RC24856 (W0909-021) September 8, 2009
Computer Science

IBM Research Report

Universally Composable Web Security Protocols for Delegation

Suresh Chari, Charanjit Jutla
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Universally Composable Web Security Protocols for Delegation [*]

Suresh Chari Charanjit Jutla
IBM Thomas J. Watson Research Center
Hawthorne, NY 10532.
Email: {schari,csjutla}@us.ibm.com

Abstract

With the advent of content mixing applications or mash-ups there has been a proliferation of web security protocols for secure delegation *i.e.* protocols which allow end-users to delegate to *service consumers* content which the end-user has stored at other *service providers*. The OAuth protocol[OAu07] has emerged as the unofficial standard for these protocols and has been widely adopted by many consumers and providers. Until now, these protocols have *not* been subjected to formal analysis and as recently shown[OAu09], can be vulnerable to attacks. In this paper, we rigorously analyze OAuth and related protocols using established cryptographic formalisms such as *universal composability*. We analyze a corrected version of the OAuth protocol and precisely characterize the intended *abstract functionality* and formally argue that the corrected OAuth protocol realizes this functionality. This work thus gives formal assurance that the corrected version will indeed make the protocol secure. Using the *universal composability* framework, we show the robustness of our definitions by using this abstract functionality in a larger protocol *AppStore* which captures the idea of delegated computation or in general a server-side data mash-up. Our work is the first to rigorously apply established cryptographic formalisms to the analysis of web security protocols. As part of our proof, to model the common case where the end-user authenticates to the service password using username and password, we develop a universally composable proof of *pwAKE*- a password based asymmetric key exchange where one of the parties authenticates with a password and the other is able to authenticate with a public key, a result which is of independent interest.

[*]This work is supported in part by the Department of Homeland Security through award FA8750-08-2-0091

Contents

1 Introduction

Modern web applications increasingly mix content and data from multiple trust domains offering sophisticated content aggregation applications. While these applications typically use public unprotected data, increasingly, a number of *service Providers* such as Google, Yahoo etc. permit end-users to *delegate* rights to access password protected data to other *service Consumers* who can then aggregate this protected data with other content. A canonical example is that of a photo-sharing website allowing an end-user to *delegate* to a printing service rights to access a subset of the end-user's stored photos: the user does not want to give the photo printing service his password at the sharing site but wants to grant access only to a set of photos that he wishes to get printed.

A number of web security protocols have been defined to allow for such a revocable and limited delegation. Initially, service Providers defined proprietary protocols for secure delegation(see [Aut, BBA, Ope]) but, more recently,OAuth[OAu07] has been defined as a conceptual unification to enable Consumers to operate against multiple service Providers with a single protocol. OAuth defines a wire protocol enabling the user to indicate to the service Provider the specific content which (s)he desires to delegate to the Consumer. At a very high level the protocol works as follows:

1. The Consumer requests a RequestToken from the Provider

2. The Consumer then redirects the user to the Provider with the Request Token. The User authenticates to the Provider and authorizes the delegation of content to the Consumer. The Provider then records this Request Token as authorized. and then redirects the user to the Consumer with the same RequestToken.

3. The Consumer then contacts the Provider to exchange this RequestToken for an AccessToken which it can use as needed to access the end-user's content

This high level description of OAuth omits a number of details which will be described in Section 2. OAuth has been implemented by a number of service Consumers and Providers and protocol as well as extensions[Con] are being used to secure interactions between web applications.

Recently, a *session fixation* vulnerability in this core protocol has been identified[OAu09] which exploits the fact that the end-user who authenticates to the Provider in Step 2 need not be the same as the end-user who returns the authorized token to the Consumer. Using this vulnerability, a malicious end-user can use a honest Consumer to obtain the data which the victim, another end-user, has stored with the service Provider. A number of solutions are being proposed[Fix] and at this time none of these have been officially adopted but a common theme is the use of a verification token [1] that the Service Provider generates after the RequestToken is authorized and returns to the Consumer in Step 3. The Consumer is then required to present this verification token to the Provider before the AccessToken is issued. Note that this will *only* work if the redirection is guaranteed to return to the Consumer and can not be interfered with in any manner. The details of this attack and the fixes are described in greater detail in Section 2.1.

In this paper we use the formalisms of *universal composability (UC)*[Can06] to analyze this and related web security protocols. First we define the ideal functionality for secure authenticated delegation: *A trusted third party generates and distributes to the parties pairwise shared secrets.* We note here that the original OAuth protocol with the vulnerability has an ideal functionality of

[1]It is assumed that this verification token binds the three parties together.

three-party entity authentication. The distinction between entity authentication and key exchange has been previously highlighted[BR95]. We show the robustness of our characterization by defining a larger protocol *AppStore* which captures the idea of delegated computation: *AppStore* consists of three parties (the Consumer, Producer and the End-User). In the ideal functionality for *AppStore* the End-User first stores content m at the Provider. Then the ideal functionality computes $f(m)$ (intended to represent the function computed by the Consumer) and sends this to the end-user. We show *AppStore* can be implemented by a hybrid protocol consisting of our abstract functionality for secure authenticated delegation and other existing functionalities for secure channels and signatures.

We formally prove that the corrected version of the handshake (Steps 1 and 2) of the OAuth Protocol described above correctly realizes the ideal functionality for secure authenticated delegation. Our work therefore gives the assurance that the corrections being considered for OAuth are indeed valid and will correctly implement secure delegation and hence justifies their inclusion in the protocol. The Universal Composability framework guarantees that any higher level protocol which uses the ideal functionality for delegation can be implemented by replacing this with the OAuth protocol.

The OAuth protocol does not specify how the end-user authenticates to the Service Provider but the most common usage of this protocol in web applications is when the end-users authenticate to the Service Provider using a username and password provided over a secure channel (SSL) with server authentication via public keys. Halevi *et al* [HK98] considered this problem before but do not give a protocol which is secure in the universally composable sense. While UC protocols for password based key exchange are known previously[CHK$^+$05] when both parties authenticate with passwords there are no UC proofs of the aasymmetric case As part of this proof we give a universally composable proof of *pwAKE*, a cryptographic primitive which may be independent interest: *pwAKE* is the password based asymmetric key exchange protocol where one party uses a password to authenticate but the other party can authenticate using a public key. This is in a harder case to deal with due to the possibility of dictionary attacks etc.and the proofs of security of OAuth with other authentication mechanisms such as public-keys should be simpler.

This paper is organized as follows: Section. 2 describes the details of the OAuth Protocol and describes the session fixation vulnerability and the proposed fixes. Section 4 introduces the ideal functionality *AppStore*, OAuth and shows how to implement *AppStore* using the ideal functionality for OAuth. Section 5 shows a protocol which realizes the ideal functionality of OAuth using a functionality for *pwAKE*.

2 Secure Delegation using the `OAuth` protocol

This section describes the OAuth protocol and how it is used for secure access delegation. While our focus is on OAuth, the formal analysis in the following sections can be adapted to AuthSub[Aut], BBAuth[BBA] and other protocols.. OAuth captures the commonality in function of all these protocols: allowing Consumers access to and end-user's content stored at a service Provider without requiring the end-user to hand over their credentials to the consumer. To participate in an OAuth transaction the Provider and Consumer have to perform the following bootstrap steps:

1. Agree on a name for the Consumer: This is referred to as the `consumer_key` in the protocol. Despite its label, this parameter is just a textual name of the Consumer.

2. Exchange shared keys and/or public-key/certificates so that the Provider can verify requests

Message Flows in the OAuth Protocol version 1.0

Participants: A Service Consumer and a Service Provider which are server entities and the end-user. **Key Steps:** To begin this process, a specific end-user contacts the Consumer wishing to authorize the Consumer to access his/her protected content at the Provider.

1. **Obtain Unauthorized RequestToken**: Service Consumer obtains an unauthorized RequestToken from the Provider. The message identifies the Consumer with the `consumer_key` and is signed by the consumer.
2. **Redirect EndUser to authorize RequestToken**: Consumer redirects the end-user's browser to a Service Provider URL. Here the Provider is expected to authenticate the user and identify the Consumer. The user is then expected to authorize the RequestToken and specify for exactly which content (s)he is delegating access to the Consumer.
3. **Redirect User to Consumer Callback URL**: After user authorizes the RequestToken the Provider notes that this token is now authorized and redirects end-user to the conumer callback URL. This URL may optionally be fixed at bootstrap time for this consumer.
4. **Consumer exchanges authorized RequestToken for an Access Token**: Consumer contacts Service Provider with the RequestToken. If this RequestToken is marked as authorized, the Service Provider returns an AccessToken to the Consumer.
5. **Consumer accesses end-user content**: Consumer includes AccessToken in requests for protected end-user's content. If AccessToken is valid and the request is signed by the consumer, the Provider sends back the end-user's delegated content.

Figure 1: Overview of the OAuth Protocol Version 1.0.

by the Consumer. The Consumer either signs the request with a public key or attaches a MAC using a shared key and to verify they exchange the shared-key or public keys.

3. Fix other essential parameters: For instance, the *callback URL* where the Provider redirects the user's browser to can be agreed to and verified in each transaction.

With these bootstrap steps, these parties can participate in an OAuth enabled transaction. In Figure 1, for convenience, we refer to Steps 1 through 3 as the Handshake portion of the OAuth protocol. The following are notable features of OAuth version 1.0:

- Messages from the Consumer to the Provider(Steps 1, 4 and 5 in Fig. 1) identify the consumer by name *i.e.* `consumer_key` and are signed by the Consumer.
- The RequestToken obtained in Step 1 is conveyed to the Provider in Step 2 of the handshake and after successful authorization the Provider returns *exactly* the same token back to the Consumer in step 3 and is then exchanged for an access token.
- Typically the Service Provider will redirect the end-user back to a previously agreed to URL *i.e.* verify that the URL provided by the Consumer in Step 2 is agreed to at bootstrap.

2.1 Session Fixation Vulnerability

Recently a vulnerability has been discovered in OAuth which cleverly exploits the fact that there is nothing in the handshake to enforce that the end-user whose first contacts the Consumer and whose browser the Consumer attempts to redirect in step 2 is the same as the end-user who authorizes the token at the Provider. A malicious user $\mathcal{M}$ can exploit this vulnerability as follows:

- To initiate an OAuth based transaction $\mathcal{M}$ first contacts the Consumer. When the Consumer redirects $\mathcal{M}$'s browser to the Provider, $\mathcal{M}$ stops the flow.
- $\mathcal{M}$ then induces the victim end-user $\mathcal{V}$ to follow this link to the Provider, typically by social engineering means. Assuming $\mathcal{V}$ can be tricked into visiting the Producer site, (s)he is typically confronted with a form to authorize content access by the Consumer.Once $\mathcal{V}$ authorizes the RequestToken the Producer will readily provide an AccessToken for $\mathcal{V}$'s content.
- If the Producer will redirect $\mathcal{V}$ to *any* URL given to it in step 2, $\mathcal{M}$ can insert a site under its control. Even if only the Consumer controls the redirection URL, $\mathcal{M}$ can attempt to continue the protocol with the consumer from Step 1 as soon at it is authorized.
- The Consumer then obtains an AccessToken and use it to access the content of $\mathcal{V}$ who is the user who authorized this token. Hence $\mathcal{M}$ can illegally view the content belonging to $\mathcal{V}$.

This vulnerability has not been exploited in any implementation but it underscores the importance of formal analysis of protocols.

2.1.1 Fixing the OAuth Protocol

Since the announcement of the vulnerability a number of solutions[Fix] have been proposed and are under consideration. The two issues fixed are that the Consumer has to be able to control where the end-user is being redirected after authorization and that the Provider issues a token after authorization in the redirect back to the Consumer which *binds* the three parties in the transaction together with the RequestToken. The first fix ensures that $\mathcal{M}$ can not force the Provider to redirect the end-user to a site under his control while the second ensures that the Consumer can not get an AccessToken without a verification token obtained through the user who authorized the delegation.

Concretely these fixes are implemented in the following modifications to the Handshake:

1. **Modification to Obtaining a RequestToken**: While obtaining a RequestToken from the ServiceProvider the Consumer also provides a `callback_url`. The Provider is expected to redirect end-users to this URL after authorization.
2. **Redirect EndUser to authorize RequestToken**: This step is as before
3. **Modification to Redirect User to Consumer Callback URL**: After user authorizes the RequestToken the Provider redirects to the `callback_url` from Step 1 along with a *verification token* which binds the parties to the RequestToken.
4. **Modification to Consumer request for Access Token**: Consumer must provide the *verification token* from Step 3 to obtain an AccessToken

This and other similar fixes are under consideration. In the following sections we will formally prove that this fix to the protocol will correctly realize the intended functionality.

3 UC Preliminaries

In this paper we will use the universally composable framework for proofs of security [Can06]. For easy reference, protocol executions in the UC framework are provided in Figs 6,7,8 in Appendix A.3. We will also use many ideal functionalities such as secure channels, authenticated communication, and public key encryption, which have been previously defined in the literature. These functionalities are defined in Figs 9,11,10,12. In its simplest form, the universal composition theorem [Can06] states that if protocol π realizes some ideal functionality $\mathcal{I}$ in the $\mathcal{F}$-hybrid model then π^ρ securely realizes $\mathcal{I}$, where ρ is a protocol that realizes ideal functionality $\mathcal{F}$. Please refer to [Can06] for formal definitions.

4 Ideal Functionalities for Delegation

In this section we define the ideal functionalities for secure authenticated delegation and delegated computation (or more generally server side mash-ups). First we define the ideal functionality for *AppStore* which captures the idea of delegated computation at a Consumer with data that a user has stored at a Provider. We then define $\mathcal{F}_{\text{OAuth}^*}$ which is the ideal functionality for the OAuth protocol's handshake. Section 4.3 shows a hybrid protocol which implements *AppStore* using the ideal functionality $\mathcal{F}_{\text{OAuth}^*}$.

4.1 The Ideal Functionality for Delegated Computation

This section defines the ideal functionality *AppStore* for delegated computation. The functionality involves three parties the *User*, the *Store*, and *the Application*. The *Store* and *Application* are intended to model the service Provider and Consumer respectively. The functionality models the user storing personal content at the *Store*, a subset of which the user wants to reveal to the *Application* to compute an arbitrary function on. It codifies the following sequence: Initially, the user P_k uploads a sequence of messages to a server P_j (*Store*). At some point of time, the user, possibly driven by the environment, also decides the identity P_i (*Application*) of the third party application, as well as the particular function f which is to be computed on data stored at P_j. The party P_i indicates to the functionality that it needs a certain subset of the data stored at P_i, specified, in general, by a probabilistic polynomial time Turing machine r (called the filter function). This is intended to capture the delegation of only a subset of data stored at the Provider and can be seen as as a restriction of the user's rights at the Provider. To model the idea of additional information that the application may use in the computation (to represent a server side data mash-up) P_i may also specify some proprietary or secret information z which is to be passed as a parameter in the evaluation of f (the user P_k leaves it to the application's discretion as to what value z is used). After, the user P_k and store P_j authenticate that the filtered (by r) information can be given to P_i, the functionality returns $f(z, r(\langle m_t \rangle))$ to the user P_i. The details of this ideal functionality are given in Fig 2.

4.2 The Ideal Functionality for Authenticated Delegation and Key Exchange

In this section we define the ideal functionality for the handshake in the corrected OAuth Protocol *i.e.* with the *verification token* issued by the Producer as described in Section 2.1.1. Before we

Functionality $\mathcal{F}_{\textbf{App,Store}}$

Participants: The User (P_k), Store (P_j) and Application (P_i).

Set-up: $\mathcal{F}$ receives input (setup-store,P_k,P_j,sid) from P_k. $\mathcal{F}$ writes public-delayed (*i.e.* sends the input to adversary $\mathcal{A}$, and waits for a `deliver` message) the message on P_j's subroutine tape for $\mathcal{F}$ and locally associates sid with P_k as the user, and P_j as the store.

Upload: $\mathcal{F}$ receives (upload, P_k, sid, m_t) from P_k. $\mathcal{F}$ writes private-delayed (*i.e.* sends the length of the input to adversary $\mathcal{A}$, and waits for a `deliver` message) the message on P_j's subroutine tape , and appends m_t to its local list of messages meant to be stored at P_j (for P_k). In practical terms this models all the user's personal data which is maintained by the service Provider.

Compute:

1. $\mathcal{F}$ receives a message (compute, P_i, P_k, P_j, f, sid) from the user P_k and public-delayed forwards it to P_i. The application P_i is meant to compute, on behalf of P_k, the function f on the list of messages stored at P_k. In practical terms this models the step where the end-user picks a specific Consumer who will be delegated access to the content stored at the Provider to perform a server-side mashup using this data.
2. After $\mathcal{F}$ receives a reply (filter, r, z, sid) from P_i, it public-delayed forwards (filter, P_i, P_k, P_j,r, sid) to both P_k and P_j The function r signifies a function which computes a restriction of the data stored by P_k at the store P_j and z signifies a secret additional input that P_i supplies as an additional parameter to the function f.. The models the Consumer using a subset of the end-user's personal data stored at the service provider as defined by the restriction r and performing s server-side mashup of this restricted data along with additional data defined by z.
3. After both P_k and P_j respond in the affirmative to $\mathcal{F}$, the functionality $\mathcal{F}$ informs the adversary $\mathcal{A}$ of the affirmative response. This models the end-user granting the Consumer the rights to access the requested restriction of the user's data.
4. On $\mathcal{A}$'s response, the functionality writes $f(z, r(\langle m_t \rangle))$ on the subroutine tape of P_k. The adversary is informed of the length of this response, as well as the length of $r(\langle m_t \rangle)$.

Corruption: We do not model forward security on corruption of various parties, but various such models can be defined. Note that on corruption of P_i, the adversary only gets $r(\langle m_t \rangle)$ and z.

Figure 2: A remote storage and a third party application functionality

do that we note that the original OAuth protocol is captured by the ideal functionality which achieves three way authentication of parties *i.e user* (P_k), the *consumer* (P_i), and the *service provider* (P_j). This ideal functionality is in Appendix A.4. However, as observed in many earlier works, authentication without a key exchange is more or less useless even in the case of two party authentication (e.g. [BR95]). We now define an ideal functionality OAuth*, which achieves an explicit key exchange after the authentication has completed. There are *two* independent keys exchanged, one between the user and consumer, and another between the consumer and the service provider. The formal definition is given in Fig 3.

This functionality also assures *forward security* [CK02] on corruption of each of the parties. It gives the consumer two (independent) keys, one which is shared with the user and the other with the service provider. It is clear from the description given in Fig 3 that if none of the three parties

Functionality $\mathcal{F}_{\textbf{OAuth}^*}$

Particpiants: The User (P_k), Service Provider (P_j), Consumer (P_i), and Adversary $\mathcal{A}$ interact with functionality $\mathcal{F}_{\text{OAuth}^*}$ (or $\mathcal{F}$).

Initiate: On receiving a subroutine input message (*sid*, P_i, P_k, P_j, r, *initiator*) from P_i, the functionality $\mathcal{F}$ public-delayed writes the message to P_k and to P_j and records locally the state of the protocol. Here r is meant to be a filter function.

Authentication Response: On receiving a subroutine input message from P_l ($l \in \{k, j\}$) of the form (*sid*, P_i, P_k, P_j, *response*), $\mathcal{F}$ forwards the message to $\mathcal{A}$. $\mathcal{F}$ records the state change locally, and if both P_k and P_j have responded, it sets the local state to *active* and chooses κ_1 (intended to be the shared key between P_i and P_j) and κ_2 (indended to be the shared key between P_i and P_k) uniformly and independently at random. This is the abstracttion of the handshake of the OAuth protocol and the functionality generates pairwise shared keys as a result of the handshake.

Activation/Key delivery: This step models the delivery of the generated pair-wise key. To model corruption, this allows the adversary to replace the generated pairwise shared key by an arbitrary value if the parties are compromised. In particular, if the adversary responds to $\mathcal{F}$ with a "deliver key" message (*sid*, P_s, κ'_1, κ'_2) ($s \in \{i, k, j\}$), and the local state is *active*, it does the following based on s

$s = i$**:** If P_k is already corrupted, set γ_2 to κ'_2, else set γ_2 to κ_2. Also, if P_j is already corrupted, set γ_1 to κ'_1, else set γ_1 to κ_1. Write (*sid*, γ_1, γ_2, *key*) on subroutine tape of P_i.

$s = k$**:** If P_i is already corrupted, set γ_2 to κ'_2, else set γ_2 to κ_2. Write (*sid*, γ_2, *key*) on subroutine tape of P_k.

$s = j$**:** If P_i is already corrupted, set γ_1 to κ'_1, else set γ_1 to κ_1. Write (*sid*, γ_1, *key*) on subroutine tape of P_j.

Corruption: On corruption of any one of the parties P_s ($s \in \{i, k, j\}$) *after* the local state of $\mathcal{F}$ has been set to *active*, the functionality takes the following action based on s

$s = i$**:** If it has already received a "deliver key" to P_k message from $\mathcal{A}$, and has not yet received a "deliver key" to P_i message, then send κ_2 to $\mathcal{A}$. Similarly, send κ_1 to $\mathcal{A}$ (with P_j replacing P_k in the condition).

$s = k$**:** If it has already received a "deliver key" to P_i message from $\mathcal{A}$, and has not yet received a "deliver key" to P_k message, then send κ_2 to $\mathcal{A}$.

$s = j$**:** If it has already received a "deliver key" to P_i message from $\mathcal{A}$, and $\mathcal{F}$ has not yet received a "deliver key" to P_j message, then send κ_1 to $\mathcal{A}$.

Figure 3: A functionality for delegation with explicit key exchange

are corrupted, then they indeed get random and independent keys, i.e. the user and the consumer end up with the same random key κ_2, and the consumer and the service provider end up with the same random (and independent of κ_2) key κ_1. In this paper we use the convention that if a party P is corrupted by adversary $\mathcal{A}$, then $\mathcal{A}$ can write to the environment on behalf of P (even in the ideal world, where P is just a dummy). Thus, e.g. if P_i is corrupted and the other two parties are not corrupted, before delivery to P_i (with κ'_1, κ'_2) has been called by $\mathcal{A}$, the functionality just writes (κ_1, κ_2) to P_i. However, since P_i is already corrupted, the adversary can just ignore it and write (κ'_1, κ'_2) to the environment as output of P_i. The information given to the adversary $\mathcal{A}$ on corruption of different parties is defined so as to model forward security. For example, if P_k is corrupted, after the key has already been instructed to be delivered to P_k, he adversary *does not* get the key κ_2. this is because, by then the user P_k may already have used the key and erased it.

4.3 Implementing $\mathcal{F}_{\textbf{App,Store}}$ using $\mathcal{F}_{\textbf{OAuth}^*}$

In this section we show how the functionality $\mathcal{F}_{\text{App,Store}}$ can be built in a hybrid model which has access to ideal functionality $\mathcal{F}_{\text{OAuth}^*}$. As mentioned earlier, we assume that the application and the service provider have public keys. We assume that the user does *not* have publicly registered (or certified) public keys, and hence has no way of naming itself globally. However, it usually can setup passwords with parties having public keys very easily, e.g. by just choosing a password and encrypting it with the public key and sending across. However, the password and the userid it has with different public key entities cannot be assumed to be linked in any way, and one must formally prove protocols secure by using different invocations of password based protocols, e.g. key exchange. We note here that in this asymmetric setting where the server has a public key and the user has a userid/password with the server, a password based *secure channel* functionality can be realized. As we show in Section 5, this can be done by first doing a password based asymmetric key exchange and then using a MAC and a pseudorandom generator to implemented secret and authenticated exchange of messages.

For ease of exposition, we assume that the user (P_k) has access to the secure channel functionality $\mathcal{F}_{\text{SC}}$ (see 12) with (*at most*) one public key entity. We can only allow at most one such channel, as $\mathcal{F}_{\text{SC}}$ allows global naming of parties. With this assumption, we have that P_j can authenticate the name P_k, across different functionalities, including $\mathcal{F}_{\text{OAuth}^*}$. This allows for the crux of the proof idea to come through, without getting bogged down in details. Although, the application P_i cannot authenticate the user with the same name P_k, the implementation of $\mathcal{F}_{\text{OAuth}^*}$ in section 5.2 shows that only P_j needs to authenticate P_k, and hence overall we are only assuming that P_j authenticates P_k in both the secure channel functionality and in OAuth*.

The protocol for $\mathcal{F}_{\text{App,Store}}$ in this hybrid model is then straightforward. First user P_k and the store P_j engage in a secure channel session, and the user uploads all the messages to P_j. Subsequently, P_k sends an unauthenticated message to P_i to initiate OAuth* involving P_i, P_k and P_j, and using function f. Then, P_i initiates $\mathcal{F}_{\text{OAuth}^*}$ with role initiator, and a function r. After both P_k and P_j respond positively to $\mathcal{F}_{\text{OAuth}^*}$, the adversary drives the distribution of keys. At the end of $\mathcal{F}_{\text{OAuth}^*}$, let P_i hold keys κ_{ij} and κ_{ik}. Similarly, let P_k hold the key κ_{ki}, and let P_j hold the key κ_{ji}. If there was no corruption, then $\kappa_{ij} = \kappa_{ji}$, and similar equalities hold for other pairwise keys as well. Hence, using a secure MAC and a pseudorandom generator (just as in [CK02], or see Appendix Claim 3), we effectively have a secure channel ideal functionality between P_i and P_k, and between P_i and P_j. Using the secure channel functionality, P_k first authenticates to P_i that it indeed wants to compute function f. Also, using the secure channel between P_j and P_i, the store forwards $r(\langle m_t \rangle)$ to P_i. Thereafter, P_i computes $f(z, r(\langle m_t \rangle))$, and returns it to P_k using the secure channel.

A formal definition of password based AppStore functionality, a new functionality ($\mathcal{F}_{\text{OAuth}^*\text{-aug}}$) which is the special case of OAuth* where the end-user authenticates to the Service Provider using paswords,and a proof of realization of this AppStore in a hybrid model including $\mathcal{F}_{\text{OAuth}^*\text{-aug}}$, and password based asymmetric key exchange will be given in the full paper.

5 Realizing the functionality $\mathcal{F}_{\mathbf{OAuth}^*}$

In this section we want to build a real protocol which realize the ideal functionality $\mathcal{F}\_oauthstar$. To do this we need to consider the authentication mechanism between the end-user and the Service Provider. The OAuth protocol does not specify this mechanism and leaves it up to the Service provider.Most web applications use the asymmetric authentication mechanism where the end-user authenticates to the Provider with a username and password over a secure channel such as an SSL/TLS session with server authentication using public keys and certificates.We first define a functionality for password based asymmetric key exchange and show how this can be used to realize the functionality $\mathcal{F}_{\mathrm{OAuth}^*}$ for the case when the end-user authenticaion to the Provider is via username/password. Note that we would have a different proof for other cases (such as for e.g. when the end-user can authenticate using a public-key) but we believe that those proofs will be simpler than the case we consider.

5.1 Ideal Functionality for Password Based Asymmetric Key Exchange

We define an ideal functionality for password based asymmetric key exchange which has some important differences from an ideal functionality for password based key exchange defined in [CHK$^+$05]. The differences stem from the fact that the latter models a key exchange where none of the parties have a public key, and the only information on which they base mutual authentication and key exchange is a password. However, here we model a situation where one of the parties has a public key (certified by a trusted authority). As we will later see in Section 6, the new functionality has a much simpler implementation (i.e. more or less what is used in practice; see e.g. [HK98]).

The new functionality is defined in Fig 4 and at a high level functions as follows: it first allows two parties, one in a client role and the other in a server role, to register a password for a given session. Then as directed by the adversary, *if the registered passwords for a session match*, it generates a new random session key and sends it to the two parties. This functionality mirrors the Strong Key Exchange functionality $\mathcal{F}_{\mathrm{KE}}$ described in Figure. 11 which abstracts the key exchange when the two parties have public keys.

The functionality $\mathcal{F}_{\mathrm{pwAKE}}$ incorporates a number of stronge features and additional properties which make it easier to use in realistic situations. For instance, it allows the parties to explicitly abort the key exchange session, as opposed to agreeing on different keys (i.e. random and independent keys). The functionality models adversaries who can try to guess the password for this session: when the adversary issues a `TestPwd` query with a wrong password, the server record is marked `interrupted`. This models the case where the server gives the client *only* a single wrong guess of the password and can be easily extended to model the case when a small finite number of wrong guesses are permitted. Later, when issuing the key to the server the functionality can definitely abort. Further, if the server is not corrupted in the meanwhile, the client can also abort. We note here that the `TestPwd` query can only be made against the server record. One consequence of the stronger definition is that the situation for `NewKey` issuance for the server and for the client are slightly different, and hence the asymmetry in the definitions too.

Finally, we remark that the definition includes a `DOS` (Denial of Service) call from the adversary. This models the fact that an adversary can always delay message delivery to the server indefinitely, and if a server has to explicitly abort a session, one must provide an ideal functionality to model timeouts. A similar need arises, if the public key (which is adversarially chosen in $\mathcal{F}_{\mathrm{PKE}}$) produces

Functionality $\mathcal{F}_{\mathbf{pw}\mathrm{AKE}}$

The functionality $\mathcal{F}_{\mathrm{pwAKE}}$ is parametrized by a security parameter k. It interacts with an adversary S and a set of parties as follows:

New Session: On receiving an input (`NewSession`, *sid*, *pw*, role) from a party P_a, the functionality sends (`NewSession`, *sid*, P_a, role) to S, as well as records (*sid*, P_a, *pw*, role) locally as `fresh`. The role can be either *client* or *server*, and for each role, only one such input is accepted.

Test: On receiving a message (`TestPwd`, *sid*, P_i, *pw'*) from S, if there is a record (*sid*, P_i, *pw*, *server*) which is `fresh`, then do: if $pw = pw'$, mark the record as `compromised`, and reply to S in the affirmative. Otherwise, mark the record as `interrupted`, and reply to S in the negative.

Denial of Service: On receiving a message (`DOS`, *sid*) from S, and if the server record is marked `fresh`, then mark it as `interrupted`.

New Key for Server: On receiving a message (`NewKeyServer`, *sid*, P_i, sk) from S, where $|sk| = k$, and if there is a record of the form (*sid*, P_i, *pw*, server) which is not marked `completed`, *and* there is a record (*sid*, P_j, *pw'*, client), the functionality does the following:

- If the server record is marked `compromised` then output (*sid*, sk) to P_i.
- Else, if the server record is marked `interrupted`, or the passwords in the server and the client records are different, then output (*sid*, aborted) to P_i.
- Else, if P_j is corrupted then output (*sid*, sk) to P_i.
- Else, if a key sk' was already sent to P_j, then send (*sid*, sk') to P_i.
- Otherwise, pick a random key sk' of length k and send (*sid*, sk') to P_i.

In all cases, mark the server record as `completed`.

New Key for Client: On receiving a message (`NewKeyClient`, *sid*, P_j, directive) from S, where directive is issue/abort, and if there is a record of the form (*sid*, P_j, *pw*, Client) which is not marked `completed`, *and* there is a record (*sid*, P_i, *pw'*, server), the functionality does the following:

- If P_i is corrupted, and if directive is abort, then send (*sid*, aborted) to P_j.
- Else, if P_i is corrupted or the server record is marked `compromised` send a new random key of length k to P_j.
- Else, if the server record is marked `interrupted`, output (*sid*, aborted) to P_j.
- Else, if the passwords in both records are the same, and a key sk' was already sent to P_i, then send (*sid*, sk') to P_j.
- Else, if the passwords are not the same, then send (*sid*, aborted) to P_i.
- Otherwise, pick a random key sk'' of length k and send (*sid*, sk'') to P_j.

In all cases, mark the client record as `completed`.

Figure 4: A Functionality for Password based Asymmetric Key Exchange

too many colliding ciphertexts, and hence the server fails to uniquely decrypt ciphertext messages. This can be seen as a denial of service attack, and in fact our proof of security of Theorem 2 uses this DOS call specifically to simulate this situation.

As is standard, the functionality models explicit corruption of any of the parties by allowing the adversary to provide random keys to distribute to the parties. The details of this functionality are given in Fig. 4.

5.2 Protocol for OAuth* using Password Based Asymmetric Key Exchange

In this section we describe a protocol which implements OAuth* in a hybrid model using ideal functionalities $\mathcal{F}$-auth for message authentication, the functionality for public-key encryption $\mathcal{F}_{\mathrm{PKE}}$, and the functionality for password based asymmetric key exchange $\mathcal{F}_{\mathrm{pwAKE}}$ define above. We note that the the functionality $\mathcal{F}$-auth can only be used for sending messages by parties which have publicly known (or certified) public keys. Note that we can define similar implemetations of OAuth* iwhich use the functionality for public key signatures (instead of $\mathcal{F}$-auth), but this would lead to a cumbersome proofs involving certification using global registration. So, we prefer this former modelling for ease of exposition, and we refer to it as the **restricted hybrid model**. Note that in the real-world server authenticaiton is done in SSL/TLS by public key signatures and registration.

As we described, we will be modeling the authentication of the end-user and Provider using password based asymmetric key exchange the ideal functionality for which is given in Fig. 4. This functionality captures a number of features unique to the paasword case such as the adversary's ability to guess passwords using the `TestPwd` interface. Since we will use $\mathcal{F}_{\mathrm{pwAKE}}$ as a subroutine in our protocols we need to modify the functionality OAuth* by requiring P_k and P_j to respond with a password each in their Authentication Response, and by saving them locally as records for client and server resp. Further, a query `TestPwd` will check the queried password against the recorded password for the server, and change the status of the record to `compromised` or `interrupted`, exactly as in $\mathcal{F}_{\mathrm{pwAKE}}$. One would also require related changes in the key delivery. We refer to this functionality as $\mathcal{F}_{\mathrm{OAuth^*\text{-}aug}}$ This simply reflects the effect of embedding a component protocol into a larger one: the adversay will be able to affect the larger protocol in a similar fashion. However, rather than defining this modified functionality in detail, we focus here on the *simpler case* where the password between P_k and P_j is of full strength, i.e. of entropy k bits, where k is the security parameter. Then, w.l.o.g. we can assume that the adversary in the hybrid model *never* calls the `TestPwd` query. [2]

We now describe the protocol to realize the enhanced version of OAuth*.

Public-Key Generation: On input (OAuth*, *sid*, P_i, P_k, P_j, *initiator*) from the Environment, the party P_i (the *consumer*) calls $\mathcal{F}_{\mathrm{PKE}}$ with input (`KeyGen`, sid_0, P_i). After the adversary replies with algorithm e it records this value.

Session Initiation The Consumer P_i initiates an OAuth session with the three parties by invoking $\mathcal{F}$-auth with input (`Send`, sid_1, P_i, P_k, $\langle \text{initiate}, e, P_j \rangle$).

Initiate Authentication to the Provider: On receiving (`Send`, sid_1, P_i, P_k, $\langle \text{initiate}, e, P_j \rangle$) from $\mathcal{F}$-auth, the end-user (party P_k) uses the password *pwd* which it had obtained earlier from the environment and engages in a password based asymmetric key exchange with P_j (the *Service Provider*) as follows. Party P_k invokes $\mathcal{F}_{\mathrm{pwAKE}}$ with input (`NewSession`, sid_2, pwd,

[2]Similarly, OAuth* needs to include a denial of service call (one each for P_i and P_j). Or one could strengthen $\mathcal{F}_{\mathrm{pwAKE}}$ by removing the DOS call option, and such a functionality can be implemented if we assume that the adversary always selects a public key e which always results in non-colliding ciphertexts with high probability.

client). Party P_k also sends an unauthenticated message to P_j to invoke the functionality with session ID sid_2 and a password which the two parties had secretly shared earlier (e.g. using the public key of public entity P_j; for sake of exposition, we do not go into details of doing this formally). Thus, P_j is expected to invoke the functionality with input (`NewSession`, sid_2, pwd, server).

Forward session-key (verification token): After the adversary calls the functionality with both the `NewKeyServer` and `NewKeyClient` messages, the parties P_j and P_k end up with keys key_j and key_k respectively (or with session aborted). If the client P_k did not have the session aborted, it calls the public key encryption functionality $\mathcal{F}_{\text{PKE}}$ with input (`Enc`, sid_0, $\langle \text{key}_k, P_k, P_i \rangle$). On receiving ciphertext c, P_k sends it unauthenticated to P_i.

Decrypt Session-key(verification token): The Consumer P_i on receiving the ciphertext c' from some party P_a, calls the Decryption part of $\mathcal{F}_{\text{PKE}}$ to obtain a message of the form $\langle \text{key}_0, P_a, P_i \rangle$.

Key Exchange with Provider(Obtaining an Access token):Using $\langle \text{key}_0, P_a, P_i \rangle$ the Consumer engages with Service Provider P_j in a password based asymmetric key exchange, with password key_0, and acting as client (with P_j acting as server and using password key_j). Let key_{ij} be the key returned to P_j, and key_{ji} be the key returned to P_i (with the possibility that the key exchange may have been aborted).

Session key between Consumer and End-User: If the previous key exchange was not aborted for P_i, it now engages in a password based asymmetric key exchange with P_a, with password key_0, and this time acting as server (and P_a acting as client, and if P_a is same as P_k it uses password key_k). At the end of this invocation, let the key output to P_i be called key_{ik}, and the key output to P_a be called key_{ki}.

Output: P_i outputs to the environment (sid, key_{ij}, key_{ik}, key). P_j outputs to the environment (sid, key_{ji}, key). P_k (and/or P_a) outputs to the environment whatever key it had obtained in the previous step (if any).

Theorem 1 *The above protocol securely realizes $\mathcal{F}_{OAuth^*\text{-}aug}$ in the above restricted hybrid model.*

Detailed proofs of the above theorem, along with complete definition of OAuth*-aug will be given in the full version of this paper.

5.3 Optimized Protocol for OAuth* using Password Based Asymmetric Key Exchange

We now show that the protocol for $\mathcal{F}_{\text{OAuth}^*}$ given in the previous sub-section can be optimized by requiring the user P_k to include another random and independent key κ_{ki} in the *encrypted* message (along with a nonce) to P_i in the **Forward session-key** step.. Later, if in **Decryption** in the key exchange with P_i, if P_j does not have its key exchange aborted, it accepts κ_{ki} (or whatever the ciphertext decrypted to) as its shared key with P_a, and sends an $\mathcal{F}$-auth message to P_a that the key from step 3 was accepted. If, on the other hand, the key exchange was aborted in step 4 for P_i, then it sends an authenticated message to P_a to abort as well.

5.4 The real world OAuth wire protocol

The above protocol description of the protocol for OAuth* (augmented by the adversarial interfaces for the username/passsword case) is described at a high level but it has a very strong correspondence with the (corrected) version of the OAuth protocol. We assume that the Consumer and Service Provider have public-keys (and certificates) and that the end-user communicates with these server entities over secure channels(SSL/TLS) and that the end-user authenticates to the Provider using a username and password. With these assumptions we note the following:

- The **Session Initiation** step corresponds to the step in the OAuth Protocol where the Consumer redirects the end-user to the Service Provider's site to begin an OAuth session. Note that technically the OAuth protocol includes a step before this to obtain a Request token. We note here that this step is *not* required for correctness. In fact as specified in the protocol here the session identifier chosen by the Consumer can be used instead. This has also been noted in some of the proposed fixes to the session fixation vulnerability[Fix]. However, as specified in the corrected protocol, the initial exchange can be used to exchange a URL where the Service Provider can redirect the end-user to the Consumer after authorization.
- The **Initiate Authentication to the Provider** corresponds directly to the username-password authentication of the end-user to the Provider.As we have mentioned this step could be replaced by other authentication mechanisms. Technically from the OAuth* functionality this would also be the step at which the Provider binds the restriction r which corresponds to the specific rights that the user wishes to delegate to the Consumer.
- The shared key exchanged with the user, corresponding to the verification token in the proposed correction to OAuth 1.0 is forwarded to the Consumer in the **Forward session-key(verification token)** step.
- The **Key Exchange with the Provider** corresponds to the AccessToken request in the OAuth protocol.

Note that the shared key between the end-user and the Consumer can be simply part of the SSL/TLS session as mentioned in the optiimization section above.While the proof of the actual wire protocol including the vulnerabilties introduced by browser artifacts such as reedirects and ceritificate verification etc.is a little beyond most formal abstractions we note that the protocol we have described for OAuth* is almost identical to the protocol flow in the real world OAuth protocol. Hence our proof of correctness is strong evidence of the formal correctness of the corrections being considered for the OAuth protocol.

5.5 What went wrong with the original protocol: Can we use $\mathcal{F}_{\mathbf{OAuth}}$ instead of $\mathcal{F}_{\mathbf{OAuth}^*}$?

Now that we have set up a considerable amount of formal machinery, a natural question to ask if the models can yield insight into the session fixation vulnerabily in the original OAuth protocol Version 1.0 as described in Section 2. The ideal functionality of the original OAuth protocol is that of entity authentication as described by $\mathcal{F}_{\text{OAuth}}$ in Fig. 14 of Section A.4. The proof that the original OAuth protocol realizes this functionality is deferred to a full version of the paper.

Here, we elaborate on how an attempt to implement the AppStore functionality $\mathcal{F}_{\text{App,Store}}$ using $\mathcal{F}_{\text{OAuth}}$ (instead of the corrected $\mathcal{F}_{\text{OAuth}^*}$) fails. Comparing with the implementaiton described in section 4.3, one notices that now an implementation must build secure sessions without having the benefit of the pairwise keys distributed by $\mathcal{F}_{\text{OAuth}^*}$. The important observation again is that the user *does not* have a public key, and hence cannot use its identity (say P_k) in a global sense. To reiterate, the end-user can have a userid-password with one particular server entity, and possibly many such with different servers, but they cannot be assumed to be associated with each other *i.e.* there is no global association of the end-user's names.

Thus, an attempted implementation may start with an unauthenticated user (say P_a) requesting a consumer P_i to start an $\mathcal{F}_{\text{OAuth}}$ functionality with IDs P_i, P_k and P_j (see Fig 14 in Appendix A.4). This can even be a request under SSL (using functionality $\mathcal{F}_{\text{SSL}}$ in fig 13). Now, if P_a is same as P_k, then the implementation behaves as desired. But, if P_a is different from P_k, there is no way for P_i to know where the request came from (as P_a is unauthenticated), and hence it will invoke $\mathcal{F}_{\text{OAuth}}$ with IDs P_i, P_k and P_j (as requested)[3]. During execution of $\mathcal{F}_{\text{OAuth}}$, P_k is somehow convinced (by P_a) to respond in the affirmative, as a result P_i and P_j are convinced about the three-way authentication, as they are sent "active" messages by $\mathcal{F}_{\text{OAuth}}$. Next, P_j delivers useful information about P_k to P_i (e.g. the stored data $\langle m \rangle$). The consumer then computes f on this data and sends it to the original initiator (using $\mathcal{F}_{\text{SSL}}$), i.e. P_a. We note here that this precisely describes the session fixation vulnerability.

6 Protocol for Password Based Asymmetric Key Exchange

In this section we give a protocol for password based asymmetric key exchange and show that it securely realizes the ideal functionality $\mathcal{F}_{\text{pwAKE}}$. The protocol is given in a hybrid model, which assumes the functionality for public key encryption $\mathcal{F}_{\text{PKE}}$ (but only for the server as decryptor), and the functionality for authenticated message delivery $\mathcal{F}$-auth (again, only for the server as sender). The functionalities $\mathcal{F}_{\text{PKE}}$ and $\mathcal{F}$-auth are defined in Figs 10, and 9 respectively. As we have discussed before, instead of $\mathcal{F}$-auth, we could also use the ideal functionality for public key signatures, but then we will need to have a mechanism for global setup (e.g. global registration) which would make the proofs rather complicated. Thus, for sake of exposition, we just assume the availability of authenticated message delivery (for just the server). We will refer to this as the **asymmetric** ($\mathcal{F}_{\text{PKE}}$, $\mathcal{F}$-auth)-**hybrid model**.

The protocol is described in Figure 5. It shows the server T which has access to the functionality $\mathcal{F}_{\text{PKE}}$ as a decryptor, and the client U which has access to the functionality as a normal encryptor. The instantiation of $\mathcal{F}_{\text{PKE}}$ shown on the left and the right side of the table is the same, with session id sid_1.

The protocol has four main phases, as shown in the table. In the first phase, the server just gets a public key (or algorithm) e from the functionality (which is really provided by the Adversary; for a discussion on this see [Can06]). In the second phase, the server chooses a fresh nonce n, and passes the nonce and the public key to the client (using $\mathcal{F}$-auth). The client U then chooses a random key k (of size given by twice the security parameter). Next, it invokes $\mathcal{F}_{\text{PKE}}$ to encrypt $\langle U, T, n, \text{pwd}, k \rangle$, where pwd is the password which has already been registered with the server T

[3]In practice (i.e. OAuth protocol Version 1.0), P_k is not even listed in this request, and hence $\mathcal{F}_{\text{OAuth}}$ is already a bit more advanced, yet not adequate.

under the userid U. Note that the server only recognizes the userid U as associated with this password pwd, and other than that U has no significance to T. In particular, the user does not have access to $\mathcal{F}$-auth to send messages as U.

The user receives a reply c (ciphertext) from the functionality $\mathcal{F}_{\mathrm{PKE}}$, which it sends to the server along with the nonce n. Now, since this is an unauthenticated message, the adversary can arbitrarily distort c to c'. In the third phase, the server receives a message n, c' (wlog, assume that the adversary does not change the nonce, as the server would immediately reject this message otherwise). The adversary can also distort the identity of the originator of this message to, say, P_a. The server calls the functionality $\mathcal{F}_{\mathrm{PKE}}$ to decrypt the ciphertext message c', and receives a reply m.

In the fourth phase, if m is not of the form $\langle U', T, n, \mathrm{pwd'}, k' \rangle$, the server aborts. If it is of the correct form, it checks if $(U', \mathrm{pwd'})$ is in its password file (i.e. previously registered). If not, it again aborts. Otherwise, k' is split into two equal length strings $k' = \langle k'_1; k'_2 \rangle$, and T sets (or outputs to the environment) the key as k'_2. It also sends an authenticated message to P_a, which includes n, and the first part of k', i.e. k'_1. In the case it aborts, it selects a new random k'_1, and sends that instead.

Now, the adversary may decide to deliver the message to U or not. However, if the adversary does deliver the message to U, then U is assured that it came from T (because of $\mathcal{F}$-auth). Hence accordingly, it will either abort or outputs the key k_2, where again k is split into two pieces $k = \langle k_1; k_2 \rangle$. The party U has an easy way to determine if T wants it to accept: it just checks if k_1 equals the authenticated message k'_1.

Having described the protocol, which we will refer to as protocol π, we now show that it securely realizes the ideal functionality for password based asymmetric key exchange.

Theorem 2 *The protocol π in Figure 5 securely realizes $\mathcal{F}_{pw\mathrm{AKE}}$ in the asymmetric ($\mathcal{F}_{\mathrm{PKE}}$, $\mathcal{F}$-auth)-hybrid model.*

Proof: See Appendix A.1.

Note that, the protocol π does not assure forward security with respect to server corruption (nor has $\mathcal{F}_{\mathrm{pwAKE}}$ been defined for forward security). For instance, if the server is corrupted after it has finished the protocol (and possibly used and erased the key k'_2), the adversary can still get k'_2 from using the $\mathcal{F}_{\mathrm{PKE}}$ decryptor on c' $(= c)$. However, as in [HK98], a Diffie-Hellman key exchange integrated in π assures forward security (in the indifferentiability security model). Moreover, as π has the ACK property [CK01], it also realizes a forward secure ideal functionality version of $\mathcal{F}_{\mathrm{pwAKE}}$, without resorting to non-information oracles [CK01]. Detailed definitions and proofs of realization will be given in the full version of this paper.

7 Conclusion

In this paper we have use the Universally Composable framework to analyze the security of web security protocols for delegation. We have defined ideal functionalities for delegated computation and secure authenticated delegation which capture the ideas of server-side mash-ups and the hand-shake portion of the corrected OAuth protocol respectively. We have shown that a protocol which is similar to the corrected OAuth protocol realizes the ideal functionality for secure authenticated delegation. This gives us assurance that the corrections being considered to the OAuth protocol are

Protocol for Password based Asymmetric (with server PK) Key Exchange

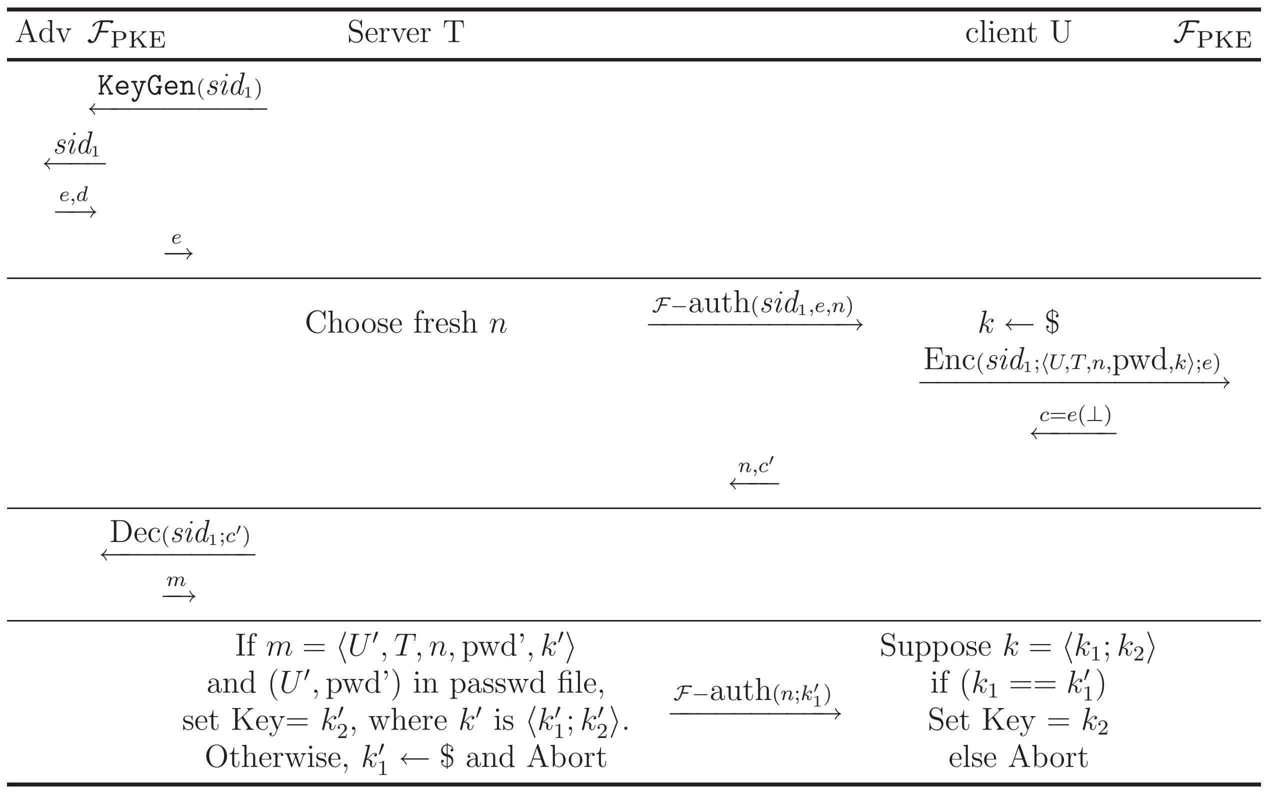
Adv $\mathcal{F}_{\text{PKE}}$
Server T
client U
$\mathcal{F}_{\text{PKE}}$
$\xleftarrow{\texttt{KeyGen}(sid_1)}$
$\xleftarrow{sid_1}$
$\xrightarrow{e,d}$
$\xrightarrow{e}$
Choose fresh n
$\xrightarrow{\mathcal{F}-\text{auth}(sid_1,e,n)}$
$k \leftarrow \$$
$\xrightarrow{\text{Enc}(sid_1;\langle U,T,n,\text{pwd},k\rangle;e)}$
$\xleftarrow{c=e(\perp)}$
$\xleftarrow{n,c'}$
$\xleftarrow{\text{Dec}(sid_1;c')}$
$\xrightarrow{m}$
If $m = \langle U', T, n, \text{pwd'}, k' \rangle$
and $(U', \text{pwd'})$ in passwd file,
set Key$= k'_2$, where k' is $\langle k'_1; k'_2 \rangle$.
Otherwise, $k'_1 \leftarrow \$$ and Abort
$\xrightarrow{\mathcal{F}-\text{auth}(n;k'_1)}$
Suppose $k = \langle k_1; k_2 \rangle$
if $(k_1 == k'_1)$
Set Key $= k_2$
else Abort

Figure 5:

indeed correct. To the best of our knowledge ours is one of the first work to consider the security of web security protocols for delegation.

References

[Aut] Authentication for web applications. Specifications available online at `http://code.google.com/apis/accounts/docs/AuthForWebApps.html`.

[BBA] Browser based authentication. Specifications available online at http://developer.yahoo.com/auth/.

[BR95] M. Bellare and P. Rogaway. Provably securesession key distribution - the three party case. In *Symposium on Theory of Computation*, 1995.

[Can06] Ran Canetti. Security and composition of cryptographic protocols: A tutorial. *SIGACT News*, 37(3 & 4), 2006.

[CHK$^+$05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 404--421. Springer, 2005.

[CK01] R. Canetti and H. Krawczyk. Analysis of key exchange protocols and their use for building secure channels. In *Eurocrypt*, number LNCS 2045, 2001.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars Knudsen, editor, *Advances in Cryptology -- EUROCRYPT '2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 337--351, Amsterdam, The Netherlands, 2002. Springer-Verlag, Berlin Germany. Extended version at http://eprint.iacr.org/2002/059.

[Con] OAuth Consumer Request 1.0 Draft 1. Specifications available online at http://oauth.googlecode.com/svn/spec/ext/consumer_request/1.0/drafts/1/spec.html.

[Fix] OAuth Session Fixation Advisory. http://wiki.oauth.net/OAuth-Session-Fixation-Advisory.

[HK98] Shai Halevi and Hugo Krawczyk. Public-key cryptography and password protocols. In *ACM Conference on Computer and Communications Security*, pages 122--131, 1998.

[OAu07] Oauth core 1.0. Technical Specifications available online at http://oauth.net/core/1.0/, Dec 2007.

[OAu09] Technical advisory available online at http://oauth.net/advisories/2009-1, Apr. 2009.

[Ope] AOL open authentication api (openauth). Specifications available online at http://dev.aol.com/api/openauth.

A Appendix

A.1 Proof of Theorem 2

In this section we sketch a proof of correctness of the protocol for password based asymmetric key exchange.

As is usual, $\text{IDEAL}_{\mathcal{F},S,\mathcal{Z}}(k,z,r)$ will denote the random variable corresponding to the output of the environment $\mathcal{Z}$, when interacting with dummy parties and adversary S involved with the ideal functionality $\mathcal{F}$, and where k is the security parameter, z is the initial input of z, and r is the randomness of all the parties concerned. Similarly, $\text{HYB}^{\mathcal{F}}_{\pi,A,\mathcal{Z}}(k,z,r)$ will denote the random variable corresponding to the output of $\mathcal{Z}$, when interacting with parties involved in protocol π (with adversary A) in the (asymmetric or otherwise) hybrid model with functionalities $\mathcal{F}$.

Theorem 2 The protocol π in Figure 5 securely realizes $\mathcal{F}_{\text{pwAKE}}$ in the asymmetric ($\mathcal{F}_{\text{PKE}}$, $\mathcal{F}$-auth)-hybrid model.

Proof: We will show that for every PPT adversary A, there exists a PPT adversary S, such that the ensembles $\text{IDEAL}_{\mathcal{F}_{\text{pwAKE}},S,\mathcal{Z}}$ and $\text{HYB}^{\mathcal{F}_{\text{PKE}},\mathcal{F}\text{-auth}}_{\pi,A,\mathcal{Z}}$ are computationally indistinguishable. In the ideal world protocol, the parties U and T run as dummies and just pass back and forth the messages between $\mathcal{Z}$ and $\mathcal{F}_{\text{pwAKE}}$. In the real (or hybrid) world the parties U and T are running the protocol π (or playing their part for it). We will show that for every adversary A, there is an adversary S which uses A as a black box and emulates its move. But, to do so it will need to simulate the real parties U and T, as well as the functionalities $\mathcal{F}_{\text{PKE}}$ and $\mathcal{F}$-auth, for A. we will show that S is able to do so, and hence as far as the environment $\mathcal{Z}$ is concerned the hybrid world and the ideal world are indistinguishable.

S simulates $\mathcal{F}_{\text{PKE}}$ as follows. Recall that S is also simulating the parties T and U. Thus, whenever T invokes $\mathcal{F}_{\text{PKE}}$ with KeyGen, it passes the session id to A, which then replies with d, e. S retains d and e, and also passes e to its simulation of T. Whenever, any party (including A) calls $\mathcal{F}_{\text{PKE}}$ with an `Enc` query (m, e'), it simulates $\mathcal{F}_{\text{PKE}}$ and records (m, c) if required (note that there can be only polynomially many such records to maintain). Therefore, it is also able to simulate `Dec` queries.

S simulates $\mathcal{F}$-auth by passing the incoming message to A, and if A responds to allow delivery, then it passes the message to its simulated copy of the recipient.

Next we see how S simulates the real world parties and their interactions with the environment $\mathcal{Z}$. In both the real and the hybrid world, $\mathcal{Z}$ prompts the parties with initial input (`New Session`, *sid*, (U',pw), role), where role is client for U, and server for T, and *pw* is the password. The values U' and pw can be different for both U and T, and U' may have no relation to U. For clarity let U' received by U be called U_u, and the password received by U be called pw_u. Similarly, the values received by T are subscripted by t, i.e. U_t and pw_t. The session ID *sid* is same for both parties.

In the ideal world, this input is forwarded directly to $\mathcal{F}_{\text{pwAKE}}$, which in turn forwards it to S, excluding the password, and including the identity P of the sender (i.e. U or T). On receiving this input (and when the role is server), S simulates T in real world by sending (`KeyGen`, sid_1) to its simulated copy of $\mathcal{F}_{\text{PKE}}$. When S receives the input with role set as client, and if the server input has already been received, and if the *sid* is same, S picks a fresh nonce n, and informs A about sending authenticated message from T to U. If A decides to write (deliver) this message to U, then so does S in its simulated copy of U. If A decides to deliver it to someone else (say, e.g. $U'' \neq U$) then w.l.o.g. the message is considered as good as lost.

If A does write the message to U, S picks a random κ (of size given by the security parameter), and since it is simulating $\mathcal{F}_{\text{PKE}}$ it generates $c = e(\perp)$, and appends $(\langle *, T, n, *, \kappa\rangle), c)$ to its list L of legitimate plaintext, ciphertext pairs, where * is a placeholder. In the hybrid world, U would have made a call for plaintext $(\langle U_u, T, n, \text{pw}_u, \kappa\rangle)$. Recall that the adversary A may also have made calls to this encryption functionality, and S maintains those in L as well.

The simulator adversary S now passes n, c to the adversary A for delivery to T. If the adversary writes n, c' to T, then so does S to its simulated copy of T. Next, using its list L, S decrypts c' as follows. If c' is in list L, associated with some (unique) plaintext m, then, it checks for m to have the correct format. Otherwise, if c' was not in the list, it sets $m = d(c')$. Thus, there are three possibilities: (a) there was a duplicate entry associated with c', and hence error was reported, (b) c' is same as c (c) c' was decrypted either using L or using d.

In case (a), S makes a denial of service call to $\mathcal{F}_{\text{pwAKE}}$. Since S was simulating $\mathcal{F}_{\text{PKE}}$ perfectly by setting ciphertexts to $e(\perp)$ every-time, the probability of getting a collision is identical in the two worlds. Next, (on a null reply from the functionality), S issues a `NewKeyServer` call to $\mathcal{F}$. Next, it gives A a randomly chosen message (simulating $\mathcal{F}$-auth), as in the hybrid world also k'_1 is chosen randomly. After A directs to deliver the message to U, S just issues a `NewKeyClient` call to $\mathcal{F}$ with an issue directive.

In case (b), given that there was no collision, S can assume that decryption in the hybrid world would yield $\langle U_u, T, n, \text{pw}_u, k'\rangle$, where k' is some value U would have chosen randomly. Thus, iff $U_t = U_u$ *and* $\text{pw}_u = \text{pw}_t$ (for simplicity, we assume that there is only one entry in the password file), the hybrid world would yield an accepting condition for the server T. Thus, S issues a `NewKeyServer` call to $\mathcal{F}_{\text{pwAKE}}$ with arbitrary sk. Next, it gives adversary A a randomly chosen message, even though S does not know if it has to simulate an accept or an abort message to U. This suffices to simulate the hybrid world, as in the hybrid world also k' is chosen randomly by U and remains independent of any information A may have (recall $c = e(\perp)$ is independent of k'). Finally, when A directs to deliver the message to U, S just issues a `NewKeyClient` call with an issue directive.

In case (c), if c' was decrypted using L, then S already has all the messages which A requested to be encrypted, including the possible passwords in those messages. Hence, corresponding to this c', let the message m be $\langle U', T, n, \text{pwd}, \gamma\rangle$. S then issues a `TestPwd` query to $\mathcal{F}_{\text{pwAKE}}$ with password set to (U', pwd). Thus, if in the hybrid world, A had managed (by sending c' instead of c) to produce a correct password (i.e. matching the one in the password file of server), $\mathcal{F}_{\text{pwAKE}}$ would also have replied in the affirmative (and marked the server record compromised). Otherwise, it would have replied in the negative and marked the server record `interrupted`. Regardless, S now issues a `NewServerKey` call to $\mathcal{F}_{\text{pwAKE}}$ with sk set to γ. Moreover, in this case, S knows whether to simulate an accept or an abort message to U (since a server record marked corrupted implies an accept message, while interrupted implies an abort message). Further S also knows γ. Thus, it can supply A with a perfectly simulated message (as part of simulating $\mathcal{F}$). Finally, after A directs to deliver the message to U, S just issues a `NewKeyClient` call with an issue directive.

Similarly, if c' was decrypted using d, then let $m = d(c')$, which would be same in both the worlds. If m is not of the correct format, then S behaves as in case (a), i.e. with a denial of service call, followed by the "new key" calls. Otherwise, if m is of the correct format, S behaves as in the previous paragraph.

It is easy to see that in all the above cases, the emulation of A by S is statistically perfect.

As far as corruption of parties is concerned, first suppose that the environment directs S to corrupt U, in which case it first informs $\mathcal{F}$ to get secrets associated with U (namely, the password (U_u, pw_u)), and then S instructs A to corrupt U by handing over its simulated local state of U, and from then on A runs U. After this, if A calls $\mathcal{F}_{\mathrm{PKE}}$ to encrypt a message $\langle U_u, T, n, \mathrm{pw}_u, \delta\rangle$, and for which S returns a ciphertext c, and which A instructs to be sent to T, then S calls $\mathcal{F}$ with (`NewKeyServer`, *sid*, T, δ). On the other hand, if A instructs to send a c' different from such a c (i.e. one which does not match the password, or is an illegitimately obtained ciphertext), then S behaves as before (i.e. the non-corruption case). It is easy to see that in both cases the emulation is perfect.

If the environment directs S to corrupt T, in which case S informs $\mathcal{F}$ that it is corrupting T, which hands it over the secret (U_t, pw_t). Next, it instructs A to corrupt T, and from then on A runs T. Now, the adversary A cannot influence the client U much, except it may try to send wrong messages authenticated on behalf of T. If it sends a wrong public key e', it does not give any additional advantage to A. It can send the wrong k_1' response in the last "authenticated" message, but this is easily simulated by S by making the `NewKeyClient` call to $\mathcal{F}$ with a similar directive (i.e. if sent k_1' is different from k_1' handed over by U to A, then issue abort directive). $\square$

A.2 Realizing Secure Channels

Let MAC be a secure Message Authentication algorithm, and let $\mathrm{E} = (\mathrm{ENC}, \mathrm{DEC})$ be a symmetric encryption scheme that is semantically secure against chosen plaintext attacks. Consider the following "generic secure channels" protocol, $\mathrm{GSC}_{\mathrm{MAC,E}}$, that operates in the $\mathcal{F}_{\mathrm{KE}}$-hybrid model.

1. When activated with input (`Establish-session`, $sid, P_j, role$), P_i sends an (`Establish-session`, $sid, P_j, role$) request to $\mathcal{F}_{\mathrm{KE}}$. When it gets the key κ from $\mathcal{F}_{\mathrm{KE}}$, it partitions the key to two segments, treated as two keys κ_e and κ_a. (κ_e and κ_a will be used for encryption and authentication, respectively.)

2. When activated with input (`Send`, sid, m), P_i computes $c = \mathrm{E}_{\kappa_e}(m)$, $a = \mathrm{MAC}_{\kappa_a}(c, l)$, and sends (sid, c, l, a) to P_j. Here l is a counter that is incremented for each message sent.

3. When receiving a message (sid, c, l, a), P_i first verifies that a is a valid tag for (c, l) with respect to key κ_a and that no message with counter value l was previously received in this SC-session. If so, then it locally outputs $\mathrm{DEC}_{\kappa_e}(c)$. Otherwise, it outputs nothing (or an error message).

4. When activated with input (`Expire-session`, sid) the SC-session within P_i erases the session state (including all keys and local randomness), and returns.

Ideally, one would like to claim that protocol $\mathrm{GSC}_{\mathrm{E,MAC}}$ securely realizes $\mathcal{F}_{\mathrm{SC}}$ in the $\mathcal{F}_{\mathrm{KE}}$-hybrid model, as long as E and MAC are secure. Unfortunately, such a strong claim does not hold (see [CK01]). Instead, [CK01] prove that a restricted version of protocol GSC securely realizes $\mathcal{F}_{\mathrm{SC}}$. They also demonstrate (using non-information oracles) how to relax functionality $\mathcal{F}_{\mathrm{SC}}$ to allow proving security of the above general form of protocol GSC.

the restricted version of the protocol GSC (mentioned above) is obtained by replacing the generic use of a semantically secure encryption scheme with the following more specific encryption mechanism. This mechanism puts a bound t on the total number of bits to be communicated by each party in the session. Initially, each party uses a pseudorandom number generator G to expand the encryption key κ_e to two pads of length t each. Next, κ_e is *erased* by both parties. The first pad is used to encrypt the messages from P_i to P_j, and the second pad is used to encrypt messages from P_j to P_i. Encryption is done via one-time-pad in the natural way (each message is encrypted by xoring it with a new portion of the pad. The ciphertext then contains the index of the used part of the pad).

Claim 3 *[CK01] Let* MAC *be a secure Message Authentication Code function and G be a pseudo-random generator. Then, protocol* $\textsc{gsc}'_{\text{MAC},G}$ *securely realizes $\mathcal{F}_{\text{SC}}$ in the $\mathcal{F}_{\text{KE}}$-hybrid model.*

A.3 Functionalities

This section describes protocol execution models as described in Figs 6,7,8, and well known UC functionailities in Figs 9, 10, 11, and 12. We also give a new definition of the SSL functionality, which reflects a unauthenticated secure session between two parties with only one of them having a known public key (see fig 13).

A.4 Ideal Functionality for the Original OAuth Protocol

This section describes the ideal functionality of the original uncorrected OAuth protocol.The ideal functionality is essentially a three way authentication with the parties being identified as the *user* (P_k), the *consumer* (P_i), and the *service provider* (P_j). The ideal functionality for this is described in the Appendix A.4. The protocol is initiated by the consumer who also specifies a *role* to which authority is delegated to it. In the most general terms the role is given by a PPT Turing Machine describing a function r which acts as a filter on information to be provided by the user and/or the service provider. The functionality is defined in Fig 14.

Protocol execution in the real world model

Participants: Parties $P_1, ..., P_n$ running protocol π with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ on input z. When a party wants to deliver a message to another party, it instructs the Adversary to do so by writing the message (and the recepient's identity) on Adversary's incoming message communication tape. All participants have the security parameter k.

1. While $\mathcal{Z}$ has not halted do:
   - (a) $\mathcal{Z}$ is activated (i.e., its activation tape is set to 1). In addition to its own readable tapes, $\mathcal{Z}$ has read access to the output tapes of all the parties and of $\mathcal{A}$. The activation ends when $\mathcal{Z}$ enters either the halting state or the waiting state. If $\mathcal{Z}$ enters the waiting state then it is assumed to have written some arbitrary value on the input tape of exactly one entity (either $\mathcal{A}$ or of one party out of $P_1, ..., P_n$). This entity is activated next.
   - (b) Once $\mathcal{A}$ is activated, it proceeds according to its program performing one of the following operations:
     - i. Deliver a (possibly fake) message m to party P_i. Delivering m means writing m on the incoming message tape of P_i, together with the identity of some party P_j as the sender of this message.
     - ii. Corrupt a party P_i. Upon corruption $\mathcal{A}$ learns the current internal state of P_i, and $\mathcal{Z}$ learns that P_i was corrupted. (Say, the state of P_i is written on $\mathcal{A}$'s input tape, and $\mathcal{Z}$ actually drove $\mathcal{A}$ to corrupt P_i.) Also, from this point on, P_i may no longer be activated.

     In addition, at any time during its activation $\mathcal{A}$ may write any information of its choice to its output tape.
     If $\mathcal{A}$ delivered a message to some party in an activation, then this party is activated once $\mathcal{A}$ enters the waiting state. Otherwise, $\mathcal{Z}$ is activated as in Step 1a.
   - (c) Once an uncorrupted party P_i is activated (either due to a new incoming message, delivered by $\mathcal{A}$, or due to a new input, generated by $\mathcal{Z}$), it proceeds according to its program and possibly writes new information on its output tape and Adversary's incoming message tape. Once P_i enters the waiting or the halt states, $\mathcal{Z}$ is activated as in Step 1a.
2. The output of the execution is the first bit of the output tape of $\mathcal{Z}$.

Figure 6: The order of events in a protocol execution in the real world model

The ideal process

Participants: Environment $\mathcal{Z}$ and ideal-process adversary $\mathcal{S}$, interacting with ideal functionality $\mathcal{F}$ and dummy parties $\tilde{P}_1, ..., \tilde{P}_n$. All participants have the security parameter k; $\mathcal{Z}$ also has input z.

- The ideal functionality acts as a joint sub-routine to all the dummy parties. Hence, all its communications to the dummy parties is trusted and secure. The ideal functionality also communicates with the adversary, and to model delayed delivery to the parties, the functionality takes delivery instructions from the adversary. The functionality may report to the adversary some crucial state changes, and on corruption of some party by the adversary (which is communicated to the functionality), the functionality may disclose more information to the adversary.
- The rest of the process is same as the real process, and the only role of the dummy parties is to copy its input from the environment to the sub-routine tape of the functionality, and similarly to copy the output from the functionality to the input tape of the encironment. The dummy parties never write on the incoming message communication tape of the Adversary.

Figure 7: The ideal process for a given ideal functionality, $\mathcal{F}$.

Protocol execution in the $\mathcal{F}$-hybrid model

Participants: Parties $P_1, ..., P_n$ running protocol π with multiple copies of an ideal functionality $\mathcal{F}$, with adversary $\mathcal{H}$, and with environment $\mathcal{Z}$ on input z. The hybrid process is similar to the ideal process, with $\mathcal{F}$ being a joint sub-routine to the parties, except that now the parties are not just dummies. They may communicate with the adversary, and have some other internal computations just as in the real process.

Figure 8: The hybrid model

Functionality $\mathcal{F}$-auth

1. Upon receiving an input (`Send`, *sid*, P_i, P_j, m) from party P_i, send the input to the adversary.
2. Upon receiving a "deliver" response from the adversary, write (*sid*, P_i, P_j, m) on the subroutine tape of P_j.
3. Upon receiving a message (`Corrupt-Sender`, *sid*, m') from the adversary *before* the "deliver" response, write (*sid*, P_i, P_j, m') on the subroutine tape of P_j *and* halt.

Figure 9: The message Authentication functionality $\mathcal{F}$-auth

Functionality $\mathcal{F}_{\mathbf{PKE}}$

For a given domain M of plaintexts, $\mathcal{F}_{\text{PKE}}$ proceeds as follows. Let $\perp \in M$ be a fixed message.

Key Generation: Upon receiving an input (`KeyGen`, *sid*, D) from some party D, send the input to the adversary. Upon receiving (`Algorithms`, *sid*, e, d) from the adversary, where e and d are descriptions of probabilistic interactive TMs, write (`Encryption Algorithm`, *sid*, e) on subroutine tape of D.

Encryption: Upon receiving input (`Enc`, *sid*, m, e') from any party E, do: If $m \notin M$ then output an error message to E. Else, if $e' \neq e$, or the decryptor D is corrupted, then let $c = e'(m)$. Else, let $c = e'(\perp)$ *and* record (m, c). Write (`Ciphertext`, *sid*, c) on the subroutine tape of E.

Decryption: Upon receiving input (`Dec`, *sid*, c) from D (and D only), do: If there is a recorded entry (m, c) for some unique m then write (`Plaintext`, m) on subroutine tape of D. Else, write (`Plaintext`, $d(c)$). If there are more than one recorded m for the given c, write an error message.

Figure 10: The public-key encryption functionality with "ideal non-malleability"

Functionality $\mathcal{F}_{\text{KE}}$

$\mathcal{F}_{\text{KE}}$ proceeds as follows, running on security parameter k, with parties $P_1, ..., P_n$ and an adversary $\mathcal{S}$.

1. Upon receiving a value (`Establish-session`$, sid, P_i, P_j, role$) from some party P_i, record the tuple $(sid, P_i, P_j, role)$ and send this tuple to the adversary. In addition, if there already is a recorded tuple $(sid, P_j, P_i, role')$ (either with $role' \neq role$ or $role' = role$) then proceed as follows:
   (a) If both P_i and P_j are uncorrupted then choose $\kappa \overset{\text{R}}{\leftarrow} \{0,1\}^k$, send (`key`$, sid, \kappa$) to P_i and P_j, send (`key`$, sid, P_i, P_j$) to the adversary, and halt.
   (b) If either P_i or P_j is corrupted, then send a message (`Choose-value`$, sid, P_i, P_j$) to the adversary; receive a value κ from the adversary, send (`key`$, sid, \kappa$) to P_i and P_j, and halt.
2. Upon corruption of either P_i or P_j, proceed as follows. If the session key is not yet sent (i.e., it was not yet written on the outgoing communication tape), then provide $\mathcal{S}$ with the session key. Otherwise provide no information to $\mathcal{S}$.

Figure 11: The (Strong) Key Exchange functionality

Functionality $\mathcal{F}_{\rm SC}$

$\mathcal{F}_{\rm SC}$ proceeds as follows, running with parties $P_1, ..., P_n$ and an adversary $\mathcal{S}$.

1. Upon receiving a value $(\texttt{Establish-session}, sid, P_j, \text{initiator})$ from some party, P_i, send (sid, P_i, P_j) to the adversary, and wait to receive a value $(\texttt{Establish-session}, sid, P_i, \text{responder})$ from P_j. Once this value is received, set a boolean variable `active`. Say that P_i and P_j are the partners of this session.
2. Upon receiving a value $(\texttt{Send}, sid, m)$ from a partner P_e, $e \in \{i, j\}$, and if `active` is set, send $(\texttt{Received}, sid, m)$ to the other partner and $(sid, P_i, |m|)$ to the adversary.
3. Upon receiving a value $(\texttt{Expire-session}, sid)$ from either partner, un-set the variable `active`.

Figure 12: The Secure Channels functionality, $\mathcal{F}_{\rm SC}$

Functionality $\mathcal{F}_{\rm SSL}$

$\mathcal{F}_{\rm SC}$ proceeds as follows, running with parties $P_1, ..., P_n$ and an adversary $\mathcal{S}$.

1. Upon receiving a value $(\texttt{Establish-session}, sid, P_j, \text{initiator})$ from some party, P_i, send (sid, P_i, P_j) to the adversary and a delayed message sid to P_j, and wait to receive a value $(\texttt{Establish-session}, sid, \text{responder})$ from P_j. Once this value is received, set a boolean variable `active`. Say that P_i and P_j are the partners of this session.
2. Upon receiving a value $(\texttt{Send}, sid, m)$ from a partner P_e, $e \in \{i, j\}$, and if `active` is set, send $(\texttt{Received}, sid, m)$ to the other partner and $(sid, P_i, |m|)$ to the adversary.
3. Upon receiving a value $(\texttt{Expire-session}, sid)$ from either partner, un-set the variable `active`.

Figure 13: The SSL functionality, $\mathcal{F}_{\rm SSL}$

Functionality $\mathcal{F}_{\textbf{OAuth}}$

Participants: The ideal functionality $\mathcal{F}_{\text{OAuth}}$ which acts as a joint subroutine to parties P_i (the *consumer*), P_k (the *user*), and P_j (the *service provider*).

Initiate: On receiving a subroutine input message (*sid*,P_i, P_k, P_j, r,*initiator*) from P_i, the functionality $\mathcal{F}$ forwards the message to adversary $\mathcal{A}$. Here, r is a function given by a PPT TM (which is meant to be a role or filter). The adversary then replies twice, once each for P_k and P_j, whereupon $\mathcal{F}$ also writes the message to P_k and P_j resp. $\mathcal{F}$ records locally the state of the protocol.

Authentication Response: On receiving a subroutine input message from P_l ($l \in \{k, j\}$) of the form (*sid*, P_i, P_k, P_j, *response*), the functionality forwards the message to $\mathcal{A}$. $\mathcal{F}$ records the state change locally, and if both P_k and P_j have responded, it sets the local state to *active*.

Activation: If the adversary responds to $\mathcal{F}$ with a "deliver" message for P_s ($s \in \{i, k, j\}$), and the local state is *active*, then $\mathcal{F}$ writes the message (*sid*, P_i, P_k, P_j, *active*) to P_s.

Corruption: On corruption of any one of the parties P_s ($s \in \{i, k, j\}$), there is nothing additional to reveal to the adversary.

Figure 14: A functionality for delegation