Universally Composable Security Analysis of OAuth v2.0

Suresh Chari* Charanjit Jutla* Arnab Roy*

Abstract

This paper defines an ideal functionality for delegation of web access to a third-party where the authentication mechanism is password-based. We give a universally-composable (UC) realization of this ideal functionality assuming the availability of an SSL-like ideal functionality. We also show that this implementation can be further refined to give a browser based implementation whenever the browser supports https redirection. This implementation matches the 'Authorization Code' mode of the OAuth Version 2.0 Internet draft proposal, with the additional requirement that the third-party along with the Authorization Server must support an SSL-like functionality.

From the universally-composable perspective, our ideal functionality definition is novel in the respect that it does not require the three parties to decide on a session identifier in advance, which is usually assumed in a UC setting. This allows us to realize the ideal functionality without any wrapper code, and thus exactly matching the desired protocol in the OAuth standard.

1 Introduction

We analyze the Web delegation protocol OAuth Version 2.0 in the Universal Composability (UC) Security framework [1]. In this framework one defines an ideal third party functionality which is handed inputs by all protocol participants, and the functionality then computes the requisite function, and returns portions of the computed function to each of the protocol participants as desired by the protocol. Security of a real-world implementation of the protocol is then proven by a simulation argument, which essentially shows that if an Adversary can gain any information in the real-world implementation, it can also obtain the same information in the ideal third party based protocol.

In the current paper, we only analyze the Authorization Code mode of OAuth 2.0. Analysis of the Implicit Grant mode of OAuth 2.0 will be given in Part II of this paper.

1.1 Index

Here is s summary of what is covered in this paper.

Section 2 gives an introduction to Ideal Functionalities.

Section 3 gives an introduction to Simulation Based Security Definitions.

*IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

Section 5 defines the OAuth 2.0 Authorization Code Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$. It assumes that the Client (also called Consumer) and Service Provider have globally known names, whereas the User (or user agent) only has local identity based on a userid, password account with these global entities.

Section 6 and Fig 4 give a real-world realization of the Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$ using SSL. This implementation is general purpose and does not restrict the User code to be a user agent (i.e. an http browser). This section rigorously proves that this is a universally composable (UC) secure implementation. Later, in Fig 5 we give a refinement of this implementation where the User code is restricted to being a User Agent capable of handling `https` redirections. Further still, in Figs 6 and 7 we give refinements where some of the initial flows are based on hyper-links being sent by authenticated or unauthenticated emails respectively by the Client (Consumer) to the User.

1.2 Security Analysis Synopsis

Here are some the salient findings related to security of OAuth 2.0 Authorization Code mode.

1. The protocol in Fig 5, which we prove to be a secure realization of our Ideal Functionality for OAuth, is more or less the same as defined in the IETF internet draft OAuth Version 2.0 Authorization Code mode [2], except for a few fine points noted below.

2. Web-Servers which serve arbitrary Users on the Internet must have public keys for authenticating themselves. Thus, both the Service Provider and the Consumer (Client) must have public keys. Further, for simplicity, assuming that the public key of the Service Provider is attested by a global Certificate Authority (CA), the Public Key of the Client should at least be pre-registered with this globally certified Service Provider (if Client itself does not already have a globally certified public key). If the Client's public key is not globally certified, then its public key must be distributed securely to applications running on User's machine and capable of verifying signatures issued with this public key (in essence, implementing SSL).

3. Given that the Service Provider is required to have a globally certified public key, one can assume that it can handle SSL requests, in particular https requests. Since the Client also has a public key with the public key securely delivered to the User, the User Machine's application must be able to run a protocol with this Client which essentially implements SSL.

4. It is advisable that the login form presented by the Service Provider to the User contain both the Globally certified name of the Service Provider (e.g. Charles Schwab) **and** the Globally known name of the Client (e.g. turboTax) as to whom temporary access is being granted. While this is not that important in cases where the hyper-link to the Service Provider's login page comes from the Client via an SSL connection, but it can be important where the hyper-link to the Service Provider's login page comes via un-authenticated email (see fig 7).

5. In the flow where the Client (Consumer) presents the Authorization Code to the Server to get back an Access Token, the Service Provider should establish that this Client is the same as the one to whose redirect URI the AccessToken was sent. The correct flows are as follows (or any equivalent flows). During Authorization Grant, when the Server is presented with

a Client's redirect URI, a public key of the Client, and a certificate on this public key, the Server must check that this redirect URI belongs to the same entity, by either checking for this information in the certificate or by checking that this fact has been pre-registered. The session id for this session is saved by the Server along with this public key of the Client. Later, when the Client comes back to present the Authorization Code to get the AccessToken, the Server must check that this request is coming from a Client with the same public key. This can be done e.g. by using SSL/TLS two-side certificate checking[1].

We point out that this check is only required if the Client and User have an authentication mechanism (e.g. a userid, password) of their own, which is most likely the case. If the Client never authenticates the User, then because of other security issues, the fine points of the previous paragraph are moot.

6. If the Client authenticates the User, say by a userid, password account (which could be same or different from the account the User has with the Provider), then every fresh SSL session between the User and the Client must re-do this authentication. Thus, in the protocol we give (Figs 4 and 5), there are two SSL sessions between the User and the Client, and the second can be replaced by just using the first if the first is still active. If however, the first session has expired (and cannot even be refreshed) and a new SSL session must be established, then the authentication of the User by the Client must be re-done in this new SSL session.

2 Introduction to Ideal Functionalities

Cryptographic protocols are inherently multi-party protocols where the parties jointly compute some function. While the issue of liveness of protocols falls in the realm of distributed computing, we are interested here in issues related to privacy (secrecy) of the parties, as well as the related issue of authentication.

These multi-party privacy constraints of a protocol are best **defined** by hypothetically employing an *ideal trusted third party*. In the simplest form of this definitional paradigm, all parties hand their respective inputs to the ideal trusted third party, which then computes the function on these inputs as desired by the protocol being defined, and then hands portions of the output to the individual parties, the portions restricted to what is desired by the protocol.

Note that in the above we assumed that the parties have a secure and persistent tunnel to the ideal third party, but this being just a definition this assumption is not a concern. In addition, if the protocol requires that a party in this multi-party protocol be a particular globally known entity, then we *further assume* that one of these secure tunnels is known to the ideal third party to be connected to that globally known entity.[2] For parties that are not globally known, the ideal third party just assumes a persistent connection to some party (e.g. this will guarantee that it gives output to the same party that supplied some input using that tunnel). These issues will be brought up again in the examples below.

In a more general setting, this definitional paradigm also allows the ideal third party to leak certain information to an **Adversary** (without loss of generality, we always assume that there is *only* one monolithic adversary). This maybe a necessary part of the definition as otherwise

[1] This essentially implements the Secure Channel Ideal Functionality (Fig 1).

[2] In the real world this can for example be implemented using a public key infrastructure (PKI).

there may not be any implementation possible in the real world (i.e. a distributed implementation without the ideal third party). In a further generalization, the ideal third party may also take directives and/or inputs from the Adversary.

Ideal Functionality $\mathcal{F}_{\text{SC}}$

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

1. Upon receiving a value (`Establish-session`, sid, P_j, initiator) from some party, P_i, send (sid, `Establish-Session`, P_i, P_j) to the adversary, and wait to receive a value (`Establish-session,` sid, P_i, 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 (`Send`, sid, m) from a partner P_e, $e \in \{i, j\}$, and if `active` is set, send (`Received,`sid, m) to the other partner and ($sid, P_i, |m|$) to the adversary.
3. Upon receiving a value (`Expire-session`, sid) from either partner, un-set the variable `active`.

Figure 1: The Secure Channels functionality, $\mathcal{F}_{\text{SC}}$

Ideal Functionality $\mathcal{F}_{\text{SSL}}$

The functionality $\mathcal{F}_{\text{SSL}}$ is between a party P_j (the Server) and an unnamed party designated as the client. This models the asymmetric situation when the server has a public idenity but a client who connects to it has no global identity.

1. Upon receiving a value (`Establish-session`, sid, P_j, `initiator`) from some party, say P_i, the functionality first sends the message (sid, `Establish-Session, initiator`, P_i, P_j) to the adversary. If the adversary responds positively the functionality records P_i as the client in this session and sends the message (sid, `Establish`$-$`request`) to P_j. When it receives the value (`Establish-session,` sid, `responder`) from P_j in response it sends (sid, `Establish-Session, respondor`, P_j) to the adversary, and it sets the session as `active` between client P_i and server P_j.
2. Upon receiving a value (`Send,`sid, m) from a party P_e, it checks to see if the session sid is `active` and that P_e is either P_i or P_j. It then sends (sid, P_e`,|m|`) to the adversary and when it receives a positive response it sends (`Received,`sid, m) to the other partner in the session.
3. Upon receiving a value (`Expire-session`, sid) from either the client or server in the session it sets state to `inactive` and public delayed informs the other party.

Figure 2: The SSL functionality, $\mathcal{F}_{\text{SSL}}$

Finally, to allow a **compositional** definition, we assume that all parties (except the ideal third party, but including the Adversary) are driven by an all encompassing entity called the **Environment**. In other words, the inputs of the parties (possibly related) are actually provided by the Environment so as to cover all possible scenarios. This notion of Environment is *not necessary* to understand ideal functionalities, but it will become important when we discuss compositional security.

Coming to our first example, we discuss the ideal functionality "Secure Channel" described in Fig 1. It is the ideal functionality representing authenticated and encrypted channels between two globally known parties, e.g. two parties with public keys in a PKI. Since, the same two parties may be involved in multiple different secure channel sessions (possibly concurrently), we assume that the two parties have decided on a session id (sid) before hand; infact this sid may be decided by the Environment as it drives all the parties anyway. We emphasize that this definition of secure channel requires that the parties have globally known names. For instance when party P_i initiates a session with an Establish-session input to the ideal functionality, it is naming a party P_j with which it wants to have the secure channel session. Similarly, the same named party P_j must respond with the Establish-session input while naming P_i. Once, both parties have supplied their Establish-Session inputs, while naming the correct counter-parties, the ideal functionality sets its local state to *active*. From then on, either party can send a message to the other party using the ideal functionality, which only leaks the length of each message to the Adversary (it is possible to have an ideal functionality which does not even reveal the length of the message to the adversary, but implementing such a functionality in the real world may be very inefficient).

Moving on to the next example, the SSL ideal functionality in Fig 2, one notices in contrast to the secure-channel functionality above that only one of the peers is supposed to have a globally known name, and the session can be initiated by any party P, and whose name is never referred to in any messages (initialization or otherwise). The adversary is leaked information about the name being used by this client (e.g. a temporary IP address), which is unavoidable in any real-world implementation.

3 Introduction to Simulation Based Security Definition

While the previous section only described a way to define the desired goal of a protocol using an ideal third party, one can also define security of a **real-world protocol** by relating it to this ideal functionality. Note that the ideal functionality has precisely characterized what output each legitimate party gets, and what is leaked to the Adversary. It also characterizes directives that an Adversary unavoidably controls, e.g. whether a message should be delivered to a counter-party or not. Thus, if we can show that the real world protocol has the same characteristics, we have managed to prove desired security constraints of the real protocol.

A real world protocol between some parties is said to **realize** an ideal functionality, if for every adversary in the the real world protocol, there is an adversary in the ideal world protocol (i.e. involving the ideal trusted third party representing the ideal functionality) such that the Environment interacting with the parties and the two adversaries (the real world and the ideal world) cannot distinguish between the two scenarios[3]. Thus, in this definition the Environment gets the same amount of information in the real and the ideal world. Since the ideal world precisely defined what the Environment gets (actually what the Adversary gets, which it reports back to the Environment), one concludes that in the real world protocol also the Environment (and hence the

[3]The main idea is the generalization of the concept of zero-knowledge. A protocol is said to be zero-knowledge if for every adversary in the real world that gains some private knowledge there exists another adversary which can gain the same knowledge from an ideal execution of the protocol. From this, one can then conclude that since the ideal execution leaks (almost) zero information which is meant to be private, then the adversary in the real world also gets zero information. Thus, the real protocol is secure in the sense that all information which is meant to be private remains private.

adversary) gets only the precisely defined information.

More rigorous definitions can be found in [1].

Such *realization proofs* are done by a **simulation** argument. Given a real world protocol, and an Adversary $\mathcal{A}$ in the real world, the proof constructs a Simulator $\mathcal{S}$ that simulates the real world protocol to $\mathcal{A}$. To do this simulation, $\mathcal{S}$ is assumed to reside in the ideal world, and hence has access to the Ideal Functionality (as an ideal world Adversary $\mathcal{S}$). If such a simulation is possible, then we have managed to construct for each adversary $\mathcal{A}$ in the real world an adversary $\mathcal{A}'$ (which is obtained by juxtaposing $\mathcal{S}$ and $\mathcal{A}$) in the ideal world, such that the Environment can not distinguish the two scenarios, and hence, by definition, security follows.

3.1 Utility of Ideal Functionality Based Definition

Since our definition of security is based on an ideal functionality, which sometimes describes in tedious details the various input/output characterizations and interactions with the adversary etc., one may wonder if one has covered all issues in such a definition. Luckily, most of the time the definitions are straightforward, and just describes the parties handing over their inputs, and then getting back the desired outputs, as well as a characterization of leakage to the Adversary. Thus, it is easy to be convinced that such a definition is the required security definition.

In many other cases, the protocol even in this ideal setting maybe more complicated, with multiple interactions. While, an advanced reader or someone well-versed with the issues may easily be convinced about the robustness of the definition, readers new to the issue may not be convinced. Fortunately, there is a way to validate these definitions, by using these protocols in higher level protocols, the latter being the end goal anyway.

In the compositional paradigm mentioned earlier, a theorem of Canetti [1] called the com[position theorem shows that once a real protocol ρ has been proven to realize an ideal functionality $\mathcal{F}$, then one can build higher level real world protocols which realize higher level functionalities, say $\mathcal{G}$, by assuming the availability of ideal functionality $\mathcal{F}$ in the real-world. The **composition theorem** proves that having obtained such a hybrid protocol (i.e. a real-world protocol using ideal functionality $\mathcal{F}$), one can convert it to be a fully real-world protocol by embedding ρ in place of $\mathcal{F}$.

If the ideal functionality $\mathcal{G}$, which is the end goal, is clearer to understand and characterize (hopefully this is the case as it is a high level goal), what we get is a real world protocol which realizes $\mathcal{G}$. Then, we do not need to worry about whether $\mathcal{F}$ was a correct characterization, and in fact we have indirectly validated that $\mathcal{F}$ had the desired properties in the sense that it helps us securely implement $\mathcal{G}$.

As an example, suppose we are defining an ideal functionality for low level protocol OAuth. This low level protocol has many flows even with an ideal third party, and an ideal characterization may not be convincing enough. However, if one realizes that the goal of OAuth is to be a sub-protocol in a bigger desired goal, say *delegated computation*, then if we can realize delegated computation with access to (how so ever defined) ideal functionality OAuth, and further we have proven that there is a real world protocol ρ which realizes this ideal functionality OAuth, then we get a real-world realization of delegated computation ideal functionality. Hopefully, the delegated computation ideal functionality is easier to understand, as it only has high level goals (and high level leaks to the adversary) which are well understood.

4 Conventions for Defining Ideal Functionalities

Delayed Messages When an ideal functionality outputs a value to a party, this delivery of value can be either **public delayed** or **private delayed**. Essentially, since the network in the real-world is assumed to be under adversarial control, one can only guarantee delivery of messages in the real-world if the adversary complies. Thus, any value to be delivered to a party is termed "delayed", which is a short form saying that the functionality requests the Adversary to deliver the value, and if the adversary responds positively, then the value is actually delivered. When the functionality terms the delivery public delayed, it means that the adversary gets to see the actual value, whereas if it is termed private delayed, then the adversary only gets to see the length of the message in bits.

Corruption Each ideal functionality has a function called **Corrupt** which the Adversary can call to declare a globally known entity (i.e. one with a public key in PKI) corrupted. The functionality then records in its local state that the party is corrupted. The functionality may choose to have a different behaviour based on whether certain parties are declared corrupted. Normally, this would mean, for example, instantly revealing some internal state of the ideal functionality to the Adversary, as that is what happens in the real-world. Ideally speaking, one would like to have the case that even if some parties in a protocol are corrupted, it does not affect the privacy of other parties, and most ideal functionality would be defined in that fashion. However, one cannot exclude some side effects from happening when some party in the protocol is corrupted.

As an interesting example related to OAuth, suppose that legitimate Clients (or Consumers) are registered with the Service Provider. In fact, under PKI attestation of the URIs of the Clients, the Service Provider just needs to know the global identity of these Clients. If a User agent contacts (under SSL) a Service Provider with a URI of a Client (attested under PKI to be associated with a globally known name P_c), and further if the User Agent manages to provide correct userid_A, password_A to the Service Provider under the same SSL session, then the Provider can assuredly provide the client P_c information related to the account userid_A. Now consider the situation where the password_A was easy to guess, and hence the User Agent was being driven by an illegitimate user (Adversary). If the Client was not authenticating the User on its own, then the Adversary would end up getting information related to userid_A (via the Client). But, if the Client was also authenticating the User under possibly a different userid_B, password_B, and if this password was tough to guess for the Adversary, then the Client will never open a session with this illegitimate user. However, a different Client P'_c which is a pre-registered Client may be malicious and may join forces with the Adversary. In this case, as soon as the illegitimate user presents userid_A, password_A, P'_c to the Service Provider, the information related to userid_A is essentially given to the Adversary by the Provider. The ideal functionality would do exactly that, but it needs to be informed that P'_c has been corrupted (i.e. it has joined forces with the Adversary).

Session Id The session identifier or *sid* is assumed to be unique (but non-secret) for each instantiation of the real-world protocol or the ideal functionality. Usually, in the Universal composability paradigm [1], the *sid* is assumed to be determined by a wrapper protocol (hence the Environment) between all legitimate parties, and this pre-determined *sid* is passed along with other inputs by the Environment to all legitimate parties of the protocol. How-

ever, this wrapper protocol could lead to an inefficient complete protocol, as it may take additional flows. Thus, in our definition of OAuth's ideal functionality, we have minimized any such wrapper protocol for determination of *sid*, so that the resulting implementation is in line with practical implementations of OAuth. One implication of this is that the ideal functionality has many more flows than one would expect. For example, the Service Provider is not pre-determined by the *sid*, and is actually passed as an input by the Client. Similarly, the Client is not pre-determined by the *sid*, and the Client's global identity is passed to the Provider (with possible Adversarial manipulation of this identity) in the ideal functionality.

5 The OAuth Ideal Functionality

Before we embark on giving a definition of the OAuth Ideal Functionality, we paraphrase the following abstract from the OAuth V2 Internet draft ([2]):

"*The OAuth 2.0 authorization protocol enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.*"

In this paper we will focus mainly on the first kind of delegation which requires an approval interaction. In the ideal functionality definition, we will refer to the three parties named above as **User** (resource owner), **Provider** (HTTP Service), and **Consumer** (third-party). Moreover in this section, we will assume that the Provider and the Consumer have globally known names, e.g. public keys with certified domain names in PKI. Only the User will *not* be assumed to have a global name. The case where the Consumer has a public key which is not certified by a global CA is dealt with in a later section.

In particular, the most interesting and wide-spread case arises where the User has a $\langle \text{userid}, \text{password} \rangle$ account with a Provider, which needs to be delegated to the Consumer. The basic idea of the definition can be summarized as follows: A User starts of using a software provided by a Consumer P_c, say as a website with PKI certified public key. To model this, the User calls the ideal functionality with an input message $(sid, \texttt{initiate}, P_u, P_c)$. Here *sid* is a non-secret but unique session identifier which is chosen by the User or Environment (wrapping the User). Note that P_u may just be a temporary identifier, e.g. a temporary IP address. The Consumer, at some point decides that it needs access to data of the User held at a Provider P_s, again P_s being a website with a PKI certified public key (see Fig 3 for a formal description and Fig 8 for a pictorial depiction). The ideal functionality then forwards the name P_s to the User, who in response provides a userid and password to the ideal functionality (corresponding to its account at P_s). The ideal functionality then provides the userid and the name P_c to P_s. At this point, P_s may abort, if for example it does not wish to provide data to P_c, or if it deems userid to be invalid. Otherwise, it responds to the ideal functionality with the pre-registered pw' corresponding to the userid provided[4].

The ideal functionality next checks the two passwords for equality (the one supplied by the

[4]Note that, all parties are really being driven by wrapper softwares around these parties, which we treat as a single monolithic Environment. The main reason for treating the Environment as monolithic is that one *cannot* assume that the individual wrapper softwares are *not* talking to each other via some other protocols and in the process leaking information. Thus, the universal composability (UC) paradigm lets one analyze the security of the protocol even if the wrapper softwares are buggy, or downright maliciously collusive.

User and the one returned by the Provider). If they are not equal, the session is aborted, and the Adversary is also leaked this information (as it is highly inefficient to not leak this information in the real-world). If they are equal then the functionality sets a local status variable, say authentication status to valid, and this status is also revealed to the Adversary.

Next, the ideal Functionality issues a randomly generated AccessToken to the Provider P_s and Consumer P_c, but only at the Adversary's directive, i.e. the Adversary can cause denial of service. This common and secret AccessToken may be used by the two parties to communicate any information related to the account corresponding to userid. Note again that the wrapper software of the Provider is given the userid and the AccessToken, and if the wrapper software is buggy, there are no security guarantees. The ideal functionality only guarantees that a userid has been provided to the Provider, and further that the password the Provider gave for this userid is same as the password provided by some arbitrary party P_u, and that the Provider shares the AccessToken with a globally known entity P_c, the identity P_c being supplied by the same party P_u.

The ideal functionality also provides a capability for the User and the Consumer to send messages to each other secretly. The authentication of messages originating at P_c do not need any particular mention since P_c has a global public key, but what is noteworthy is that the delivery of these messages is *only* to the User who provided the correct password to the userid submitted to the Provider.[5]

The OAuth Ideal Functionality is defined in detail in Figure 3. In this detailed definition, one notices that the Adversary makes the "Issue Key" calls. As already mentioned, this makes sense as the network is assumed to be insecure, and hence the Adversary is assumed to control the network. Thus all network flow is directed by the Adversary, including the issuing of keys. Note that the Adversary does not control what keys are delivered (unless it has compromised the session), but only control when and if the keys are delivered.

There are certain other intricacies related to **password-based authentication**, which can be ignored in a first reading. Since the password in password-based accounts is usually human memorizable, it can not be assumed to have full 128 bit entropy (when talking about 128-bit security for the protocol). Thus, the ideal functionality we define must not guarantee 128-bit security, since any implementation of such an ideal functionality would require random 128 bit passwords.

However, as described in [**?**, **?**], one can define an ideal functionality which guarantees that an Adversary can only by-pass the 128-bit security by performing online guessing attacks. Thus, other than these guessing attacks (which are un-avoidable), 128-bit security is guaranteed by the ideal functionality. So, how does one define an ideal functionality where such guessing attacks are feasible? The functionality allows the Adversary to call it with a guess of the password, and if the guess is correct, the Adversary is allowed to set shared keys of its choosing (basically giving it access to the secure channel being setup), and if the guess is incorrect then the session is considered interrupted, and the legitimate parties (essentially) abort.

[5]Note that if the Adversary does not deflect the protocol, and change userid to some other userid', then the account corresponds to userid as provided by P_u. If however, the Adversary (say posing as another P'_u) hijacks (or deflects) the flow of the protocol by injecting a different userid', then OAuth will continue in normal fashion only if the Adversary provides the correct password for userid', and then it will get results corresponding to this account.

6 Implementation of Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$

The implementation of the Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$ assuming ideal functionalities for SSL and Secure Channels is given in Figure 4. This implementation does not assume that the User program is just a User Agent (i.e. a browser implementing http). However, care has been taken to make this implementation easily refined into one where the User code can just be replaced by a browser that supports redirection. Indeed, such a implementation is given in Fig 5. Further, another implementation where a User may instead be sent a hyper-link of the Provider in an authenticated email (instead of a response to an https request) from the Consumer is given in Fig 6. This implementation is a further refinement of the one in Fig 5. Next, an implementation is given in Fig 7, where the hyper-link is sent via an unauthenticated channel, e.g. posted on a bulletin board. this implementation is a further refinement of the previous ones. Special attention should be paid to the notes at the bottom of the figures.

Theorem 1 *The implementation realizes the functionality* $\mathcal{F}_{\text{OAUTH}^*}$.

Proof: We will show that for every probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a PPT adversary $\mathcal{A}$', such that the ensembles, corresponding to the view of the environment in the experiment where it interacts with the adversary $\mathcal{A}$' involved with the ideal functionality, and the experiment where it interacts with the adversary $\mathcal{A}$ involved with the real implementation, are indistinguishable. The adversary $\mathcal{A}$' in the ideal world is obtained by composing a simulator $\mathcal{S}$ with $\mathcal{A}$ itself, where $\mathcal{S}$ using access to ideal functionality (as ideal world adversary) simulates the real-world to $\mathcal{A}$. Further, $\mathcal{S}$ will ensure that the interaction of $\mathcal{F}_{\text{OAUTH}^*}$ with environment is also indistinguishable with interaction of real parties with the environment. Note that in the ideal world experiment, the parties P_u, P_c, P_s run as dummies and just pass back and forth the messages between $\mathcal{Z}$ and $\mathcal{F}_{\text{OAUTH}^*}$, whereas in the real world, the parties P_u, P_c, P_s are running the protocol π. Thus to simulate the real world to $\mathcal{A}$, $\mathcal{S}$ will need to simulate the real parties P_u, P_c, P_j to A. We will show that S is able to do so (using access to ideal functionality$\mathcal{F}_{\text{OAUTH}^*}$), and hence as far as the environment Z is concerned the ideal world and the real world are indistinguishable.

We will show later how $\mathcal{S}$ simulates SSL and SC, but for now we deal with how $\mathcal{S}$ simulates the real world parties. To begin with, in both worlds, the environment $\mathcal{Z}$ sends the input $(sid, \texttt{initiate}, P_u, P_c)$ to P_u. In the real-world, this prompts P_u to Establish an SSL session by calling the ideal functionality $\mathcal{F}_{\text{SSL}}$, which in turn calls the Adversary $\mathcal{A}$ for a response. This call to $\mathcal{A}$ needs to be simulated by $\mathcal{S}$, but since in the ideal world $\mathcal{S}$ gets called by $\mathcal{F}_{\text{OAUTH}^*}$ as well, $\mathcal{S}$ can at that point simulate a call to $\mathcal{A}$. If $\mathcal{A}$ responds positively in the real-world, $\mathcal{S}$ in the ideal world calls $\mathcal{F}_{\text{OAUTH}^*}$ positively (note $\mathcal{S}$ is using $\mathcal{A}$ as a blackbox). Note that this leads to an output via P_c to the environment of value $(sid, \texttt{params-req}, P_u, P_c)$ in both worlds.

Next, in both worlds, the environment $\mathcal{Z}$ sends input $(sid, \texttt{params}, P_s)$ to P_c. In the real world, this prompts P_c to **send** a message via the same SSL session to P_u, which is promptly output back to the environment. In the ideal world, the simulator $\mathcal{S}$ gets notified by the ideal functionality $\mathcal{F}_{\text{OAUTH}^*}$ of this input from P_c (as it is being sent to P_u public delayed), and hence $\mathcal{S}$ can simulate a call to $\mathcal{A}$ regarding the size of the message in bits (which is what $\mathcal{F}_{\text{SSL}}$ does in the real world). Further note that the message is also output to the environment in the ideal world.

Similarly, as long as none of the SSL session establishments are replaced by the Adversary by its own establishment using corrupted parties, the simulation can be done by $\mathcal{S}$.

Now consider the cases where one or more of the SSL sessions $sid^{\text{UC1}}_{\text{SSL}}$, $sid^{\text{UC2}}_{\text{SSL}}$, $sid^{\text{UP}}_{\text{SSL}}$ may be initiated by a corrupt principal (P'_u). We don't have to consider the secure channel $sid^{\text{CS}}_{\text{SC}}$ because a secure channel authenticates both the peers, and we do not consider corruption of the client and/or the provider here.

Case $sid^{\text{UC1}}_{\text{SSL}}$ is initiated by a corrupt principal: First we see how an Adversary $\mathcal{A}$ in the real-world accomplishes this. Assume that the Adversary has managed to corrupt a user (lets call it P'_u; infact the user P'_u in this case may just be the Adversary). From now on we will identify P'_u with $\mathcal{A}$, which goes well with our convention that there is only one monolithic Adversary. Next, when the legitimate user P_u makes an Establish Session call to $\mathcal{F}_{\text{SSL}}$, the Adversary $\mathcal{A}$ does not respond with a positive response, and hence in a sense that SSL connection never takes place. Instead, the Adversary (via P'_u) initiates a new SSL establish session call to another instance of $\mathcal{F}_{\text{SSL}}$ naming the same consumer P_c as server.

since the Adversary is making this Establish session call to $\mathcal{F}_{\text{SSL}}$, the Simulator which is treating $\mathcal{A}$ as a black box can see this call being issued. Similarly, when $\mathcal{A}$ (via P'_u) sends a message $(sid, \texttt{par-req})$ by calling the Send function of $\mathcal{F}_{\text{SSL}}$, again $\mathcal{S}$ gets to see this call in the plain, and checks for integrity of the message, and if so, just responds positively to the ideal functionality $\mathcal{F}_{\text{OAUTH}^*}$'s public delayed output to P_c. Thus, in both worlds P_c outputs the same value `par-req` to $\mathcal{Z}$. Note that in the real-world P_c has no idea whether P'_u is legitimate or not, and hence as long as the message was of proper syntax, it will output it to $\mathcal{Z}$.

Case $sid^{\text{UP}}_{\text{SSL}}$ is initiated by a corrupt principal: Again, as in the previous case, the Adversary may not respond positively to a legitimate SSL establish-session request, and instead start its own SSL session with the Provider P_s. However, since it is $\mathcal{A}$ that sends a message $(sid, \texttt{initiate-req}, userid'', pw'', P''_c)$ using this SSL session, $\mathcal{S}$ is able to see the message. If the message is not syntactically correct then $\mathcal{S}$ just stops. $\mathcal{S}$ then sends the message $(sid, \texttt{initiate-req}, userid'', P''_c)$ to $\mathcal{F}_{\text{OAUTH}^*}$ which the latter is expecting. If $userid$ or P_c have been altered from what $\mathcal{F}$ sent to $\mathcal{S}$, then F sets its deflection status to deflected. Note that this can only happen if $\mathcal{A}$ started its own SSL session, although $\mathcal{A}$ may choose to keep $userid$ and P_c same (which although not revealed to $\mathcal{A}$ in the real-world protocol, may still be easily guess-able).

Next, the environment receives the output $(sid, \texttt{initiate-req}, userid'', P''_c)$ in both the worlds. It responds with $(sid, \texttt{password}, pw')$ if the environment (i.e. P_s's wrapper software) determines that $userid''$ and P''_c are valid parameters (or it may respond with `bad-params`; this case is easy to handle and we skip this case). Note that if P''_c is different from P_c, but still a pre-registered Consumer, then the environment is likely to pass it as a valid parameter. Further, since the SSL session is being initiated by the Adversary, the initial user P_u is out of the picture. There are two sub-cases depending on the deflection status of $\mathcal{F}$.

normal: In this case, on the input pw' from P_s, the functionality F sets its authentication status based on whether pw' is same as pw (supplied by P_u). However, this is not what happens in the real-world, where pw' is compared with pw'', the latter being supplied by $\mathcal{A}$. Thus $\mathcal{S}$ must call `TestPwd` function of $\mathcal{F}$ with pw'', which leads to F ignoring (or clearing) the authentication status, and instead setting the usurpation status based on whether pw' is same as pw''. (Note that $\mathcal{S}$ had obtained pw'' when $\mathcal{A}$ issued a send

message call to $\mathcal{F}_{\text{SSL}}$ – see previous paragraph.) Now, both the real-world and the ideal-world are in sync.

`deflected:` In this case, on the input pw' from P_s, the functionality F does not set its authentication status anyway, so $\mathcal{S}$ just needs to call F's `TestPwd` function with pw''.

Next in the real world, P_s sends the message $(sid, \texttt{authgrant}, AuthCode)$ over $sid^{\texttt{UP}}_{\text{SSL}}$ only if passwords matched, in which case in the ideal world $\mathcal{S}$ (since it s informed the usurpation status) just generates random Authcode of its own and delivers it to $\mathcal{A}$.

Note that, in the real-world the Adversary must initiate its own SSL session UC2 with the Consumer, otherwise the Authcode it obtained from the Provider cannot be delivered back to the Provider via the consumer. If $\mathcal{A}$ indeed starts its own SSL session with P_c, then since it will call the SSL session with a Send Authcode, the Simulator can check if this Authcode is same as the one it generated for it.

Case $sid^{\texttt{CS}}_{\text{SC}}$ is initiated by a corrupt party P'_c. In the real-world P_s checks that this P'_c is same as the parameter sent to it in $sid^{\texttt{UP}}_{\text{SSL}}$, or more precisely its replacement initiated by Adversary $\mathcal{A}$. Thus, P_s only continues with the protocol in the real-world if $sid^{\texttt{UP}}_{\text{SSL}}$ was replaced by Adversary, which had passed a P'_c then and which was validated by the Environment (i.e. the environment did not respond with `bad−params`. In such a scenario, recall that the ideal-world has set its deflection status as deflected (note corrupt P'_c is not same as P_c). This means that the authentication status is empty. Now, if in the real-world P'_c also happens to report the correct Authcode in $sid^{\texttt{CS}}_{\text{SC}}$, then P_s and P'_c (i.e. $\mathcal{A}$) will end up sharing a common random AccessToken. To emulate this situation, $\mathcal{S}$ in the ideal world just calls Issue AccesToken to Provider with a randomly chosen k. This would lead to setting the AccessToken of the Provider to k if usurpation status was `compromised` (i.e. $pw'' = pw'$) and P'_c is corrupted (which it is). The simulator $\mathcal{S}$ also sends a message to P'_c (i.e. $\mathcal{A}$) $(\texttt{accesstoken}, k)$ under the appropriated $sid^{\texttt{CS}}_{\text{SC}}$, thus completely emulating the real-world.

Case $sid^{\text{UC2}}_{\text{SSL}}$ is initiated by a corrupt principal: If in the real-world the first two SSL sessions were legitimate, and only this one is initiated by $\mathcal{A}$ then $\mathcal{A}$ can provide the correct Authcode to Consumer only with negligible probability, as Authcode is generated randomly. Thus, in this case in the real-world, the Provider and hence the Consumer will not output a common and random AccessToken. Since, the Simulator $\mathcal{S}$ knows that the first two SSL sessions were legitimate, and this one is not, it just does not make the calls to $\mathcal{F}_{\text{OAUTH}^*}$'s Issue Access Token functions.

If on the other hand, the SSL session between User and Provider was taken over by the Adversary, then, we know from the previous case that the Adversary knows the usurpation status, and in case of compromised status, has provided a random Authcode of its own. If $\mathcal{A}$ in this corrupted SSL session sends the same Authcode to P_c, the $\mathcal{S}$ gets to see that, and hence it means in the real world since P_c and P_s behave honestly, P_s will end up issuing valid (and randomly generated) AccessToken to P_c. So, $\mathcal{S}$ just makes the Issue AccessToken calls for both Consumer and Provider with the same randomly chosen value k. Thus, the view of the Environment in both the worlds will be same.

As for the $(sid, \texttt{Send}, m)$ values given by the Environment to P_c in the real world, note that P_c only sends it to the peer if it had obtained an AccessToken, which is only possible if if the

password pw'' the Adversary had provided matched the password P_s had produced, which means the usurpation status is `compromised`. Thus in this situation, if this last SSL session was started by the Adversary, in the real world the Adversary would get the message m, and the same would hold in the ideal world.

□

7 Acknowledgements

The authors would like to thank Michael Steiner for several helpful comments. The authors would also like to thank Pau-Chen Chang, Mark Mcgloin, David Robinson, Jeffrey Hoy and Prashant Kulkarni for helpful discussions, and Mark in particular for encouraging us to write this paper.

References

[1] Ran Canetti, " Universally Composable Security: A New Paradigm for Cryptographic Protocols", Proc. Foundations of Computer Science, 2001: 136-145.

[2] The OAuth 2.0 Authorization Protocol draft-ietf-oauth-v2-20. `http://tools.ietf.org/html/draft-ietf-oauth-v2-20`.

Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$

Participants: An arbitrary User P_u, Service Provider P_s, Consumer P_c, and Adversary $\mathcal{S}$ interact with functionality $\mathcal{F}_{\text{OAUTH}^*}$ (or $\mathcal{F}$).

Status Variables: Deflection Status (default: normal), Usurpation Status (default: normal), Authentication Status (default: empty).

Initiate: On receiving an input message $(sid, \texttt{initiate}, P_u, P_c)$ from P_u, output (public delayed) the message $(sid, \texttt{params-req})$ to P_c. On receiving a reply $(sid, \texttt{params}, P_s)$ from P_c, output (public delayed) the message $(sid, \texttt{params}, P_s)$ to P_u.

On receiving a response $(sid, \texttt{credentials}, userid, pw)$ from P_u, output $(sid, \texttt{initiate-req}, userid, P_c)$ to adversary $\mathcal{S}$. On receiving a response $(sid, \texttt{initiate-req}, userid', P_c')$ from adversary S, output $(sid, \texttt{initiate-req}, userid', P_c')$ to P_s. Further, if $userid'$ is not the same as $userid$ or P_c' is not the same as P_c, set the *deflection status* as `deflected`.

Next, on receiving a response $(sid, \texttt{password}, pw')$ from P_s, record pw' locally. If the *deflection status* is not set as `deflected`, then check if $pw = pw'$, and if the two passwords are not the same then set the *authentication status* as `aborted` and output $(sid, \texttt{abort-oauth})$ to the Adversary. Otherwise (if the passwords are same), set *authentication status* as `valid` and output $(sid, \texttt{initiate-oauth})$ to the Adversary. If the deflection status is set as `deflected`, skip the above steps and wait for a TestPwd call from $\mathcal{S}$.

Instead of responding with a pw', P_s may instead respond with $(sid, \texttt{bad-params})$, in which case the functionality sets the *authentication status* as `aborted` and outputs $(sid, \texttt{bad-params})$ to $\mathcal{S}$.

The following call may be made by the Adversary $\mathcal{S}$ (and only $\mathcal{S}$).

TestPwd: On receiving a message $(sid, \texttt{testpwd}, pw'')$ from S: If pw' is not recorded locally, then ignore this call. Else, if $pw'' = pw'$ then set usurpation status as `compromised`, otherwise set usurpation status as `interrupted`. Report the usurpation status to $\mathcal{S}$. In either case reset *authentication status* to empty. Note that deflection status is immaterial here, and $\mathcal{S}$ can make this call even if deflection status is normal.

The following two calls may also be made by the Adversary $\mathcal{S}$ (and only $\mathcal{S}$) in any order.

Issue Access Token to Consumer: On receiving a message $(sid, \texttt{IssueKey2Consumer}, k)$ from S, if the authentication status is set as `valid` then output a randomly generated *AccessToken* to P_c. Else, if the usurpation status is set as `compromised`, output k (provided as a parameter by the Adversary) to P_c. In all other cases, obtain fresh $r \leftarrow \$$, and output r to P_c .

Issue Access Token to Service Provider: On receiving a message $(sid, \texttt{IssueKey2SP}, k)$ from S: if the authentication status is set as `valid` then output the same *AccessToken* as above to P_s. Else, if the usurpation status is set as `compromised`, *and* P_c' is same as P_c or P_c' is corrupted, output k (provided as a parameter by the Adversary) to P_s. In all other cases, obtain fresh $r \leftarrow \$$, and output r to P_s .

Send: On receiving an input $(sid, \texttt{Send}, m)$ from $\hat{P}$ which is either P_u or P_c, check if the authentication status is set as `valid`. If so, send $(sid, \hat{P}, |m|)$ to Adversary $\mathcal{S}$, and after receiving a positive deliver response from $\mathcal{S}$, output $(sid, \texttt{Received}, m)$ to the counter-party of $\hat{P}$ (i.e. P_c or P_u resp.). If instead, the usurpation status is set as `compromised` and if `Send` originates from P_c (i.e. $\hat{P}$ is P_c), and $P_c' == P_c$, then send $(sid, \hat{P}, m)$ to adversary $\mathcal{S}$ (i.e. the Adversary gets the message m). In all other cases, send $(sid, \hat{P}, |m|)$ to $\mathcal{S}$ and take no further action.

Figure 3: A functionality for delegation with explicit key exchange

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using $\mathcal{F}_{\text{SSL}}$ and $\mathcal{F}_{\text{SC}}$

Participants: The User Agent (P_u), Service Provider (P_s), Consumer (P_c), and Adversary $\mathcal{A}$.

Initiate: On receiving $(sid, \texttt{initiate}, P_u, P_c)$ from $\mathcal{Z}$, P_u initiates an SSL session with session id $sid^{\text{UC1}}_{\text{SSL}}$ with P_c and sends the message $(sid, \texttt{params-req}, P_u, P_c)$. On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over SSL session $sid^{\text{UC1}}_{\text{SSL}}$, P_c outputs $(sid, \texttt{params-req}, P_u, P_c)$. On receiving $(sid, \texttt{params}, P_s)$ from $\mathcal{Z}$, P_c sends the message $(sid, \texttt{params}, P_s)$ to P_u using SSL session $sid^{\text{UC1}}_{\text{SSL}}$.

On receiving $(sid, \texttt{params}, P_s)$ over $sid^{\text{UC1}}_{\text{SSL}}$, P_u outputs $(sid, \texttt{params}, P_s)$ to $\mathcal{Z}$. On receiving $(sid, \texttt{credentials}, userid, pw)$ from $\mathcal{Z}$, P_u initiates an SSL session with P_s with session id $sid^{\text{UP}}_{\text{SSL}}$ and sends $(sid, \texttt{initiate-req}, userid, pw, P_c)$. On receiving $(sid, \texttt{initiate-req}, userid, pw, P_c)$ over the SSL session $sid^{\text{UP}}_{\text{SSL}}$, P_s queries the environment with $(sid, \texttt{initiate-req}, userid, P_c)$. The environment responds with $(sid, \texttt{password}, pw')$. P_s tests whether $pw = pw'$. If the check succeeds, P_s generates a random $AuthCode$, records $(sid, P_s, P_c, Authcode)$, and sends the message $(sid, \texttt{authgrant}, AuthCode)$ over $sid^{\text{UP}}_{\text{SSL}}$.

If the environment responds to P_s with $\texttt{bad-params}$, P_s just aborts.

On receiving $(sid, \texttt{authgrant}, Authcode)$ over $sid^{\text{UP}}_{\text{SSL}}$, P_u sends $(sid, \texttt{authgrant}, Authcode)$ to P_c over a *new* SSL session $sid^{\text{UC2}}_{\text{SSL}}$.

On receiving $(sid, \texttt{authgrant}, Authcode)$ over $sid^{\text{UC2}}_{\text{SSL}}$, P_c initiates a secure channel $\mathcal{F}_{\text{SC}}$ with P_s with sid $sid^{\text{CS}}_{\text{SC}}$ and sends P_s the message $(sid, \texttt{authgrant}, P_c, Authcode)$. On receiving $(sid, \texttt{authgrant}, P_c, Authcode)$ over $sid^{\text{CS}}_{\text{SC}}$, P_s checks it against the recorded information, and makes sure that the Secure channel is with the same global entity P_c as recorded earlier. If the check succeeds then it generates $AccessToken$ and sends $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\text{CS}}_{\text{SC}}$. It also outputs $(sid, \texttt{keyCS}, AccessToken)$ to the environment.

On receiving $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\text{CS}}_{\text{SC}}$, P_c outputs $(sid, \texttt{keyCS}, AccessToken)$ to the environment.

Send: On receiving $(sid, \texttt{send}, m)$ from the environment, P_u sends the message $(sid, \texttt{send}, m)$ to the peer over SSL session $sid^{\text{UC2}}_{\text{SSL}}$. On the other hand, on receiving $(sid, \texttt{send}, m)$ from the environment, P_c checks that it has obtained an AccessToken from P_s, and only then it sends the message $(sid, \texttt{send}, m)$ to the peer over SSL session $sid^{\text{UC2}}_{\text{SSL}}$.

Notes:
1. P_u can use the older SSL session $sid^{\text{UC1}}_{\text{SSL}}$ instead of a fresh $sid^{\text{UC2}}_{\text{SSL}}$, if the former is still active.
2. If the Consumer also requires a password-based authentication of the User (possibly with a different userid, password than the one which the User has with the Provider), then if a fresh $sid^{\text{UC2}}_{\text{SSL}}$ is initiated, then this password-based authentication by the Consumer must take place afresh as well.

Figure 4: OAuth v2 Authorization Grant Flow

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using HTTPS Redirection

Participants: An arbitrary User Agent P_u, a Service Provider P_s, a Consumer P_c, and Adversary $\mathcal{A}$. The Service Provider and Consumer are assumed to have SSL supporting URIs with PKI certificates.

Initiate: On receiving an input $(sid, \texttt{initiate}, P_u, P_c)$ from the environment, P_u initiates an **https** session with session id $sid^{\texttt{UC1}}_{\text{SSL}}$ with the URI of P_c and query parameters $(sid, \texttt{params-req}, P_u, P_c)$.

On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over https $sid^{\texttt{UC1}}_{\text{SSL}}$ which it outputs to the environment, P_c obtains the input $(sid, \texttt{params}, P_s)$ from the environement. Then P_c responds with $(sid, \text{redirect uri:}P_s, \texttt{client-params})$ to P_u using SSL session $sid^{\texttt{UC1}}_{\text{SSL}}$. Since P_s is a globally known entity, the Consumer P_c can obtain a valid https URI of P_s. The query parameters called `client-params` includes the https redirection URI of P_c itself, and other credentials of P_c. Since the response is a redirection https URI, the user agent P_u automatically initiates an SSL session with P_s with session id $sid^{\texttt{UP}}_{\text{SSL}}$ and query parameter $(sid, \texttt{client-params})$.

On receiving this message, P_s responds to P_u over $sid^{\texttt{UP}}_{\text{SSL}}$, with a login-password form, to which the user agent responds by outputting the form along with P_s's identity to the environment (which in this case is possibly just a human User along with a certificate checker). The Environment (or the human) responds with a *userid* and password pw, which the User agent forwards to P_s using $sid^{\texttt{UP}}_{\text{SSL}}$. Next, P_s queries the environment with the received userid along with `client-params` (since P_c is also a globally known entity, this is just equivalent to outputting the identifier P_c itself), to obtain a password pw' corresponding to this userid (or a `bad-params` response). On pw validation (i.e. $pw = pw'$), P_s generates a fresh random number $Authcode$, and saves $(sid, \texttt{initiate}, P_s, P_c, Authcode)$ in its local memory, and responds over $sid^{\texttt{UP}}_{\text{SSL}}$ with the redirect URI of P_c (obtained from `client-params`) and query parameter $(sid, \texttt{authgrant}, AuthCode)$.

Since the user-agent P_u receives a redirect URI, which is an https URI of P_c, it automatically starts a new SSL session $sid^{\texttt{UC2}}_{\text{SSL}}$ with P_c, over which it sends query paramter $(sid, \texttt{authgrant}, AuthCode)$. On receiving $(sid, \texttt{authgrant}, Authcode)$ over $sid^{\texttt{UC2}}_{\text{SSL}}$, P_c initiates a secure channel $\mathcal{F}_{\text{SC}}$ with P_s with sid $sid^{\texttt{CS}}_{\text{SC}}$ and sends P_s the message $(sid, \texttt{authgrant}, P_c, Authcode)$.

On receiving $(sid, \texttt{authgrant}, P_c, Authcode)$ over $sid^{\texttt{CS}}_{\text{SC}}$, P_s checks it against the recorded information. If the check succeeds then it generates a random $AccessToken$ and sends $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\texttt{CS}}_{\text{SC}}$. It also outputs $(sid, KeyCS, AccessToken)$ to the environment.

On receiving $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\texttt{CS}}_{\text{SC}}$, P_c outputs $(sid, KeyCS, AccessToken)$ to the environment.

Send: This is same as in Fig 4.

Notes: 1. The User agent may check if the rediection URI of P_c is same as the URI P_c it used in SSL session $sid^{\texttt{UC1}}_{\text{SSL}}$, and if that session is still alive, it can use that same session instead of a new $sid^{\texttt{UC2}}_{\text{SSL}}$.

Figure 5: OAuth v2 Authorization Grant Flow

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using Authenticated E-mail and HTTPS Redirection

Participants: An arbitrary User Agent P_u, a Service Provider P_s, a Consumer P_c, and Adversary $\mathcal{A}$. The Service Provider and Consumer are assumed to have SSL supporting URIs with PKI certificates.

Initiate: On receiving an input $(sid, \texttt{initiate}, P_u, P_c)$ from the environment, P_u initiates an **https** session with session id $sid_{\text{SSL}}^{\texttt{UC1}}$ with the URI of P_c and query parameters $(sid, \texttt{params-req}, P_u, P_c)$.

On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over https $sid_{\text{SSL}}^{\texttt{UC1}}$ which it outputs to the environment, P_c obtains the input $(sid, \texttt{params}, P_s)$ from the environement. Then P_c responds with $(sid, \text{uri:}P_s, \texttt{client-params})$ to P_u using $\mathcal{F}_{\text{AUTH}}(P_c)$, Since P_s is a globally known entity, the Consumer P_C can obtain a valid https URI of P_s. The query parameters called `client-params` includes the https redirection URI of P_c itself, and other credentials of P_c. The user agent P_u on receiving this authenticated message from P_c initiates an SSL session with P_s (using the supplied uri of P_s) with session id $sid_{\text{SSL}}^{\texttt{UP}}$ and query parameter $(sid, \texttt{client-params})$.[a]

On receiving this message, P_s responds to P_u over $sid_{\text{SSL}}^{\texttt{UP}}$, with a login-password form, and the rest of the implementation is same as in Fig 5

Notes:

1. The same notes as in Fig 5 apply here.
2. The ideal functionality $\mathcal{F}_{\text{AUTH}}$ is defined in the appendix. It essentially, delivers a message only if the sender is P_c, thus guaranteeing the receiver that the message received came from P_c. This, for example, can be the case where the User has a trusted email server, and the e-mail received has a certified signature of P_c.
3. Note that session $sid_{\text{SSL}}^{\texttt{UC1}}$ need not be an SSL session, but $sid_{\text{SSL}}^{\texttt{UC2}}$ must be an SSL session.

[a] technically, the User must click on the supplied link, but we will assume the worst case that the user always clicks on the link.

Figure 6: OAuth v2 Authorization Grant Flow in Email Settings

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using Bulletin Board and HTTPS Redirection

Participants: An arbitrary User Agent P_u, a Service Provider P_s, a Consumer P_c, and Adversary $\mathcal{A}$. The Service Provider and Consumer are assumed to have SSL supporting URIs with PKI certificates.

Initiate: On receiving an input $(sid, \texttt{initiate}, P_u, P_c)$ from the environment, P_u initiates an **https** session with session id $sid^{\texttt{UC1}}_{\text{SSL}}$ with the URI of P_c and query parameters $(sid, \texttt{params-req}, P_u, P_c)$.

On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over https $sid^{\texttt{UC1}}_{\text{SSL}}$ which it outputs to the environment, P_c obtains the input $(sid, \texttt{params}, P_s)$ from the environement. Then P_c responds with $(sid, \text{uri:}P_s, \texttt{client-params})$ to P_u using an un-authenticated and unencrypted channel. Since P_s is a globally known entity, the Consumer P_C can obtain a valid https URI of P_s. The query parameters called `client-params` includes the https redirection URI of P_c itself, and other credentials of P_c. The user agent P_u on receiving this un-authenticated message from P_c, initiates an SSL session with P_s (using the supplied uri P_s) with session id $sid^{\texttt{UP}}_{\text{SSL}}$ and query parameter $(sid, \texttt{client-params})$.[a]

On receiving this message, P_s first checks that the `client-params` has the URI of some P_c along with its Application name which are globally known (e.g. via PKI). Next, P_s responds to P_u over $sid^{\texttt{UP}}_{\text{SSL}}$, with a login-password form *along with the Application name of P_c displayed on the form*, to which the user agent responds by outputting the form along with P_s's identity to the environment (which in this case is possibly just a human User, along with a PKI certificate checker). We will assume here that the User Agent gets human assist in verifying the the P_c displayed on the form is the same as the P_c initially supplied to the User-agent in the `initiate` call. [b]

The Environment (or the human) responds with a *userid* and password pw, which the User agent forwards to P_s using $sid^{\texttt{UP}}_{\text{SSL}}$. Rest of the implementation is same as in Fig 5

Notes: 1. The same notes as in Fig 6 apply here.

[a] technically, the User must click on the supplied link, but we will assume the worst case that the user always clicks on the link.

[b] More rigorously, this mode would require a different ideal functionality where both P_s and P_c are provided in the `params` output to P_u, instead of P_u providing P_c in the `initiate` call.

Figure 7: OAuth v2 Authorization Grant Flow in Bulletin Board Settings

Ideal Functionality Flow $\mathcal{F}_{\text{OAUTH}^*}$

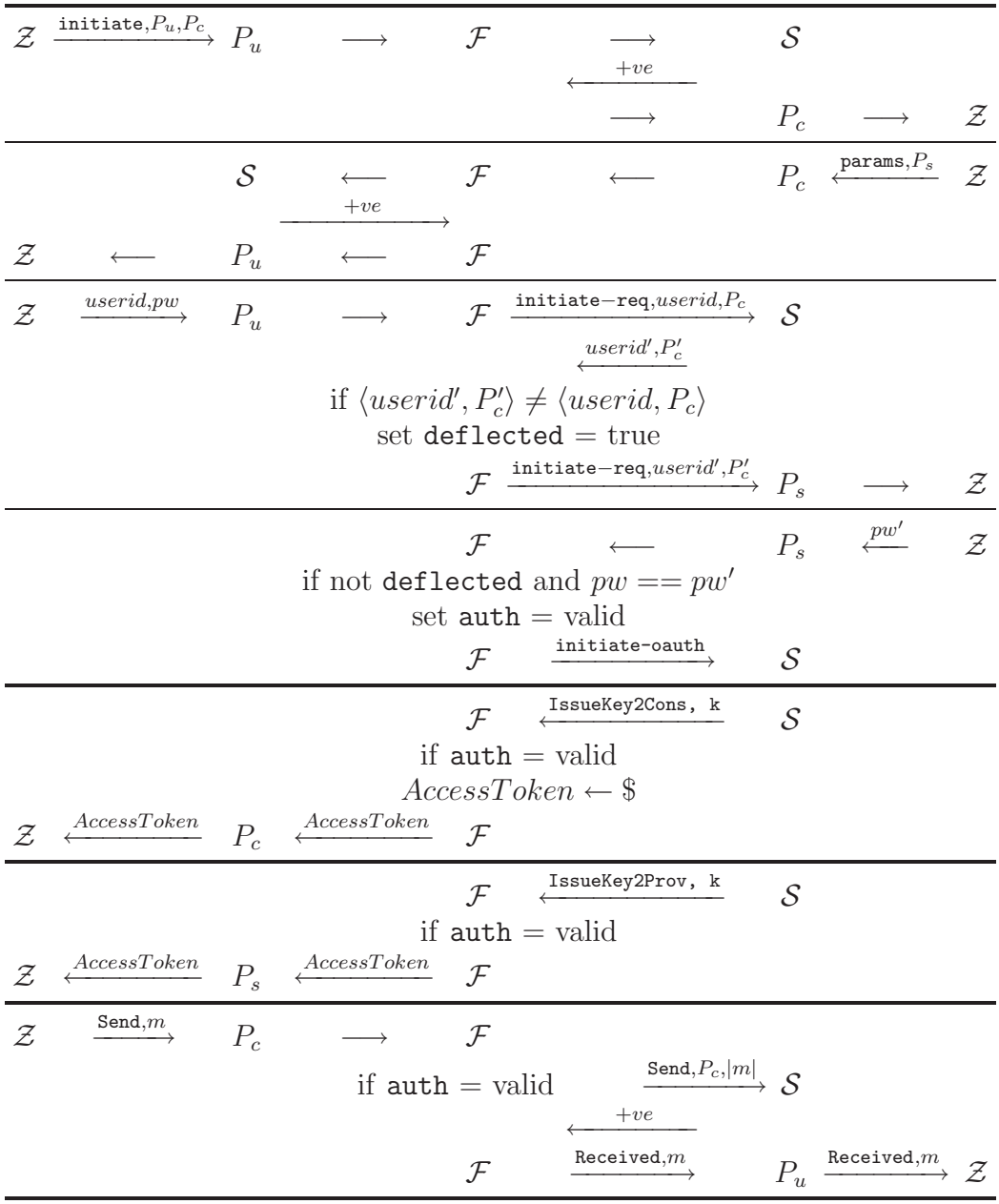

Figure 8: An example flow of the Ideal World Implementation involving $\mathcal{F}_{\text{OAUTH}^*}$, with Adversary mostly responding positively. The symbol $\mathcal{Z}$ stands for the Environment, $\mathcal{F}$ for the ideal functionality, and $\mathcal{S}$ for the Adversary.

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using $\mathcal{F}_{\text{SSL}}$

$\mathcal{Z}$	$\xrightarrow{\texttt{initiate},P_u,P_c}$	P_u	$\xrightarrow{\texttt{Estab-sess},P_u,P_c}$	$\mathcal{F}_{\text{SSL}}^{\text{UC}1}$	$\xrightarrow{+ve}$	$\mathcal{A}$		
					$\xrightarrow{\texttt{Estab-req}}$	P_c		
		$\mathcal{A}$	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UC}1}$	$\xleftarrow{\texttt{Estab-sess}}$	P_c		
		P_u	$\xrightarrow{\texttt{Send},\langle\texttt{par-req}\rangle}$	$\mathcal{F}_{\text{SSL}}^{\text{UC}1}$	$\longrightarrow$	P_c	$\xrightarrow{\texttt{params-req}}$	$\mathcal{Z}$
$\mathcal{Z}$	$\xleftarrow{P_s}$	P_u	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UC}1}$	$\xleftarrow{\texttt{Send}}$	P_c	$\xleftarrow{\texttt{params},P_s}$	$\mathcal{Z}$
$\mathcal{Z}$	$\xrightarrow{\texttt{cred},userid,pw}$	P_u	$\xrightarrow{\texttt{Estab-sess},P_u,P_s}$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\xrightarrow{+ve}$	$\mathcal{A}$		
					$\xrightarrow{\texttt{Estab-req}}$	P_s	$\longrightarrow$	$\mathcal{Z}$
		$\mathcal{A}$	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\longleftarrow$	P_s	$\xleftarrow{\texttt{Estab-sess}}$	$\mathcal{Z}$
		P_u	$\xrightarrow{\texttt{Send},\langle userid,pw,P_c\rangle}$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\longrightarrow$	P_s	$\xrightarrow{userid,P_c}$	$\mathcal{Z}$
						P_s	$\xleftarrow{\texttt{password},pw'}$	$\mathcal{Z}$
						if $pw == pw'$		
		P_u	$\xleftarrow{Authcode}$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\xleftarrow{\texttt{Send},Authcode}$	$AuthCode \leftarrow \$$		
		P_u	$\xrightarrow{\texttt{Estab-sess},P_u,P_c}$	$\mathcal{F}_{\text{SSL}}^{\text{UC}2}$	$\xrightarrow{+ve}$	$\mathcal{A}$		
					$\xrightarrow{\texttt{Estab-req}}$	P_c	$\longrightarrow$	$\mathcal{Z}$
		$\mathcal{A}$	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UC}2}$	$\longleftarrow$	P_c	$\xleftarrow{\texttt{Estab-sess}}$	$\mathcal{Z}$
		P_u	$\xrightarrow{\texttt{Send},AuthCode}$	$\mathcal{F}_{\text{SSL}}^{\text{UC}2}$	$\longrightarrow$	P_c		
		P_c	$\xrightarrow{\texttt{Estab-sess},P_c,P_s}$	$\mathcal{F}_{\text{SC}}$	$\xleftarrow{\texttt{Estab-sess},P_s,P_c}$	P_s		
		P_c	$\xrightarrow{\texttt{Send},Authcode}$	$\mathcal{F}_{\text{SC}}$	$\xrightarrow{Authcode}$	P_s		
						if $Authcode$ same		
$\mathcal{Z}$	$\xleftarrow{AccessToken}$	P_c	$\xleftarrow{AccessToken}$	$\mathcal{F}_{\text{SC}}$	$\xleftarrow{\texttt{Send},AccessToken}$	$AccessToken \leftarrow \$$	$\xrightarrow{AccessToken}$	$\mathcal{Z}$
$\mathcal{Z}$	$\xrightarrow{\texttt{Send},m}$	P_c						
if AccessToken $\neq \perp$			$\xrightarrow{\texttt{Send},m}$	$\mathcal{F}_{\text{SSL}}^{\text{UC}2}$	$\xrightarrow{\texttt{Received},m}$	P_u	$\xrightarrow{\texttt{Received},m}$	$\mathcal{Z}$

Figure 9: An example flow of the Real World Realization of $\mathcal{F}_{\text{OAUTH}^*}$ using $\mathcal{F}_{\text{SSL}}$ and $\mathcal{F}_{\text{SC}}$, with Adversary mostly responding positively.

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 10: 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 11: 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 12: 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 13: The message Authentication functionality $\mathcal{F}$-auth