S}}$ in a message m_3 from the machine secchan as described above, it looks it up in the user metadata $MetaU_{\mathsf{S}}$. By our setup assumption, the resulting output is always $(\mathsf{done}, cid, id_{\mathsf{U}})$, which is no information flow about $login_{\mathsf{U,S}}$.

The instance of $login_{\mathsf{U,S}}$ in the persistent table $MetaU_{\mathsf{S}}$ is used whenever S receives a supposed login message, i.e., a message of the form $m_3' = (\mathsf{receive}, cid, \mathsf{GET}(path, query, login, leak))$, or the same with POST. The resulting output designates whether the input $login$ is present in $MetaU_{\mathsf{S}}$ and, if yes, contains the corresponding interface user identity id. This is the typical information flow of an online dictionary attack, which is permitted by the lemma. (Recall that $login$ is assumed to contain the account name, so we do not allow cross-account queries for passwords here.)

Proof of the lemma. We now prove the statement of the lemma based on the information flow invariants established so far. The server S only outputs $(\mathsf{done}, cid, id_{\mathsf{U}})$ in State Received_Authentication, and only if it received a message $m_3' = (\mathsf{receive}, cid, \mathsf{GET/POST}(path, query, login, leak))$ at port $\mathsf{channel\_in}_{\mathsf{S}}?$ with $login = login_{\mathsf{U,S}}$ (by the uniqueness of id_{U} in $MetaU_{\mathsf{S}}$).

It only receives m_3' if the channel partner C of cid has sent a message $m_3 = (\mathsf{receive}, cid, \mathsf{GET/POST}(path, query, login, leak))$. From the information flow analysis we know that such a message with the value $login = login_{\mathsf{U,S}}$ only occurs if this channel partner C is the browser B of user U. This finishes the proof. ■

B Details of Browser B

Table 6 denotes the volatile variables of machine B, whereas Table 5 contains the interface of B, i.e., its ports and messages expected. We describe the functions and predicates used in the browser model as follows: The I/O automata in Section 3.5 use several predefined predicates and functions. The function $\mathsf{ctype}(adr)$ with $\mathsf{ctype} : \mathsf{URLHostPath} \longrightarrow \mathsf{ChType}$ determines the channel type corresponding to the argument adr. If the address is HTTPS the channel type is secure, in other cases $\mathsf{insecure}$. The functions $\mathsf{path}(adr) : \mathsf{URLHostPath} \longrightarrow \mathsf{URLHost}$, $\mathsf{path}(adr) : \mathsf{URLHostPath} \longrightarrow \mathsf{URLPath}$, and $\mathsf{query}(adr) : \mathsf{URLHostPath} \longrightarrow \Sigma^*$ return parts of an URL argument adr. The predicate $\mathsf{fmatch}(form, form\_in)$ with $\mathsf{fmatch} : \Sigma^* \times \Sigma^* \longrightarrow \mathsf{Bool}$ checks whether the parameter names of $form$ and the user inputs $form\_in$ match. The function $\mathsf{fmerge}(form, form\_in) : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$ merges a given $form$ with the user inputs $form\_in$ to a new form. The predicate $\mathsf{parsable}(m) : \Sigma^* \longrightarrow \mathsf{Bool}$ checks whether message m is parsable according to the HTTP specification for HTTP responses. The function $\mathsf{leak2server}(V)$ with $\mathsf{leak2server} : (\Sigma^*)^{|Vars_{\mathsf{B}}|} \longrightarrow (\Sigma^* \times \Sigma^*)^*$ generates a finite sequence of name and value pairs that model an information flow from the current variable assignments V into the browser's communication with a server.

Port	**Type**	**Parameters**	**Description**
$\mathsf{in_B}$?	do_leak		Leak command from OS
$\mathsf{out_B}$!	leak	$info : \Sigma^*$	Info leakage of B to OS
$\mathsf{gui_{U,B}}$?			*Inputs from user* U
	enter_address	$wid : \Sigma^*$, adr : URLHostPath	Input in address line
	trigger_address	$wid : \Sigma^*$, adr : URLHostPath	Clicking of a link
	submit_form	$wid : \Sigma^*$, $m : \Sigma^*$, adr : URLHostPath	Submission of a form
	channel_change	$wid : \Sigma^*$, d : {accept, reject}	Consent to sec level
	server_cert	$wid : \Sigma^*$, d : {accept, reject}, $sid : \Sigma^*$	Result of cert verify
	authenticate	$wid : \Sigma^*$, $login : \Sigma^*$, $sid : \Sigma^*$, $store$: Bool	User authentication
	log_off		User logs off from B
$\mathsf{gui_{B,U}}$!			*Outputs to user* U
	error	$wid : \Sigma^*$, $type$: {con, res}	Error notification
	established	$wid : \Sigma^*$, $host$: URLHost, $sid : \Sigma^*$, ch_type : ChType	A channel was established
	channel_change	$wid : \Sigma^*$, $host$: URLHost, ch_type : ChType	Channel sec level changed
	verify_server_cert	$wid : \Sigma^*$, $host$: URLHost, $sid : \Sigma^*$	Request to verify cert
	request_uauth	$wid : \Sigma^*$, $host$: URLHost, $sid : \Sigma^*$, ch_type : ChType	Request for user auth
	show_page	$wid : \Sigma^*$, $m : \Sigma^*$, $sid : \Sigma^*$	Rendering a payload page
$\mathsf{self_B}$!, $\mathsf{self_B}$?			*Selfdelegation of browser* B
	trigger_address	adr : URLHostPath	Triggers a redirect
	submit_form	$m : \Sigma^*$, adr : URLHostPath	Scripted form submission
$\mathsf{channel\_in_B}$?			*Inputs from* secchan
	accepted	$cid : \Sigma^*$, $host$: URLHost, $sid : \Sigma^*$	Server accepted channel
	receive	$cid : \Sigma^*$, $m : \Sigma^*$	Received a message
	closed	$cid : \Sigma^*$	Server closed channel
	con_error	$cid : \Sigma^*$	Connection error notify
	ca_untrusted	$sid : \Sigma^*$	Browser does not trust CA
	inval_cert	$sid : \Sigma^*$	Cert was completely invalid
$\mathsf{channel\_out_B}$!			*Outputs to* secchan
	new	$host$: URLHost, $type$: ChType	Establish a new channel
	send	$cid : \Sigma^*$, $m : \Sigma^*$	Send a message to channel
	close	$cid : \Sigma^*$	Close channel
	accept_server_cert	$sid : \Sigma^*$	User accepted server cert

Table 5: Input and output types of browser machine B.

Name	Domain	Description	Init.
adr	URLHostPath	Address to retrieve	undef
$host$	URLHost	Hostname of an address	undef
ch_type	ChType	Channel security type for this protocol run	insecure
sid	Σ^*	Identity of a server connected by a secure channel	undef
$source\_uri$	Bool	Whether the request was triggered by an entity with an own URI	false
$method$	{GET, POST}	Method type of the HTTP request	GET
ch	Channel	Internal representation of a channel.	undef
cid	Σ^*	Unique identifier of a channel of the channel model	undef
m	Σ^*	Payload of a POSTFormS	undef
$form$	Σ^*	Form with values to be posted	undef
$form\_in$	Σ^*	User input to a form	undef
$store$	Bool	Flag whether to store the login data	false
$auto\_req$	Bool	Flag whether to send a follow-up request automatically	undef

Table 6: Volatile local variables of the browser machine B