arXiv:1704.08539v1 [cs.CR] 27 Apr 2017

The Web SSO Standard *OpenID Connect*: In-Depth Formal Security Analysis and Security Guidelines

Daniel Fett, Ralf Küsters, and Guido Schmitz
University of Stuttgart, Germany
Email: {daniel.fett,ralf.kuesters,guido.schmitz}@sec.uni-stuttgart.de

***Abstract*—Web-based single sign-on (SSO) services such as *Google Sign-In* and *Log In with Paypal* are based on the *OpenID Connect* protocol. This protocol enables so-called relying parties to delegate user authentication to so-called identity providers. OpenID Connect is one of the newest and most widely deployed single sign-on protocols on the web. Despite its importance, it has not received much attention from security researchers so far, and in particular, has not undergone any rigorous security analysis.**

In this paper, we carry out the first in-depth security analysis of OpenID Connect. To this end, we use a comprehensive generic model of the web to develop a detailed formal model of OpenID Connect. Based on this model, we then precisely formalize and prove central security properties for OpenID Connect, including authentication, authorization, and session integrity properties.

In our modeling of OpenID Connect, we employ security measures in order to avoid attacks on OpenID Connect that have been discovered previously and new attack variants that we document for the first time in this paper. Based on these security measures, we propose security guidelines for implementors of OpenID Connect. Our formal analysis demonstrates that these guidelines are in fact effective and sufficient.

I. Introduction

OpenID Connect is a protocol for delegated authentication in the web: A user can log into a relying party (RP) by authenticating herself at a so-called identity provider (IdP). For example, a user may sign into the website tripadvisor.com using her Google account.

Although the names might suggest otherwise, OpenID Connect (or *OIDC* for short) is not based on the older OpenID protocol. Instead, it builds upon the OAuth 2.0 framework, which defines a protocol for delegated *authorization* (e.g., a user may grant a third party website access to her resources at Facebook). While OAuth 2.0 was not designed to provide *authentication*, it has often been used for this purpose as well, leading to several severe security flaws in the past [12], [45].

OIDC was created not only to retrofit authentication into OAuth 2.0 by using cryptographically secured tokens and a precisely defined method for user authentication, but also to enable additional important features. For example, using the *Discovery* extension, RPs can automatically identify the IdP that is responsible for a given identity. With the *Dynamic Client Registration* extension, RPs do not need a manual set-up process to work with a specific IdP, but can instead register themselves at the IdP on the fly.

Created by the OpenID Foundation and standardized only in November 2014, OIDC is already very widely used. Among others, it is used and supported by Google, Amazon, Paypal, Salesforce, Oracle, Microsoft, Symantec, Verizon, Deutsche Telekom, PingIdentity, RSA Security, VMWare, and IBM. Many corporate and end-user single sign-on solutions are based on OIDC, for example, well-known services such as *Google Sign-In* and *Log In with Paypal*.

Despite its wide use, OpenID Connect has not received much attention from security researchers so far (in contrast to OpenID and OAuth 2.0). In particular, there have been no formal analysis efforts for OpenID Connect until now. In fact, the only previous works on the security of OpenID Connect are a large-scale study of deployments of Google's implementation of OIDC performed by Li and Mitchell [36] and an informal evaluation by Mainka et al. [38].

In this work, we aim to fill the gap and formally verify the security of OpenID Connect.

Contributions of this Paper. We provide the first in-depth formal security analysis of OpenID Connect. Based on a comprehensive formal web model and strong attacker models, we analyze the security of all flows available in the OIDC standard, including many of the optional features of OIDC and the important Discovery and Dynamic Client Registration extensions. More specifically, our contributions are as follows.

Attacks on OIDC and Security Guidelines. We first compile an overview of attacks on OIDC, common pitfalls, and their respective mitigations. Most of these attacks were documented before, but we point out new attack variants and aspects.

Starting from these attacks and pitfalls, we then derive security guidelines for implementors of OIDC. Our guidelines are backed-up by our formal security analysis, showing that the mitigations that we propose are in fact effective and sufficient.

Formal model of OIDC. Our formal analysis of OIDC is based on the expressive Dolev-Yao style model of the web infrastructure (FKS model) proposed by Fett, Küsters, and Schmitz [19]. This web model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for instance, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. It is the most comprehensive web model to date. Among others, HTTP(S) requests and responses, including several headers, such as cookie, location, referer, authorization, strict transport security (STS), and origin headers, are modeled. The model of web browsers captures the concepts of windows, documents, and iframes, including the complex navigation rules, as well as modern technologies, such as web storage, web messaging (via postMessage), and referrer policies. JavaScript is modeled in an abstract way by so-called *scripts* which can be sent around and, among others, can create iframes, access other windows, and initiate XMLHttpRequests. Browsers may be corrupted dynamically by the adversary.

The FKS model has already been used to analyze the security of the BrowserID single sign-on system [19], [21], the security and privacy of the SPRESSO SSO system [22], and the security of OAuth 2.0 [23], each time uncovering new and severe attacks that have been missed by previous analysis attempts.

Using the generic FKS model, we build a formal model of OIDC, closely following the standard. We employ the defenses and mitigations discussed earlier in order to create a model with state-of-the-art security features in place. Our model includes RPs and IdPs that (simultaneously) support all modes of OIDC and can be dynamically corrupted by the adversary.

Formalization of security properties. Based on this model of OIDC, we formalize four main security properties of OIDC: authentication, authorization, session integrity for authentication, and session integrity for authorization. We also formalize further OIDC specific properties.

Proof of Security for OpenID Connect. Using the model and the formalized security properties, we then show, by a manual yet detailed proof, that OIDC in fact satisfies the security properties. This is the first proof of security of OIDC. Being based on an expressive and comprehensive formal model of the web, including a strong attacker model, as well as on a modeling of OpenID Connect which closely follows the standard, our security analysis covers a wide range of attacks.

Structure of this Paper. We provide an informal description of OIDC in Section II. Attacks and security guidelines are discussed in Section III. In Section IV, we briefly recall the FKS model. The model and analysis of OIDC are then presented in Section V. Related work is discussed in Section VI. We conclude in Section VII. All details of our work, including the proofs, are provided in the appendix.

II. OpenID Connect

The OpenID Connect protocol allows users to authenticate to RPs using their existing account at an IdP.[1] (Typically, this is an email account at the IdP.) OIDC was defined by the OpenID Foundation in a *Core* document [42] and in extension documents (e.g., [41], [43]). Supporting technologies were standardized at the IETF, e.g., [32], [33]. (Recall that OpenID Connect is not to be confused with the older OpenID standards, which are very different to OpenID Connect.)

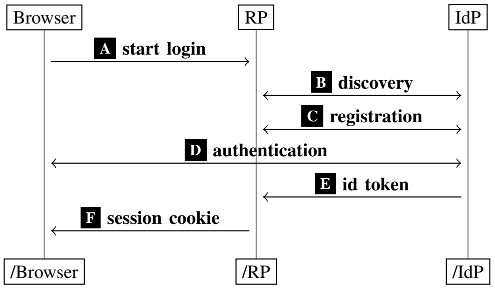

Figure 1. OpenID Connect — high level overview.

Central to OIDC is a cryptographically signed document, the *id token*. It is created by the user's IdP and serves as a one-time proof of the user's identity to the RP.

A high-level overview of OIDC is given in Figure 1. First, the user requests to be logged in at some RP and provides her email address $\boxed{\text{A}}$. RP now retrieves operational information (e.g., some URLs) for the remaining protocol flow (*discovery*, $\boxed{\text{B}}$) and registers itself at the IdP $\boxed{\text{C}}$. The user is then redirected to the IdP, where she authenticates herself $\boxed{\text{D}}$ (e.g., using a password). The IdP issues an id token to RP $\boxed{\text{E}}$, which RP can then verify to ensure itself of the user's identity. (The way of how the IdP sends the id token to the RP is subject to the different modes of OIDC, which are described in detail later in this section. In short, the id token is either relayed via the user's browser or it is fetched by the RP from the IdP directly.) The id token includes an identifier for the IdP (the *issuer*),[2] a user identifier (unique at the respective IdP), and is signed by the IdP. The RP uses the issuer identifier and the user identifier to determine the user's identity. Finally, the RP may set a session cookie in the user's browser which allows the user to access the services of RP $\boxed{\text{F}}$.

Before we explain the modes of operation of OIDC, we first present some basic concepts used in OIDC. At the end of this section, we discuss the relationship of OIDC to OAuth 2.0.

A. Basic Concepts

We have seen above that id tokens are essential to OIDC. Also, to allow users to use any IdP to authenticate to any RP, the RP needs to *discover* some information about the IdP. Additionally, the IdP and the RP need to establish some sort of relationship between each other. The process to establish such a relationship is called *registration*. Both, discovery and registration, can be either a manual task or a fully automatic process. Further, OIDC allows users to *authorize* an RP to access user's data at IdP on the user's behalf. All of these concepts are described in the following.

Authentication and ID Tokens. The goal of OIDC is to *authenticate* a user to an RP, i.e., the RP gets assured of the identity of the user interacting with the RP. This assurance is based on id tokens. As briefly mentioned before, an id token is a document signed by the IdP. It contains several *claims*,

[1] Note that the OIDC standard also uses the terms *client* for RP and *OpenID provider (OP)* for the IdP. We here use the more common terms RP and IdP.

[2] The issuer identifier of an IdP is an HTTPS URL without any query or fragment components.

i.e., information about the user and further meta data. More precisely, an id token contains a user identifier (unique at the respective IdP) and the issuer identifier of the IdP. Both identifiers in combination serve as a global user identifier for authentication. Also, every id token contains an identifier for the RP at the IdP, which is assigned during registration (see below). The id token may also contain a nonce chosen by the RP during the authentication flow as well as an expiration timestamp and a timestamp of the user's authentication at the IdP to prevent replay attacks. Further, an id token may contain information about the particular method of authentication and other claims, such as data about the user and a hash of some data sent outside of the id token.

When an RP validates an id token, it checks in particular whether the signature of the token is correct (we will explain below how RP obtains the public key of the IdP), the issuer identifier is the one of the currently used IdP, the id token is issued for this RP, the nonce is the one RP has chosen during this login flow, and the token has not expired yet. If the id token is valid, the RP trusts the claims contained in the id token and is confident in the user's identity.

Discovery and Registration. The OIDC protocol is heavily based on redirection of the user's browser: An RP redirects the user's browser to some IdP and vice-versa. Hence, both parties, the RP and the IdP, need some information about the respective URLs (so-called *endpoints*) pointing to each other. Also, the RP needs a public key of the IdP to verify the signature of id tokens. Further, an RP can contact the IdP directly to exchange protocol information. This exchange may include authentication of the RP at the IdP.

More specifically, an RP and an IdP need to exchange the following information: (1) a URL where the user can authenticate to the IdP (*authorization endpoint*), (2) one or more URLs at RP where the user's browser can be redirected to by the IdP after authentication (*redirection endpoint*), (3) a URL where the RP can contact the IdP in order to retrieve an id token (*token endpoint*), (4) the issuer identifier of the IdP, (5) the public key of the IdP to verify the id token's signature, (6) an identifier of the RP at IdP (*client id*), and optionally (7) a secret used by RP to authenticate itself to the token endpoint (*client secret*). (Recall that *client* is another term for RP, and in particular does not refer to the browser.)

This information can be exchanged manually by the administrator of the RP and the administrator of the IdP, but OIDC also allows one to completely automate the discovery of IdPs [43] and dynamically register RPs at an IdP [41].

During the automated discovery, the RP first determines which IdP is responsible for the email address provided by the user who wants to log in using the WebFinger protocol [33]. As a result, the RP learns the issuer identifier of the IdP and can retrieve the URLs of the authorization endpoint and the token endpoint from the IdP. Furthermore, the RP receives a URL where it can retrieve the public key to verify the signature of the id token (*JWKS URI*), and a URL where the RP can register itself at the IdP (*client registration endpoint*).

If the RP has not registered itself at this IdP before, it starts the registration ad-hoc at the client registration endpoint: The RP sends its redirection endpoint URLs to the IdP and receives a new client id and (optionally) a client secret in return.

Authorization and Access Tokens. OIDC allows users to authorize RPs to access the user's data stored at IdPs or act on the user's behalf at IdPs. For example, a photo printing service (the RP) might access or manage the user's photos on Google Drive (the IdP). For authorization, the RP receives a so-called *access token* (besides the id token). Access tokens follow the concept of so-called *bearer tokens*, i.e., they are used as the only authentication component in requests from an RP to an IdP. In our example, the photo printing service would have to add the access token to each HTTP request to Google Drive.

B. Modes

OIDC defines three modes: the *authorization code mode*, the *implicit mode*, and the *hybrid mode*. While in the authorization code mode, the id token is retrieved by an RP from an IdP in direct server-to-server communication (back channel), in the implicit mode, the id token is relayed from an IdP to an RP via the user's browser (front channel). The hybrid mode is a combination of both modes and allows id tokens to be exchanged via the front and the back channel at the same time.

We now provide a detailed description of all three modes.

Authorization Code Mode. In this mode, an RP redirects the user's browser to an IdP. At the IdP, the user authenticates and then the IdP issues a so-called *authorization code* to the RP. The RP now uses this code to obtain an id token from the IdP.

Step-by-Step Protocol Flow. The protocol flow is depicted in Figure 2. First, the user starts the login process by entering her email address[3] in her browser (at some web page of an RP), which sends the email address to the RP in $\boxed{1}$.

Now, the RP uses the OIDC discovery extension [43] to gather information about the IdP: As the first step (in this extension), the RP uses the WebFinger mechanism [33] to discover information about which IdP is responsible for this email address. For this discovery, the RP contacts the server of the email domain in $\boxed{2}$ (in the figure, the server of the user's email domain is depicted as the same party as the IdP). The result of the WebFinger request in $\boxed{3}$ contains the issuer identifier of the IdP (which is also a URL). With this information, the RP can continue the discovery by requesting the OIDC configuration from the IdP in $\boxed{4}$ and $\boxed{5}$. This configuration contains meta data about the IdP, including all endpoints at the IdP and a URL where the RP can retrieve the public key of the IdP (used to later verify the id token's signature). If the RP does not know this public key yet, the RP retrieves the key (Steps $\boxed{6}$ and $\boxed{7}$). This concludes the OIDC discovery in this login flow.

Next, if the RP is not registered at the IdP yet, the RP starts the OIDC dynamic client registration extension [41]: In Step $\boxed{8}$ the RP contacts the IdP and provides its redirect URIs.

[3] Note that OIDC also allows other types of user ids, such as personal URLs.

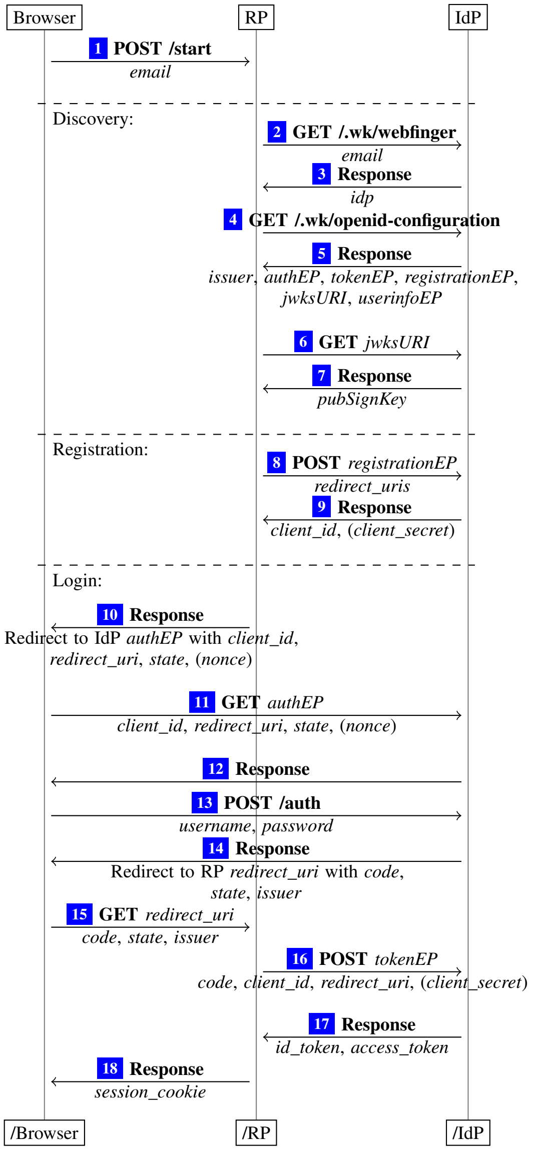

Figure 2. OpenID Connect authorization code mode. Note that data depicted below the arrows is either transferred in URI parameters, HTTP headers, or POST bodies.

In return, the IdP issues a client id and (optionally) a client secret to the RP in Step 9. This concludes the registration.

Now, the core part of the OIDC protocol starts: the RP redirects the user's browser to the IdP in 10. This redirect contains the information that the authorization code mode is used. Also, this redirect contains the client id of the RP, a redirect URI, and a *state* value, which serves as a Cross-Site Request Forgery (CSRF) token when the browser is later redirected back to the RP. The redirect may also optionally include a nonce, which will be included in the id token issued later in this flow. This data is sent to the IdP by the browser 11. The user authenticates to the IdP 12, 13, and the IdP redirects the user's browser back to the RP in 14 and 15 (using the redirect URI from the request in 11). This redirect contains an authorization code, the *state* value as received in 10, and the issuer identifier.[4] If the *state* value and the issuer identifier are correct, the RP contacts the IdP in 16 at the token endpoint with the received authorization code, its client id, its client secret (if any), and the redirect URI used to obtain the authorization code. If these values are correct, the IdP responds with a fresh access token and an id token to the RP in 17. If the id token is valid, then the RP considers the user to be logged in (under the identifier composed from the user id in the id token and the issuer identifier). Hence, the RP may set a session cookie at the user's browser in 18.

Implicit Mode. The implicit mode (depicted in Figure 3 in Appendix A) is similar to the authorization code mode, but instead of providing an authorization code, the IdP issues an id token right away to the RP (via the user's browser) when the user authenticates to the IdP. Hence, the Steps 1–13 of the authorization code mode (Figure 2) are the same. After these steps, the IdP redirects the user's browser to the redirection endpoint at the RP, providing an id token, (optionally) an access token, the *state* value, and the issuer identifier. These values are not provided as a URL parameter but in the URL fragment instead. Hence, the browser does not send them to the RP at first. Instead, the RP has to provide a JavaScript that retrieves these values from the fragment and sends them to the RP. If the id token is valid, the issuer is correct, and the *state* matches the one previously chosen by the RP, the RP considers the user to be logged in and issues a session cookie.

Hybrid Mode. The hybrid mode (depicted in Figure 4 in Appendix A) is a combination of the authorization code mode and the implicit mode: First, this mode works like the implicit mode, but when IdP redirects the browser back to RP, the IdP issues an authorization code, and either an id token or an access token or both.[5] The RP then retrieves these values as in the implicit mode (as they are sent in the fragment like in the implicit mode) and uses the authorization code to obtain a (potentially second) id token and a (potentially second) access token from IdP.

C. Relationship to OAuth 2.0

Technically, OIDC is derived from OAuth 2.0. It goes, however, far beyond what was specified in OAuth 2.0 and introduces many new concepts: OIDC defines a method for authentication (while retaining the option for authorization) using a new type of tokens, the id token. Some messages and tokens in OIDC can be cryptographically signed or encrypted while OAuth 2.0 does neither use signing nor encryption. The new hybrid flow combines features of the implicit mode and the authorization code mode. Importantly, with ad-hoc discovery and dynamic registration, OIDC standardizes and automates a process that is completely out of the scope of OAuth 2.0.

[4] The issuer identifier will be included in this message in an upcoming revision of OIDC to mitigate the IdP Mix-Up attack, see Section III-A.

[5] The choice of the IdP to issue either an id token or an access token or both depends on the IdP's configuration and the request in Step 3 in Figure 4.

These new features and their interplay potentially introduce new security flaws. It is therefore not sufficient to analyze the security of OAuth 2.0 to derive any guarantees for OIDC. OIDC rather requires a new security analysis. (See Section V for a more detailed discussion. In Section III we describe attacks that cannot be applied to OAuth 2.0.)

III. Attacks and Security Guidelines

In this section, we present a concise overview of known attacks on OIDC and present additions that have not been documented so far. We also summarize mitigations and implementation guidelines that have to be implemented to avoid these attacks.

The main focus of this work is to prove central security properties of OIDC, by which these mitigations and implementation guidelines are backed up. Moreover, further (potentially unknown types of) attacks on OIDC that can be captured by our security analysis are ruled out as well.

The rest of the section is structured as follows: we first present the attacks, mitigations and guidelines, then point out differences to OAuth 2.0, and finally conclude with a brief discussion.

A. Attacks, Mitigations, and Guidelines

(Mitigations and guidelines are presented along with every class of attack.)

IdP Mix-Up Attacks. In two previously reported attacks [23], [38], the aim was to confuse the RP about the identity of the IdP. In both attacks, the user was tricked into using an *honest* IdP to authenticate to an *honest* RP, while the RP is made to believe that the user authenticated to the attacker. The RP therefore, after successful user authentication, tries to use the authorization code or access token at the attacker, which then can impersonate the user or access the user's data at the IdP. We present a detailed description of an application of the IdP Mix-Up attack to OpenID Connect in Appendix A.

The IETF OAuth Working Group drafted a proposal for a mitigation technique [31] that is based on a proposal in [23] and that also applies to OpenID Connect. The proposal is that the IdP puts its identity into the response from the authorization endpoint. (This is already included in our description of OIDC above, see the issuer in Step $\boxed{14}$ in Figure 2.) The RP can then check that the user authenticated to the expected IdP.

Attacks on the State Parameter. The *state* parameter is used in OIDC to protect against attacks on session integrity, i.e., attacks in which an attacker forces a user to be logged in at some RP (under the attacker's account). Such attacks can arise from session swapping or CSRF vulnerabilities.

OIDC recommends the use of the *state* parameter. It should contain a nonce that is bound to the user's session. Attacks that can result from omitting or incorrectly using *state* were described in the context of OAuth 2.0 in [8], [35], [37], [44].

The nonce for the *state* value should be chosen freshly for *each login attempt* to prevent an attack described in [23] (Section 5.1) where the same state value is used first in a user-initiated login flow with a malicious IdP and then in a login flow with an honest IdP (forcefully initiated by the attacker with the attacker's account and the user's browser).

Code/Token/State Leakage. Care should be taken that a value of *state* or an authorization code is not inadvertently sent to an untrusted third party through the *Referer* header. The *state* and the authorization code parameters are part of the redirection endpoint URI (at the RP), the *state* parameter is also part of the authorization endpoint URI (at the IdP). If, on either of these two pages, a user clicks on a link to an external page, or if one of these pages embeds external resources (images, scripts, etc.), then the third party will receive the full URI of the endpoint, including these parameters, in the Referer header that is automatically sent by the browser.

Documents delivered at the respective endpoints should therefore be vetted carefully for links to external pages and resources. In modern browsers, *referrer policies* [18] can be used to suppress the Referer header. As a second line of defense, both parameters should be made single-use, i.e., *state* should expire after it has been used at the redirection endpoint and authorization code after it has been redeemed at IdP.

In a related attack, an attacker that has access to logfiles or browsing histories (e.g., through malicious browser extensions) can steal authentication codes, access tokens, id tokens, or *state* values and re-use these to impersonate a user or to break session integrity. A subset of these attacks was dubbed *Cut-and-Paste Attacks* by the IETF OAuth working group [31].

There are drafts for RFCs that tackle specific aspects of these leakage attacks, e.g., [13] which discusses binding the *state* parameter to the browser instance, and [30] which discusses binding the access token to a TLS session. Since these mitigations are still very early IETF drafts, subject to change, and not easy to implement in the majority of the existing OIDC implementations, we did not model them.

In our analysis, we assume that implementations keep logfiles and browsing histories (of honest browsers) secret and employ referrer policies as described above.

Naïve RP Session Integrity Attack. So far, we have assumed that after Step $\boxed{10}$ (Figure 2), the RP remembers the user's choice (which IdP is used) in a session; more precisely, the user's choice is stored in RP's session data. This way, in Step $\boxed{15}$, the RP looks up the user's selected IdP in the session data. In [23], this is called *explicit user intention tracking*.

There is, however, an alternative to storing the IdP in the session. As pointed out by [23], some implementations put the identity of the IdP into the *redirect_uri* (cf. Step $\boxed{10}$), e.g., by appending it as the last part of the path or in a parameter. Then, in Step $\boxed{15}$, the RP can retrieve this information from the URI. This is called *naïve user intention tracking*.

RPs that use naïve user intention tracking are susceptible to the naïve RP session integrity attack described in [23]: An attacker obtains an authorization code, id token, or access token for his own account at an honest IdP (HIdP). He then waits for a user that wants to log in at some RP using the attacker's

IdP (AIdP) such that AIdP obtains a valid *state* for this RP. AIdP then redirects the user to the redirection endpoint URI of RP using the identity of HIdP plus the obtained *state* value and code or (id) token. Since the RP cannot see that the user originally wanted to log in using AIdP instead of HIdP, the user will now be logged in under the attacker's identity.

Therefore, an RP should always use sessions to store the user's chosen IdP (explicit user intention tracking), which, as mentioned, is also what we do in our formal OIDC model.

307 Redirect Attack. Although OIDC explicitly allows for any redirection method to be used for the redirection in Step $\boxed{14}$ of Figure 2, IdPs should not use an HTTP 307 status code for redirection. Otherwise, credentials entered by the user at an IdP will be repeated by the browser in the request to RP (Step $\boxed{15}$ of Figure 2), and hence, malicious RPs would learn these credentials and could impersonate the user at the IdP. This attack was presented in [23]. In our model, we exclusively use the 303 status code, which prevents re-sending of form data.

Injection Attacks. It is well known that Cross-Site Scripting (XSS) and SQL Injection attacks on RPs or IdPs can lead to theft of access tokens, id tokens, and authorization codes (see, for example, [8], [27], [37], [38], [44]). XSS attacks can, for example, give an attacker access to session ids. Besides using proper escaping (and Content Security Policies [46] as a second line of defense), OIDC endpoints should therefore be put on domains separate from other, potentially more vulnerable, web pages on IdPs and RPs.[6] (See *Third-Party Resources* below for another motivation for this separation.)

In OIDC implementations, data that can come from untrusted sources (e.g., client ids, user attributes, *state* and *nonce* values, redirection URIs) must be treated as such: For example, a malicious IdP might try to inject user attributes containing malicious JavaScript to the RP. If the RP displays this data without applying proper escaping, the JavaScript is executed.

We emphasize that in a similar manner, attackers can try to inject additional parameters into URIs by appending them to existing parameter values, e.g., the *state*. Since data is often passed around in OIDC, proper escaping of such parameters can be overlooked easily.

As a result of such parameter injection attacks or independently, parameter pollution attacks can be a threat for OIDC implementations. In these attacks, an attacker introduces duplicate parameters into URLs (see, e.g., [6]). For example, a simple parameter pollution attack could be launched as follows: A malicious RP could redirect a user to an honest IdP, using a client id of some honest RP but appending two redirection URI parameters, one pointing to the honest RP and one pointing to the attacker's RP. Now, if the IdP checks the first redirection URI parameter, but afterwards redirects to the URI in the second parameter, the attacker learns authentication data that belongs to the honest RP and can impersonate the user.

Mitigations against all these kinds of injection attacks are well known: implementations have to vet incoming data carefully, and properly escape any output data. In our model, we assume that these mitigations are implemented.

CSRF Attacks and Third-Party Login Initiation. Some endpoints need protection against CSRF in addition to the protection that the *state* parameter provides, e.g., by checking the origin header. Our analysis shows that the RP only needs to protect the URI on which the login flow is started (otherwise, an attacker could start a login flow using his own identity in a user's browser) and for the IdP to protect the URI where the user submits her credentials (otherwise, an attacker could submit his credentials instead).

In the OIDC Core standard [42], a so-called *login initiation endpoint* is described which allows a third party to start a login flow by redirecting a user to this endpoint, passing the identity of an IdP in the request. The RP will then start a login flow at the given IdP. Members of the OIDC foundation confirmed to us that this endpoint is essentially an intentional CSRF protection bypass. We therefore recommend login initiation endpoints not to be implemented (they are not a mandatory feature), or to require explicit confirmation by the user.

Server-Side Request Forgery (SSRF). SSRF attacks can arise when an attacker can instruct a server to send HTTP(S) requests to other hosts, causing unwanted side-effects or revealing information [40]. For example, if an attacker can instruct a server behind a firewall to send requests to other hosts behind this firewall, the attacker might be able to call services or to scan the internal network (using timing attacks). He might also instruct the server to retrieve very large documents from other sources, thereby creating Denial of Service attacks.

SSRF attacks on OIDC were described for the first time in [38], in the context of the OIDC Discovery extension: An attacker could set up a malicious discovery service that, when queried by an RP, answers with links to arbitrary, network-internal or external servers (in Step $\boxed{5}$ of Figure 2).

We here, for the first time, point out that not only RPs can be vulnerable to SSRF, but also IdPs. OIDC defines a way to indirectly pass parameters for the authorization request (cf. Step $\boxed{11}$ in Figure 2). To this end, the IdP accepts a new parameter, *request_uri* in the authorization request. This parameter contains a URI from which the IdP retrieves the additional parameters (e.g., *redirect_uri*). The attacker can use this feature to easily mount an SSRF attack against the IdP even without any OIDC extensions: He can put an arbitrary URI in an authorization request causing the IdP to contact this URI.

This new attack vector shows that not only RPs but also IdPs have to protect themselves against SSRF by using appropriate filtering and limiting mechanisms to restrict unwanted requests that originate from a web server (cf. [40]).

SSRF attacks typically depend on an application specific context, such as the structure of and (vulnerable) services in an internal network. In our model, attackers can trigger SSRF requests, but the model does not contain vulnerable

[6] Since scripts on one origin can often access documents on the same origin, origins of the OIDC endpoints should be free from untrusted scripts.

applications/services aside from OIDC. (Our analysis focuses on the security of the OIDC standard itself, rather than on specific applications and services.) Timing and performance properties, while sometimes relevant for SSRF attacks, are also outside of our analysis.

Third-Party Resources. RPs and IdPs that include third-party resources, e.g., tracking or advertisement scripts, subject their users to token theft and other attacks by these third parties. If possible, RPs and IdPs should therefore avoid including third-party resources on any web resources delivered from the same origins[6] as the OIDC endpoints (see also Section V-F). For newer browsers, *subresource integrity* [2] can help to reduce the risks associated with embedding third-party resources. With subresource integrity, websites can instruct supporting web browsers to reject third-party content if this content does not match a specific hash. In our model, we assume that websites do not include untrusted third-party resources.

Transport Layer Security. The security of OIDC depends on the confidentiality and integrity of the transport layer. In other words, RPs and IdPs should use HTTPS. Endpoint URIs that are provided for the end user and that are communicated, e.g., in the discovery phase of the protocol, should only use the `https://` scheme. HTTPS Strict Transport Security and Public Key Pinning can be used to further strengthen the security of the OIDC endpoints. (In our model, we assume that users enter their passwords only over HTTPS web sites because otherwise, any authentication could be broken trivially.)

Session Handling. Sessions are typically identified by a nonce that is stored in the user's browser as a cookie. It is a well known best practice that cookies should make use of the *secure* attribute (i.e., the cookie is only ever used over HTTPS connections) and the *HttpOnly* flag (i.e., the cookie is not accessible by JavaScript). Additionally, after the login, the RP should replace the session id of the user by a freshly chosen nonce in order to prevent session fixation attacks: Otherwise, a network attacker could set a login session cookie that is bound to a known *state* value into the user's browser (see [48]), lure the user into logging in at the corresponding RP, and then use the session cookie to access the user's data at the RP (*session fixation*, see [39]). In our model, RPs use two kinds of sessions: Login sessions (which are valid until just before a user is authenticated at the RP) and service sessions (which signify that a user is already signed in to the RP). For both sessions, the secure and HttpOnly flags are used.

B. Relationship to OAuth 2.0

Many, but not all of the attacks described above can also be applied to OAuth 2.0. The following attacks in particular are only applicable to OIDC: (1) Server-side request forgery attacks are facilitated by the ad-hoc discovery and dynamic registration features. (2) The same features enable new ways to carry out injection attacks. (3) The new OIDC feature third-party login initiation enables new CSRF attacks. (4) Attacks on the id token only apply to OIDC, since there is no such token in OAuth 2.0.

It is interesting to note that on the other hand, some attacks on OAuth 2.0 cannot be applied to OIDC (see [23], [37] for further discussions on these attacks): (1) OIDC setups are less prone to open redirector attacks since placeholders are not allowed in redirection URIs. (2) TLS is mandatory for some messages in OIDC, while it is optional in OAuth 2.0. (3) The nonce value can prevent some replay attacks when the state value is not used or leaks to an attacker.

C. Discussion

In this section, our focus was to provide a concise overview of known attacks on OIDC and present some additions, namely SSRF at IdPs and third-party login initiation, along with mitigations and implementation guidelines. Our formal analysis of OIDC, which is the main focus of our work and is presented in the next sections, shows that the mitigations and implementation guidelines presented above are effective and that we can exclude other, potentially unknown types of attacks.

IV. The FKS Web Model

Our formal security analysis of OIDC is based on the FKS model, a generic Dolev-Yao style web model proposed by Fett et al. in [19]. Here, we only briefly recall this model following the description in [23] (see Appendices B ff. for a full description, and [19], [21], [22] for comparison with other models and discussion of its scope and limitations).

The FKS model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The FKS model defines a general communication model, and, based on it, web systems consisting of web browsers, DNS servers, and web servers as well as web and network attackers.

Communication Model. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes communicate via *events*, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from a "pool" of waiting events and is delivered to one of the processes that listens to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on.

As usual in Dolev-Yao models (see, e.g., [1]), messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures). For example, in the web model, an HTTP request is represented as a term r containing a nonce, an HTTP method, a domain name, a path, URI parameters, request headers, and a message body. For instance, an HTTP request for the URI http://ex.com/show?p=1 is represented as $r := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{ex.com}, \mathtt{/show}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$ where the body and the list of request headers is empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r,k'\rangle,\mathsf{pub}(k_{\text{ex.com}}))$, where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x,\mathsf{pub}(y)),y) = x$. With this, we have that, for example, $\mathsf{dec_a}(\mathsf{enc_a}(\langle r,k'\rangle,\mathsf{pub}(k_{\text{ex.com}})),k_{\text{ex.com}}) \equiv \langle r,k'\rangle$, i.e., these two terms are equivalent w.r.t. the equational theory.

A *(Dolev-Yao) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

The so-called *attacker process* is a Dolev-Yao process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. Attackers can corrupt other parties.

A *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below). We give an annotated example for a script in Algorithm 22 in the appendix. Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

A *system* is a set of processes. A *configuration* of this system consists of the states of all processes in the system, the pool of waiting events, and a sequence of unused nonces. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation step. The transition from one configuration to the next configuration in a run is called a *processing step*. We write, for example, $Q = (S,E,N) \rightarrow (S',E',N')$ to denote the transition from the configuration (S,E,N) to the configuration (S',E',N'), where S and S' are the states of the processes in the system, E and E' are pools of waiting events, and N and N' are sequences of unused nonces.

A *web system* formalizes the web infrastructure and web applications. It contains a system consisting of honest and attacker processes. Honest processes can be web browsers, web servers, or DNS servers. Attackers can be either *web attackers* (who can listen to and send messages from their own addresses only) or *network attackers* (who may listen to and spoof all addresses and therefore are the most powerful attackers). A web system further contains a set of scripts (comprising honest scripts and the attacker script).

In our analysis of OIDC, we consider either one network attacker or a set of web attackers (see Section V). In our OIDC model, we need to specify only the behavior of servers and scripts. These are not defined by the FKS model since they depend on the specific application, unless they become corrupted, in which case they behave like attacker processes and attacker scripts; browsers are specified by the FKS model (see below). The modeling of OIDC servers and scripts is outlined in Section V and with full details provided in Appendices F and G.

Web Browsers. An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions, such as following a link, are modeled as non-deterministic actions of the web browser. User credentials are stored in the initial state of the browser and are given to selected web pages when needed. Besides user credentials, the state of a web browser contains (among others) a tree of windows and documents, cookies, and web storage data (localStorage and sessionStorage).

A *window* inside a browser contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window. Each represents one loaded web page and contains (among others) a script and a list of subwindows (modeling iframes). The script, when triggered by the browser, is provided with all data it has access to, such as a (limited) view on other documents and windows, certain cookies, and web storage data. Scripts then output a command and a new state. This way, scripts can navigate or create windows, send XMLHttpRequests and postMessages, submit forms, set/change cookies and web storage data, and create iframes. Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the relevant web standards.

A browser can output messages on the network of different types, namely DNS and HTTP(S) (including XMLHttpRequests), and it processes the responses. Several HTTP(S) headers are modeled, including, for example, cookie, location, strict transport security (STS), and origin headers. A browser, at any time, can also receive a so-called trigger message upon which the browser non-deterministically choses an action, for instance, to trigger a script in some document. The script now outputs a command, as described above, which is then further processed by the browser. Browsers can also become corrupted, i.e., be taken over by web and network attackers. Once corrupted, a browser behaves like an attacker process.

V. Analysis

We now present our security analysis of OIDC, including a formal model of OIDC, the specifications of central security properties, and our theorem which establishes the security of OIDC in our model.

More precisely, our formal model of OIDC uses the FKS model as a foundation and is derived by closely following the OIDC standards Core, Discovery, and Dynamic Client Registration [41]–[43]. (As mentioned above, the goal in this work is to analyze OIDC itself instead of concrete implementations.) We then formalize the main security properties for OIDC, namely authentication, authorization, session integrity for authentication, and session integrity for authorization. We also formalize secondary security properties that capture important aspects of the security of OIDC, for example, regarding the outcome of the dynamic client registration. We then state and prove our main theorem. Finally, we discuss the relationship of our work to the analysis of OAuth 2.0 presented in [23] and conclude with a discussion of the results.

We refer the reader to Appendices F–I for full details, including definitions, specifications, and proofs. To provide an intuition of the abstraction level, syntax, and concepts that we use for the modeling without reading all details, we extensively annotated Algorithms 17, 20, and 22 in Appendix F.

A. Model

Our model of OIDC includes all features that are commonly found in real-world implementations, for example, all three modes, a detailed model of the Discovery mechanism [43] (including the WebFinger protocol [33]), and Dynamic Client Registration [41] (including dynamic exchange of signing keys). RPs, IdPs, and, as usual in the FKS model, browsers can be corrupted by the adversary dynamically.

We do not model less used features, in particular OIDC logout, self-issued OIDC providers ("personal, self-hosted OPs that issue self-signed ID Tokens", [42]), and ACR/AMR (Authentication Class/Methods Reference) values that can be used to indicate the level of trust in the authentication of the user to the IdP.

Since the FKS model has no notion of time, we overapproximate by never letting tokens, e.g., id tokens, expire. Moreover, we subsume user claims (information about the user that can be retrieved from IdPs) by user identifiers, and hence, in our model users have identities, but no other properties.

We have two versions of our OIDC model, one with a network attacker and one with an unbounded number of web attackers, as explained next. The reason for having two versions is that while the authentication and authorization properties can be proven assuming a network attacker, such an attacker could easily break session integrity. Hence, for session integrity we need to assume web attackers (see the explanations for session integrity in Section V-B).

OIDC Web System with a Network Attacker. We model OIDC as a class of web systems (in the sense of Section IV) which can contain an unbounded finite number of RPs, IdPs, browsers, and one network attacker.

More formally, an *OIDC web system with a network attacker* ($\mathcal{OIDC}^n$) consists of a network attacker, a finite set of web browsers, a finite set of web servers for the RPs, and a finite set of web servers for the IdPs. Recall that in $\mathcal{OIDC}^n$, since we have a network attacker, we do not need to consider web attackers (as the network attacker subsumes all web attackers). All non-attacker parties are initially honest, but can become corrupted dynamically upon receiving a special message and then behave just like a web attacker process.

As already mentioned in Section IV, to model OIDC based on the FKS model, we have to specify the protocol specific behavior only, i.e., the servers for RPs and IdPs as well as the scripts that they use. We start with a description of the servers.

Web Servers. Since RPs and IdPs both are web servers, we developed a generic model for HTTPS server processes for the FKS model. We call these processes *HTTPS server base processes*. Their definition covers decrypting received HTTPS messages and handling HTTP(S) requests to external webservers, including DNS resolution.

RPs and IdPs are derived from this HTTPS server base process. Their models follow the OIDC standard closely and include the mitigations discussed in Section III.

An RP waits for users to start a login flow and then non-deterministically decides which mode to use. If needed, it starts the discovery and dynamic registration phase of the protocol, and finally redirects the user to the IdP for user authentication. Afterwards, it processes the received tokens and uses them according to their type (e.g., with an access token, the RP would retrieve an id token from the IdP). If an id token is received that passes all checks, the user will be logged in. As mentioned briefly in Section III, RPs manage two kinds of sessions: The *login sessions*, which are used only during the user login phase, and *service sessions*.

The IdP provides several endpoints according to its role in the login process, including its OIDC configuration endpoint and endpoints for receiving authentication and token requests.

Scripts. Three scripts (altogether 30 lines of code) can be sent from honest IdPs and RPs to web browsers. The script *script_rp_index* is sent by an RP when the user visits the RP's web site. It starts the login process. The script *script_rp_get_fragment* is sent by an RP during an implicit or hybrid mode flow to retrieve the data from the URI fragment. It extracts the access token, authorization code, and *state* from the fragment part of its own URI and sends this information in the body of a POST request back to the RP. IdP sends the script *script_idp_form* for user authentication at the IdP.

OIDC Web System with Web Attackers. We also consider a class of web systems where the network attacker is replaced by an unbounded finite set of web attackers and a DNS server is introduced. We denote such a system by $\mathcal{OIDC}^w$ and call it an *OIDC web system with web attackers*. Such web systems are used to analyze session integrity, see below.

B. Main Security Properties

Our primary security properties capture authentication, authorization and session integrity for authentication and authorization. We will present these security properties in the following, with full details in Appendix H.

Authentication Property. The most important property for OIDC is the authentication property. In short, it captures that a network attacker (and therefore also web attackers) should be unable to log in as an honest user at an honest RP using an honest IdP.

Before we define this property in more detail, recall that in our modeling, an RP uses two kinds of sessions: login sessions, which are only used for the login flow, and service sessions, which are used after a user/browser was logged in (see Section III-A for details). When a login session has finished successfully (i.e., the RP received a valid id token), the RP uses a fresh nonce as the service session id, stores this id in the session data of the login session, and sends the service session id as a cookie to the browser. In the same step, the RP also stores the issuer, say d, that was used in the login flow and the identity (email address) of the user, say id, as a pair $\langle d, id \rangle$, referred to as a global user identifier in Section II-A.

Now, our authentication property defines that a network attacker should be unable to get hold of a service session id by which the attacker would be considered to be logged in at an honest RP under an identity governed by an honest IdP for an honest user/browser.

In order to define the authentication property formally, we first need to define the precise notion of a service session. In the following, as introduced in Section IV, (S, E, N) denotes a configuration in the run ρ with its components S, a mapping from processes to states of these processes, E, a set of events in the network that are waiting to be delivered to some party, and N, a set of nonces that have not been used yet. By $\mathsf{governor}(id)$ we denote the IdP that is responsible for a given user identity (email address) id, and by $\mathsf{dom}(\mathsf{governor}(id))$, we denote the set of domains that are owned by this IdP. By $S(r).\texttt{sessions}[lsid]$ we denote a data structure in the state of r that contains information about the login session identified by $lsid$. This data structure contains, for example, the identity for which the login session with the id $lsid$ was started and the service session id that was issued after the login session.

We can now define that there is a service session identified by a nonce n for an identity id at some RP r iff there exists a login session (identified by some nonce $lsid$) such that n is the service session associated with this login session, and r has stored that the service session is logged in for the id id using an issuer d (which is some domain of the governor of id).

Definition 1 (Service Sessions). We say that there is a *service session identified by a nonce n for an identity id at some RP r* in a configuration (S, E, N) of a run ρ of an OIDC web system iff there exists some login session id $lsid$ and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ such that $S(r).\texttt{sessions}[lsid][\texttt{loggedInAs}] \equiv \langle d, id \rangle$ and $S(r).\texttt{sessions}[lsid][\texttt{serviceSessionId}] \equiv n$.

By $d_\emptyset(S(\mathsf{attacker}))$ we denote all terms that can be computed (derived in the usual Dolev-Yao style, see Section IV) from the attacker's knowledge in the state S. We can now define that an OIDC web system with a network attacker is secure w.r.t. authentication iff the attacker can never get hold of a service session id (n) that was issued by an honest RP r for an identity id of an honest user (browser) at some honest IdP (governor of id).

Definition 2 (Authentication Property). Let $\mathcal{OIDC}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDC}^n$ is secure w.r.t. authentication* iff for every run ρ of $\mathcal{OIDC}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id)$ being an honest IdP, every service session identified by some nonce n for id at r, we have that n is not derivable from the attackers knowledge in S (i.e., $n \not\in d_\emptyset(S(\mathsf{attacker}))$).

Authorization Property. Intuitively, authorization for OIDC means that a network attacker should not be able to obtain or use a protected resource available to some honest RP at an IdP for some user unless certain parties involved in the authorization process are corrupted. As the access control for such protected resources relies only on access tokens, we require that an attacker does not learn access tokens that would allow him to gain unauthorized access to these resources.

To define the authorization property formally, we need to reason about the state of an honest IdP, say i. In this state, i creates *records* containing data about successful authentications of users at i. Such records are stored in $S(i).\texttt{records}$. One such record, say x, contains the authenticated user's identity in $x[\texttt{subject}]$, two[7] access tokens in $x[\texttt{access\_tokens}]$, and the client id of the RP in $x[\texttt{client\_id}]$.

We can now define the authorization property. It defines that an OIDC web system with a network attacker is secure w.r.t. authorization iff the attacker cannot get hold of an access token that is stored in one of i's records for an identity of an honest user/browser b and an honest RP r.

Definition 3 (Authorization Property). Let $\mathcal{OIDC}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDC}^n$ is secure w.r.t. authorization* iff for every run ρ of $\mathcal{OIDC}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $i \in \mathsf{IdP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ owned by b and $\mathsf{governor}(id) = i$, every nonce n, every term $x \in S(i).\texttt{records}$ with $x[\texttt{subject}] \equiv id$, $n \in x[\texttt{access\_tokens}]$, and the client id $x[\texttt{client\_id}]$ having been issued by i to r,[8] we have that n is not derivable from the attackers knowledge in S (i.e., $n \not\in d_\emptyset(S(\mathsf{attacker}))$).

Session Integrity for Authentication. The two session integrity properties capture that an attacker should be unable to forcefully log a user/browser in at some RP. This includes attacks such as CSRF and session swapping. Note that we define these properties over $\mathcal{OIDC}^w$, i.e., we consider web attackers instead of a network attacker. The reason is that OIDC deployments typically use cookies to track the login sessions of users. Since a network attacker can put cookies into browsers over unencrypted connections and these cookies are then also used for encrypted connections, cookies have no integrity in the presence of a network attacker (see also [48]). In particular, a network attacker could easily break the session integrity of typical OIDC deployments.

For session integrity for authentication we say that a user/browser that is logged in at some RP must have expressed

[7] In the hybrid mode, IdPs can issue two access tokens, cf. Section II-B.

[8] See Definition 54 in Appendix H-B.

her wish to be logged in to that RP in the beginning of the login flow. Note that not even a malicious IdP should be able to forcefully log in its users (more precisely, its user's browsers) at an honest RP. If the IdP is honest, then the user must additionally have authenticated herself at the IdP with the same user account that RP uses for her identification. This excludes, for example, cases where (1) the user is forcefully logged in to an RP by an attacker that plays the role of an IdP, and (2) where an attacker can force an honest user to be logged in at some RP under a false identity issued by an honest IdP.

In our formal definition of session integrity for authentication (below), $\mathsf{loggedIn}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$ denotes that in the processing step Q (see below), the browser b was authenticated (logged in) to an RP r using the IdP i and the identity u in an RP login session with the session id *lsid*. (Here, the processing step Q corresponds to Step $\boxed{18}$ in Figure 2.) The user authentication in the processing step Q is characterized by the browser b receiving the service session id cookie that results from the login session *lsid*.

By $\mathsf{started}^{Q'}_{\rho}(b,r,\mathit{lsid})$ we denote that the browser b, in the processing step Q' triggered the script *script_rp_index* to start a login session which has the session id *lsid* at the RP r. (Compare Section IV on how browsers handle scripts.) Here, Q' corresponds to Step $\boxed{1}$ in Figure 2.

By $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,\mathit{lsid})$ we denote that in the processing step Q'', the user/browser b authenticated to the IdP i. In this case, authentication means that the user filled out the login form (in *script_idp_form*) at the IdP i and, by this, consented to be logged in at r (as in Step $\boxed{13}$ in Figure 2).

Using these notations, we can now define security w.r.t. session integrity for authentication of an OIDC web system with web attackers in a straightforward way:

Definition 4 (Session Integr. for Authentication). Let $\mathcal{OIDC}^w$ be an OIDC web system with web attackers. We say that *$\mathcal{OIDC}^w$ is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDC}^w$, every processing step Q in ρ with $Q = (S,E,N) \rightarrow (S',E',N')$ (for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{IdP}$, every identity u that is owned by b, every $r \in \mathsf{RP}$ that is honest in S, every nonce *lsid*, with $\mathsf{loggedIn}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$, we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b,r,\mathit{lsid})$, and (2) if i is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,\mathit{lsid})$.

Session Integrity for Authorization. For session integrity for authorization we say that if an RP uses some access token at some IdP in a session with a user, then that user expressed her wish to authorize the RP to interact with *some* IdP. Note that one cannot guarantee that the IdP with which RP interacts is the one the user authorized the RP to interact with. This is because the IdP might be malicious. In this case, for example in the discovery phase, the malicious IdP might just claim (in Step $\boxed{3}$ in Figure 2) that some other IdP is responsible for the authentication of the user. If, however, the IdP the user is logged in with is honest, then it should be guaranteed that the user authenticated to that IdP and that the IdP the RP interacts with on behalf of the user is the one intended by the user.

For the formal definition, we use two additional predicates: $\mathsf{usedAuthorization}^{Q}_{\rho}(b,r,i,\mathit{lsid})$ means that the RP r, in a login session (session id *lsid*) with the browser b used some access token to access services at the IdP i. By $\mathsf{actsOnUsersBehalf}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$ we denote that the RP r not only used *some* access token, but used one that is bound to the user's identity at the IdP i.

Again, starting from our informal definition above, we define security w.r.t. session integrity for authorization of an OIDC web system with web attackers in a straightforward way (and similarly to session integrity for authentication):

Definition 5 (Session Integr. for Authorization). Let $\mathcal{OIDC}^w$ be an OIDC web system with web attackers. We say that *$\mathcal{OIDC}^w$ is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDC}^w$, every processing step Q in ρ with $Q = (S,E,N) \rightarrow (S',E',N')$ (for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{IdP}$, every identity u that is owned by b, every $r \in \mathsf{RP}$ that is honest in S, every nonce *lsid*, we have that (1) if $\mathsf{usedAuthorization}^{Q}_{\rho}(b,r,i,\mathit{lsid})$, then there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b,r,\mathit{lsid})$, and (2) if i is honest in S and $\mathsf{actsOnUsersBehalf}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,\mathit{lsid})$.

C. Secondary Security Properties

We define the following secondary security properties that capture specific aspects of OIDC. We use these secondary security properties during our proof of the above main security properties. Nonetheless, these secondary security properties are important and interesting in their own right.

We define and prove the following properties (see the corresponding lemmas in Appendices I-C and I-E for details):

Integrity of Issuer Cache: If a relying party requests the issuer identifier from an identity provider (cf. Steps $\boxed{2}$–$\boxed{3}$ in Figure 2), then the RP will only receive an origin that belongs to this IdP in the response. In other words, honest IdPs do not use attacker-controlled domains as issuer identifiers, and the attacker is unable to alter this information on the way to the RP or in the *issuer cache* at the RP.

Integrity of OIDC Configuration Cache: (1) Honest IdPs only use endpoints under their control in their OIDC configuration document (cf. Steps $\boxed{4}$–$\boxed{5}$ in Figure 2) and (2) this information (which is stored at the RP in the so-called *OIDC configuration cache*) cannot be altered by an attacker.

Integrity of JWKS Cache: RPs receive only "correct" signing keys from honest IdPs, i.e., keys that belong to the respective IdP (cf. Steps $\boxed{6}$–$\boxed{7}$ in Figure 2).

Integrity of Client Registration: Honest RPs register only redirection URIs that point to themselves and that these URIs always use HTTPS. Recall that when an RP registers at an

IdP, the IdP issues a freshly chosen client id to the RP and then stores RP's redirection URIs.

Third Parties Do Not Learn Passwords: Attackers cannot learn user passwords. More precisely, we define that $\mathsf{secretOfID}(id)$, which denotes the password for a given identity id, is not known to any party except for the browser b owning the id and the identity provider i governing the id (as long as b and i are honest).

Attacker Does Not Learn ID Tokens: Attackers cannot learn id tokens that were issued by honest IdPs for honest RPs and identities of honest browsers.

Third Parties Do Not Learn State: If an honest browser logs in at an honest RP using an honest IdP, then the attacker cannot learn the *state* value used in this login flow.

D. Theorem

The following theorem states that OIDC is secure w.r.t. authentication and authorization in presence of the network attacker, and that OIDC is secure w.r.t. session integrity for authentication and authorization in presence of web attackers. For the proof we refer the reader to Appendix I.

Theorem 1. Let $\mathcal{OIDC}^n$ be an OIDC web system with a network attacker. Then, $\mathcal{OIDC}^n$ is secure w.r.t. authentication and authorization. Let $\mathcal{OIDC}^w$ be an OIDC web system with web attackers. Then, $\mathcal{OIDC}^w$ is secure w.r.t. session integrity for authentication and authorization.

E. Comparison to OAuth 2.0

As described in Section II-C, OIDC is based on OAuth 2.0. Since a formal proof for the security of OAuth 2.0 was conducted in [23], one might be tempted to think that a proof for the security of OIDC requires little more than an extension of the proof in [23]. The specific set of features of OIDC introduces, however, important differences that call for new formulations of security properties and require new proofs:

Dynamic Discovery and Registration: Due to the dynamic discovery and registration, RPs can directly influence and manipulate the configuration data that is stored in IdPs. In OAuth, this configuration data is fixed and assumed to be "correct", greatly limiting the options of the attacker. See, for example, the variant [38] of the IdP Mix-up attack that only works in OIDC (mentioned in Section III-A).

Different set of modes: Compared to OAuth, OIDC introduces the hybrid mode, but does not use the resource owner password credentials mode and the client credentials mode.

New endpoints, messages, and parameters: With additional endpoints (and associated HTTPS messages), the attack surface of OIDC is, also for this reason, larger than that of OAuth. The registration endpoints, for example, could be used in ways that break the security of the protocol, which is not possible in OAuth where these endpoints do not exist. In a similar vein, new parameters like nonce, request_uri, and the id token, are contained in several messages (some of which are also present in the original OAuth flow) and potentially change the security requirements for these messages.

Authentication mechanism: The authentication mechanisms employed by OIDC and OAuth are quite different. This shows, in particular, in the fact that OIDC uses the id token mechanism for authentication, while OAuth uses a different, non-standardized mechanism. Additionally, unlike in OAuth, authentication can happen multiple times during one OIDC flow (see the description of the hybrid mode in Section II-B). This greatly influences (the formulation of) security properties, and hence, also the security proofs.

In summary, taking all these differences into account, our security proofs had to be carried out from scratch. At the same time, our proof is more modular than the one in [23] due to the secondary security properties we identified. Moreover, our security properties are similar to the ones by Fett et al. in [23] only on a high level. The underlying definitions in many aspects differ from the ones used for OAuth.[9]

F. Discussion

Using our detailed formal model, we have shown that OIDC enjoys a high level of security regarding authentication, authorization, and session integrity. To achieve this security, it is essential that implementors follow the security guidelines that we stated in Section III. Clearly, in practice, this is not always feasible—for example, many RPs want to include third-party resources for advertisement or user tracking on their origins. As pointed out, however, not following the security guidelines we outline can lead to severe attacks.

We have shown the security of OIDC in the most comprehensive model of the web infrastructure to date. Being a model, however, some features of the web are not included in the FKS model, for example browser plugins. Such technologies can under certain circumstances also undermine the security of OIDC in a manner that is not reflected in our model. Also, user-centric attacks such as phishing or clickjacking attacks are also not covered in the model.

Nonetheless, our formal analysis and the guidelines (along with the attacks when these guidelines are not followed) provide a clear picture of the security provided by OIDC for a large class of adversaries.

VI. Related Work

As already mentioned in the introduction, the only previous works on the security of OIDC are [36], [38]. None of these works establish security guarantees for the OIDC standard: In [36], the authors find implementation errors in deployments of Google Sign-In (which, as mentioned before, is based on OIDC). In [38], the authors describe a variant of the IdP Mix-Up attack (see Section III), highlight the possibility of SSRF attacks at RPs, and show some implementation-specific flaws.

[9] As an example, in [23], the definitions rely on a notion of *OAuth sessions* which are defined by *connected HTTP(S) messages*, i.e., messages that are created by a browser or server in response to another message. In our model, the attacker is involved in each flow of the protocol (for providing the client id, without receiving any prior message), making it hard to apply the notion of OAuth sessions. We instead define the properties using the existing session identifiers. (See Definitions 54, 52, 56–60 in Appendix H for details.)

In our work, however, we aim at establishing and proving security properties for OIDC.

In general, there have been only few formal analysis efforts for web applications, standards, and browsers so far. Most of the existing efforts are based on formal representations of (parts of) web browsers or very limited models of web mechanisms and applications [3]–[5], [9]–[11], [14]–[17], [25], [26], [28], [34], [47].

Only [7], [8] and [19], [21]–[23] were based on a generic formal model of the web infrastructure. In [8], Bansal, Bhargavan, Delignat-Lavaud, and Maffeis analyze the security of OAuth 2.0 with the tool ProVerif in the applied pi-calculus and the WebSpi library. They identify previously unknown attacks on the OAuth 2.0 implementations of Facebook, Yahoo, Twitter, and many other websites. They do not, however, establish security guarantees for OAuth 2.0 and their model is much less expressive than the FKS model.

The relationship of our work to [19], [21]–[23] has been discussed in detail throughout the paper.

VII. Conclusion

Despite being the foundation for many popular and critical login services, OpenID Connect had not been subjected to a detailed security analysis, let alone a formal analysis, before. In this work, we filled this gap.

We developed a detailed and comprehensive formal model of OIDC based on the FKS model, a generic and expressive formal model of the web infrastructure. Using this model, we stated central security properties of OIDC regarding authentication, authorization, and session integrity, and were able to show that OIDC fulfills these properties in our model. By this, we could, for the first time, provide solid security guarantees for one of the most widely deployed single sign-on systems.

To avoid previously known and newly described attacks, we analyzed OIDC with a set of practical and reasonable security measures and best practices in place. We documented these security measures so that they can now serve as guidelines for secure implementations of OIDC.

VIII. Acknowledgements

This work was partially supported by *Deutsche Forschungsgemeinschaft* (DFG) through Grant KU 1434/10-1.

References

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *POPL 2001*, pages 104–115. ACM Press, 2001.

[2] Subresource Integrity – W3C Recommendation 23 June 2016. Jun. 23, 2016.

[3] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *CSF 2010*, pages 290–304. IEEE Computer Society, 2010.

[4] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. An authentication flaw in browser-based single sign-on protocols: Impact and remediations. *Computers & Security*, 33:41–58. Elsevier, 2013.

[5] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps. In V. Shmatikov, editor, *FMSE 2008*, pages 1–10. ACM, 2008.

[6] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *NDSS 2011*. The Internet Society, 2011.

[7] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *POST 2013*, volume 7796 of *LNCS*, pages 126–146. Springer, 2013.

[8] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *Journal of Computer Security*, 22(4):601–657. IOS Press, 2014.

[9] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *NDSS 2015*. The Internet Society, 2015.

[10] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *NSS 2011*, pages 97–104. IEEE, 2011.

[11] A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *USENIX conference on Web application development*, pages 11–11. USENIX Association, 2010.

[12] J. Bradley. The problem with OAuth for Authentication. Blog post. Jan. 2012. http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html.

[13] J. Bradley, T. Lodderstedt, and H. Zandbelt. Encoding claims in the OAuth 2 state parameter using a JWT – draft-bradley-oauth-jwt-encoded-state-05. IETF. Dec. 2015. https://tools.ietf.org/html/draft-bradley-oauth-jwt-encoded-state-05.

[14] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. CookiExt: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[15] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta. Provably Sound Browser-Based Enforcement of Web Session Integrity. In *CSF 2014*, pages 366–380. IEEE Computer Society, 2014.

[16] S. Calzavara, R. Focardi, N. Grimm, and M. Maffei. Micro-policies for Web Session Security. In *CSF 2016*, pages 179–193. IEEE Computer Society, 2016.

[17] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Krügel, G. Vigna, and Y. Chen. Protecting Web-Based Single Sign-on Protocols against Relying Party Impersonation Attacks through a Dedicated Bi-directional Authenticated Secure Channel. In *RAID 2014*, volume 8688 of *LNCS*, pages 276–298. Springer, 2014.

[18] J. Eisinger and E. Stark. Referrer Policy – Editor's Draft, 28 March 2016. W3C. Mar. 2016. https://w3c.github.io/webappsec-referrer-policy/.

[19] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *S&P 2014*, pages 673–688. IEEE Computer Society, 2014.

[20] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. Technical Report arXiv:1411.7210, arXiv, 2014. http://arxiv.org/abs/1411.7210.

[21] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. In *ESORICS 2015*, volume 9326 of *LNCS*, pages 43–65. Springer, 2015.

[22] D. Fett, R. Küsters, and G. Schmitz. SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web. In *CCS 2015*, pages 1358–1369. ACM, 2015.

[23] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *CCS 2016*, pages 1204–1215. ACM, 2016.

[24] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. Technical Report arXiv:1601.01229, arXiv, 2016. Available at http://arxiv.org/abs/1601.01229.

[25] T. Groß, B. Pfitzmann, and A. Sadeghi. Browser Model for Security Analysis of Browser-Based Protocols. In *ESORICS 2005*, volume 3679 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2005.

[26] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified Security for Browser Extensions. In *S&P 2011*, pages 115–130. IEEE Computer Society, 2011.

[27] D. Hardt (ed.). RFC6749 – The OAuth 2.0 Authorization Framework. IETF. Oct. 2012. https://tools.ietf.org/html/rfc6749.

[28] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 24(2):181–234. IOS Press, 2016.

[29] HTML5, W3C Recommendation. Oct. 28, 2014.

[30] M. Jones, J. Bradley, and B. Campbell. OAuth 2.0 Token Binding – draft-ietf-oauth-token-binding-01. IETF. Mar. 2016. https://tools.ietf.org/html/draft-ietf-oauth-token-binding-01.

[31] M. Jones, J. Bradley, and N. Sakimura. OAuth 2.0 Mix-Up Mitigation – draft-ietf-oauth-mix-up-mitigation-01. IETF. Jul. 2016. https://tools.ietf.org/html/draft-ietf-oauth-mix-up-mitigation-01.

[32] M. Jones, J. Bradley, and N. Sakimura. RFC7519 – JSON Web Token (JWT). IETF. May 2015. https://tools.ietf.org/html/rfc7519.

[33] P. Jones, G. Salgueiro, M. Jones, and J. Smarr. RFC7033 – WebFinger. IETF. Sep. 2013. https://tools.ietf.org/html/rfc7033.

[34] F. Kerschbaum. Simple Cross-Site Attack Prevention. In *SecureComm 2007*, pages 464–472. IEEE Computer Society, 2007.

[35] W. Li and C. J. Mitchell. Security issues in OAuth 2.0 SSO implementations. In *ISC 2014*, pages 529–541. Springer, 2014.

[36] W. Li and C. J. Mitchell. Analysing the Security of Google's Implementation of OpenID Connect. In *DIMVA 2016*, volume 9721, pages 357–376. Springer, 2016.

[37] T. Lodderstedt (ed.), M. McGloin, and P. Hunt. RFC6819 – OAuth 2.0 Threat Model and Security Considerations. IETF. Jan. 2013. https://tools.ietf.org/html/rfc6819.

[38] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich. SoK: Single Sign-On Security – An Evaluation of OpenID Connect. In *EuroS&P 2017*, 2017. IEEE Computer Society, 2017.

[39] Open Web Application Security Project (OWASP). Session fixation. https://www.owasp.org/index.php/Session_Fixation.

[40] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow. Uses and Abuses of Server-Side Requests. In *RAID 2016*, volume 9854 of *LNCS*, pages 393–414. Springer, 2016.

[41] N. Sakimura, J. Bradley, and M. Jones. OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1. OpenID Foundation. Nov. 8, 2014. http://openid.net/specs/openid-connect-registration-1_0.html.

[42] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0 incorporating errata set 1. OpenID Foundation. Nov. 8, 2014. http://openid.net/specs/openid-connect-core-1_0.html.

[43] N. Sakimura, J. Bradley, M. Jones, and E. Jay. OpenID Connect Discovery 1.0 incorporating errata set 1. OpenID Foundation. Nov. 8, 2014. http://openid.net/specs/openid-connect-discovery-1_0.html.

[44] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *CCS 2012*, pages 378–390. ACM, 2012.

[45] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security 2013*, pages 399–314. USENIX Association, 2013.

[46] M. West. Content Security Policy Level 3 – W3C Working Draft, 13 September 2016. W3C. Sep. 2016. https://www.w3.org/TR/2016/WD-CSP3-20160913/.

[47] S. Yoshihama, T. Tateishi, N. Tabuchi, and T. Matsumoto. Information-Flow-Based Access Control for Web Browsers. *IEICE Transactions*, 92-D(5):836–850, 2009.

[48] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *USENIX Security 2015)*, pages 707–721. USENIX Association, 2015.

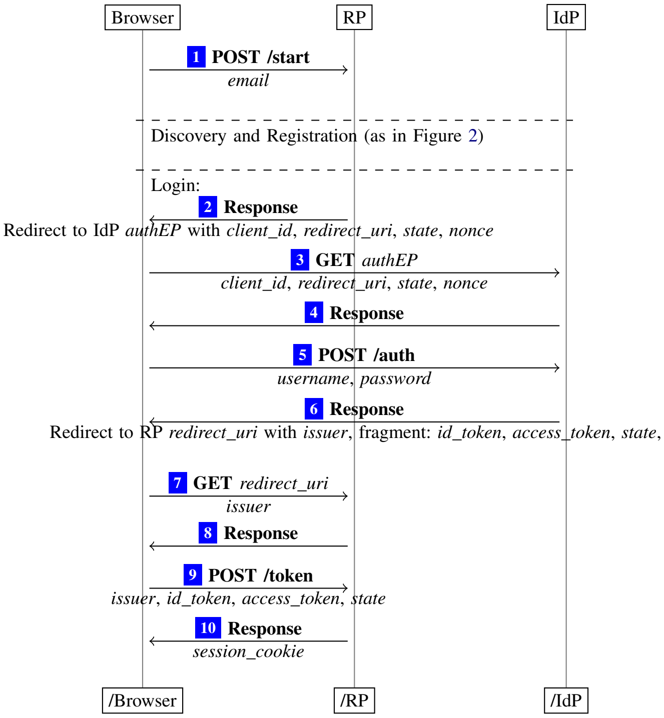
Browser
RP
IdP
1 POST /start
email
Discovery and Registration (as in Figure 2)
Login:
2 Response
Redirect to IdP authEP with client_id, redirect_uri, state, nonce
3 GET authEP
client_id, redirect_uri, state, nonce
4 Response
5 POST /auth
username, password
6 Response
Redirect to RP redirect_uri with issuer, fragment: id_token, access_token, state,
7 GET redirect_uri
issuer
8 Response
9 POST /token
issuer, id_token, access_token, state
10 Response
session_cookie
/Browser
/RP
/IdP

Figure 3. OpenID Connect implicit mode.

Appendix A
The IdP Mix-Up Attack

As described in Section III-A above, in the IdP Mix-Up attack, an honest RP gets confused about which IdP is used in a login flow. The honest RP assumes that the login uses the attacker's IdP and interacts with this IdP, while the user's browser interacts with an honest IdP and relays the data acquired at this IdP to the RP. As a result, the attacker learns information such as authorization codes and access tokens he is not supposed to know and that allow him to break the authentication and authorization properties.

There exist several variants of this attack [23], [38]. Here, we describe two variants of this attack using the hybrid mode of OIDC. The normal flow of the hybrid mode is depicted in Figure 4, the attack is depicted in Figures 5 and 6 (without the mitigation against the Mix-Up attack presented in Section III-A).

To start the login flow, the user selects an IdP at RP (by entering her email address) in Step 1. This step is the only difference between the two variants that we describe: In Variant 1, the user selects a malicious IdP, say AIdP. In Variant 2, the user selects an honest IdP, but the request is intercepted by the attacker and altered such that the attacker replaces the honest IdP by AIdP (*email* is replaced by *email*$'$ in Steps 1 and 2 in Figure 5).[10]

Now, RP starts with the discovery phase of the protocol. As RP thinks that the user wants to login with AIdP, it retrieves the OIDC configuration from AIdP (Steps 5 and 6). In this configuration, the attacker does not let all endpoint URLs point to himself, as would be usual for OIDC, but instead sets the authorization endpoint to be the one of HIdP. Next, the RP registers itself at AIdP (Steps 9 and 10). In this step, AIdP issues the same client id to RP which RP is registered with at HIdP (client id are always public). This is important as HIdP will later redirect the user's browser back to RP and checks the redirect URI based on the client id.

Next, RP redirects the user's browser to HIdP (Variant 1) or AIdP (Variant 2) in Step 11. In Variant 1 of the attack, a vigilant user might now be able to detect that she tried to log in using AIdP but instead is redirected to HIdP. This does not happen in Variant 2, but here the attacker needs to replace the redirection to AIdP by a redirection to HIdP (which should not be any problem if he succeeded in altering the first step of the protocol).

The user then authenticates at HIdP and is redirected back to RP along with an authorization code and an access token (depending on the sub-mode of the hybrid flow, IdPs do not send id tokens in this step). Now, RP retrieves the authorization code and the access token from the user's browser and continues the login flow. As RP still assumes that AIdP is used in this case, it tries to redeem the authorization code for an id token (and a second access token) at AIdP in Step 19.

[10] This initial request is often unencrypted in practice, see [23].

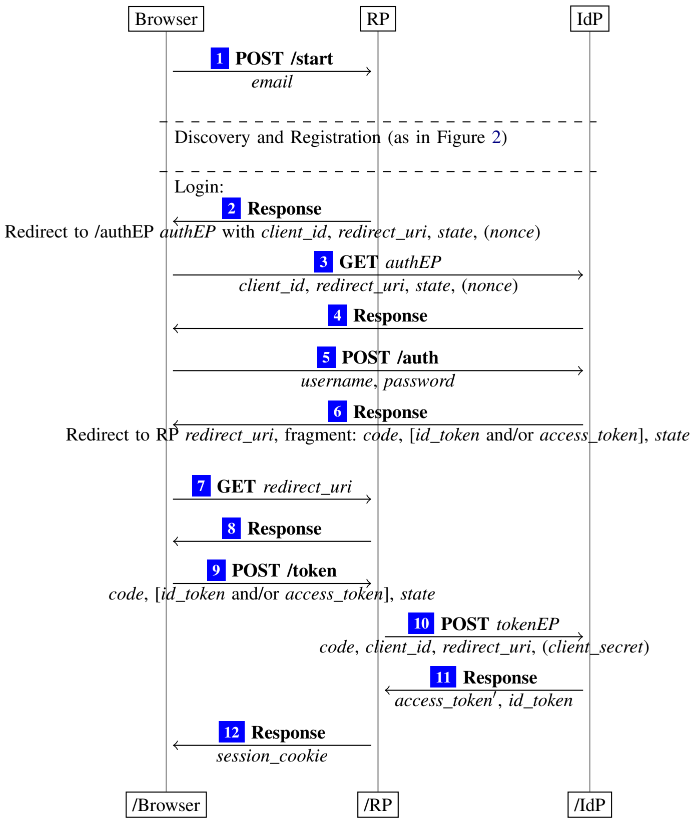
Browser
RP
IdP
1 POST /start
email
Discovery and Registration (as in Figure 2)
Login:
2 Response
Redirect to /authEP authEP with client_id, redirect_uri, state, (nonce)
3 GET authEP
client_id, redirect_uri, state, (nonce)
4 Response
5 POST /auth
username, password
6 Response
Redirect to RP redirect_uri, fragment: code, [id_token and/or access_token], state
7 GET redirect_uri
8 Response
9 POST /token
code, [id_token and/or access_token], state
10 POST tokenEP
code, client_id, redirect_uri, (client_secret)
11 Response
access_token′, id_token
12 Response
session_cookie
/Browser
/RP
/IdP

Figure 4. OpenID Connect hybrid mode.

As the authorization code has not been redeemed at HIdP yet, the code is still valid and the attacker may start a second login flow (pretending to be the user) at RP (Steps 20 ff.). The attacker skips the authentication at HIdP and returns to RP with the authorization code he has learned before. RP now redeems this code at HIdP and receives an id token issued for the honest user and consequently assumes that the attacker has the identity of the user and logs the attacker in.

In another variation of the attack, if HIdP does not issue client secrets to RPs, the attacker can also redeem the authorization code by himself (Steps 26 f.). In this case, the attacker receives an access token valid for the user's account. With this access token, he can retrieve data of the user or act on the user's behalf at HIdP. (As he redeems the authorization code, he cannot use it to log himself into the RP in this case.)

In any case, the attacker can also respond to the authorization code sent to his token endpoint in Step 24 with a mock access token and a mock id token (which will not be used in the following). In the next step, the RP might then use the access token learned *from the honest IdP* in Step 18 to retrieve data of the user from AIdP (Steps 24 ff.).[11] Then the attacker learns also this access token, which (as described in the paragraph above) grants him unauthorized access to the user's account at HIdP.

This shows that, using the IdP Mix-Up attack, an attacker can successfully impersonate users at RPs and access their data at honest IdPs. The mitigation presented in Section III-A would have prevented the attack in Step 16 ff.

[11] Depending on the RP implementation, the RP might choose to use the mock access token or the one learned from the honest IdP in this step. In the real-world implementation mod_auth_openidc, the access token from the honest IdP was used.

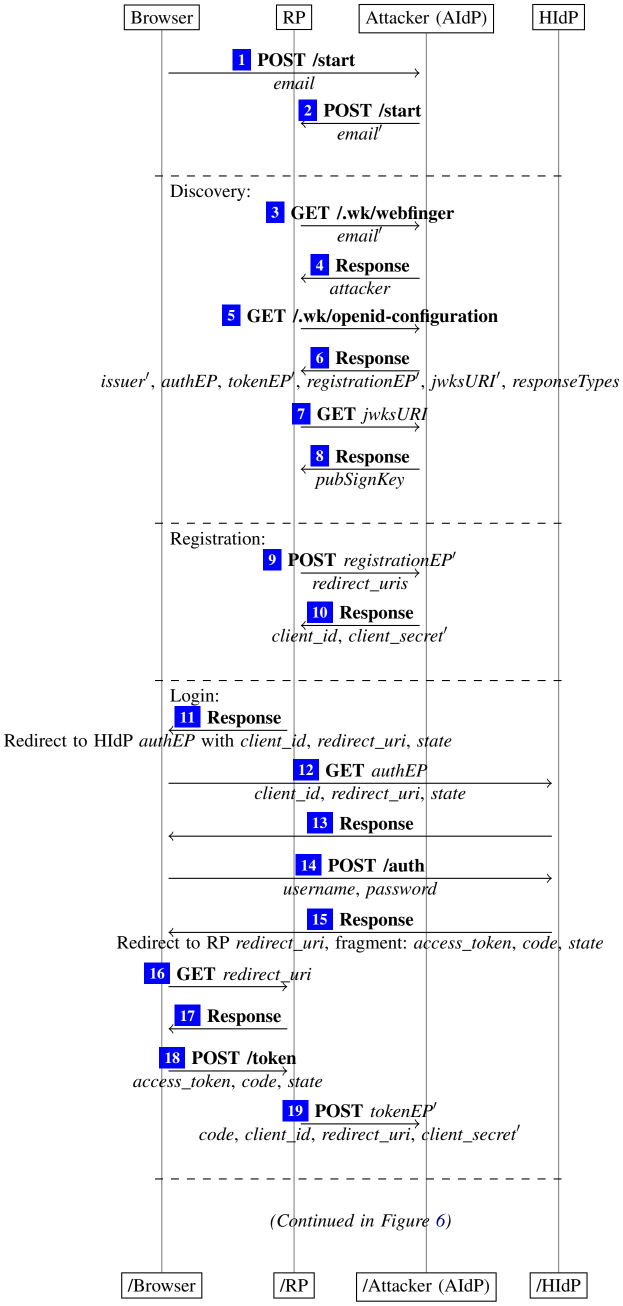

Figure 5. Attack on OpenID Connect hybrid mode.

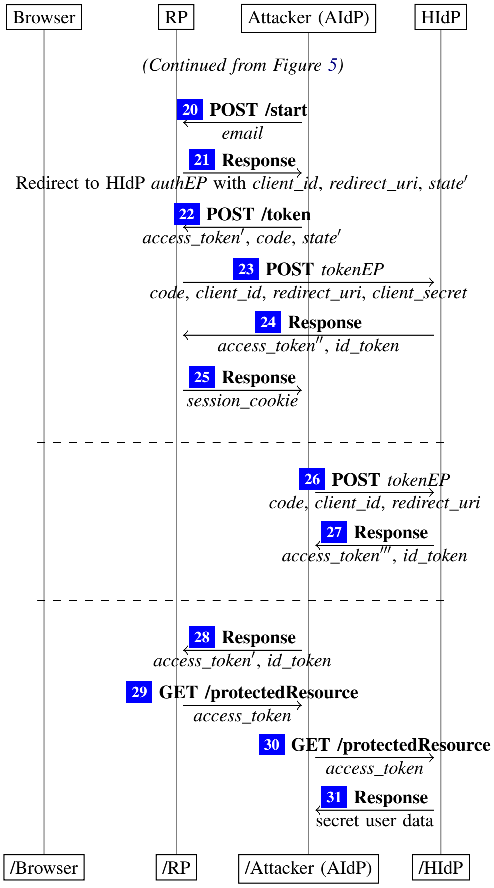

Figure 6. Attack on OpenID Connect hybrid mode (continued).

$$
\begin{aligned}
\mathsf{dec_a}(\mathsf{enc_a}(x,\mathsf{pub}(y)),y) &= x && (1)\\
\mathsf{dec_s}(\mathsf{enc_s}(x,y),y) &= x && (2)\\
\mathsf{checksig}(\mathsf{sig}(x,y),\mathsf{pub}(y)) &= \top && (3)\\
\mathsf{extractmsg}(\mathsf{sig}(x,y)) &= x && (4)\\
\pi_i(\langle x_1,\dots,x_n\rangle) &= x_i \ \text{ if } 1 \le i \le n && (5)\\
\pi_j(\langle x_1,\dots,x_n\rangle) &= \diamond \ \text{ if } j \notin \{1,\dots,n\} && (6)
\end{aligned}
$$

Figure 7. Equational theory for Σ.

Appendix B
The FKS Web Model

In this and the following two sections, we present the FKS model for the web infrastructure as proposed in [19], [20], and [24], along with the addition of a generic model for HTTPS web servers that harmonizes the behavior of such servers and facilitates easier proofs.

A. Communication Model

We here present details and definitions on the basic concepts of the communication model.

Terms, Messages and Events. The signature Σ for the terms and messages considered in this work is the union of the following pairwise disjoint sets of function symbols:

- constants $C = \mathsf{IPs} \cup \mathbb{S} \cup \{\top, \bot, \diamond\}$ where the three sets are pairwise disjoint, $\mathbb{S}$ is interpreted to be the set of ASCII strings (including the empty string ε), and IPs is interpreted to be a set of (IP) addresses,
- function symbols for public keys, (a)symmetric encryption/decryption, and signatures: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot,\cdot)$, and $\mathsf{extractmsg}(\cdot)$,
- n-ary sequences $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc., and
- projection symbols $\pi_i(\cdot)$ for all $i \in \mathbb{N}$.

For strings (elements in $\mathbb{S}$), we use a specific font. For example, `HTTPReq` and `HTTPResp` are strings. We denote by $\mathsf{Doms} \subseteq \mathbb{S}$ the set of domains, e.g., `example.com` $\in \mathsf{Doms}$. We denote by $\mathsf{Methods} \subseteq \mathbb{S}$ the set of methods used in HTTP requests, e.g., `GET`, `POST` $\in \mathsf{Methods}$.

The equational theory associated with the signature Σ is given in Figure 7.

Definition 6 (Nonces and Terms). By $X = \{x_0, x_1, \dots\}$ we denote a set of variables and by $\mathcal{N}$ we denote an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol in Σ for some $n \ge 0$ and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.

By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by the theory associated with Σ. For example, we have that $\pi_1(\mathsf{dec_a}(\mathsf{enc_a}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{pub}(k)), k)) \equiv \mathsf{a}$.

Definition 7 (Ground Terms, Messages, Placeholders, Protomessages). By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

We define the set $V_{\text{process}} = \{\nu_1, \nu_2, \dots\}$ of variables (called placeholders). The set $\mathcal{M}^\nu := \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$ is called the set of *protomessages*, i.e., messages that can contain placeholders.

Example 1. For example, $k \in \mathcal{N}$ and $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc_a}(\langle a,b,c\rangle, \mathsf{pub}(k))$ is interpreted to be the message $\langle a,b,c\rangle$ (the sequence of constants a, b, c) encrypted by the public key $\mathsf{pub}(k)$.

Definition 8 (Normal Form). Let t be a term. The *normal form* of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Figure 7. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 9 (Pattern Matching). Let *pattern* $\in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches pattern* iff t can be acquired from *pattern* by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim$ *pattern*. For a sequence of patterns *patterns* we write $t \dot{\sim}$*patterns* to denote that t matches at least one pattern in *patterns*.

For a term t' we write $t'|\mathit{pattern}$ to denote the term that is acquired from t' by removing all immediate subterms of t' that do not match *pattern*.

Example 2. For example, for a pattern $p = \langle \top, * \rangle$ we have that $\langle \top, 42 \rangle \sim p$, $\langle \bot, 42 \rangle \not\sim p$, and

$$\langle \langle \bot, \top \rangle, \langle \top, 23 \rangle, \langle \mathtt{a}, \mathtt{b} \rangle, \langle \top, \bot \rangle \rangle | p = \langle \langle \top, 23 \rangle, \langle \top, \bot \rangle \rangle \ .$$

Definition 10 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$, and $t_1, \dots, t_n \in \mathcal{T}_N$.

By $\tau[t_1/x_1, \dots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \dots, n\}$.

Definition 11 (Events and Protoevents). An *event (over* IPs *and* $\mathcal{M}$*)* is a term of the form $\langle a, f, m \rangle$, for a, $f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events. Events over IPs and $\mathcal{M}^\nu$ are called *protoevents* and are denoted $\mathcal{E}^\nu$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^\nu\langle\rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle\rangle$, $\langle \langle a, f, m \rangle, \langle a', f', m' \rangle, \dots \rangle$, etc.).

Notations.

Definition 12 (Sequence Notations). For a sequence $t = \langle t_1, \dots, t_n \rangle$ and a set s we use $t \subset^{\langle\rangle} s$ to say that $t_1, \dots, t_n \in s$. We define $x \in^{\langle\rangle} t \iff \exists i \colon t_i = x$. For a term y we write $t +^{\langle\rangle} y$ to denote the sequence $\langle t_1, \dots, t_n, y \rangle$. For a sequence $r = \langle r_1, \dots, r_m \rangle$ we write $t \cup r$ to denote the sequence $\langle t_1, \dots, t_n, r_1, \dots, r_m \rangle$. For a finite set M with $M = \{m_1, \dots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \dots, m_n \rangle$. (The order of the elements does not matter; one is chosen arbitrarily.)

Definition 13. A *dictionary over* X *and* Y is a term of the form

$$\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$$

where $k_1, \dots, k_n \in X$, $v_1, \dots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \dots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$.

We note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$. Figure 8 shows the short notation for dictionary operations. For a dictionary $z = [k_1 : v_1, k_2 : v_2, \dots, k_n : v_n]$ we write $k \in z$ to say that there exists i such that $k = k_i$. We write $z[k_j]$ to refer to the value v_j. (Note that if a dictionary contains two elements $\langle k, v \rangle$ and $\langle k, v' \rangle$, then the notations and operations for dictionaries apply non-deterministically to one of both elements.) If $k \notin z$, we set $z[k] := \langle\rangle$.

$$[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n]\,[k_i] = v_i \tag{7}$$

$$\begin{aligned}[k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1} \dots, k_n : v_n] - k_i = \\ [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1} \dots, k_n : v_n]\end{aligned} \tag{8}$$

Figure 8. Dictionary operators with $1 \leq i \leq n$.

Given a term $t = \langle t_1, \dots, t_n \rangle$, we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 14. A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

Example 3. For the term $\tau = \langle a, b, \langle c, d, \langle e, f \rangle \rangle \rangle$ and the pointer $\overline{p} = \langle 3, 1 \rangle$, the subterm of τ at the position $\overline{p}$ is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3, 1 \rangle = \tau.3.\overline{p} = \tau.3.3.1 = e$.

To improve readability, we try to avoid writing, e.g., $o.2$ or $\pi_2(o)$ in this document. Instead, we will use the names of the components of a sequence that is of a defined form as pointers that point to the corresponding subterms. E.g., if an *Origin* term is defined as $\langle \mathit{host}, \mathit{protocol} \rangle$ and o is an Origin term, then we can write $o.\mathtt{protocol}$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

Atomic Processes, Systems and Runs. An atomic process takes its current state and an event as input, and then (non-deterministically) outputs a new state and a set of events.

Definition 15 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

$$p = (I^p, Z^p, R^p, s_0^p)$$

where $I^p \subseteq \mathsf{IPs}$, $Z^p \in \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^\nu \langle\rangle} \times \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (input event and old state map to sequence of output events and new state), and $s_0^p \in Z^p$ is the initial state of p. For any new state s and any sequence of nonces $(\eta_1, \eta_2, \dots)$ we demand that $s[\eta_1/\nu_1, \eta_2/\nu_2, \dots] \in Z^p$. A *system* $\mathcal{P}$ is a (possibly infinite) set of atomic processes.

Definition 16 (Configurations). A *configuration of a system* $\mathcal{P}$ is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence[12] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$.

Definition 17 (Concatenating terms and sequences). For a term $a = \langle a_1, \dots, a_i \rangle$ and a sequence $b = (b_1, b_2, \dots)$, we define the *concatenation* as $a \cdot b := (a_1, \dots, a_i, b_1, b_2, \dots)$.

Definition 18 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a non-deterministically chosen occurence of that element in X is removed.

Definition 19 (Processing Steps). A *processing step of the system* $\mathcal{P}$ is of the form

$$(S, E, N) \xrightarrow[p \to E_{\text{out}}]{e_{\text{in}} \to p} (S', E', N')$$

where

1) (S, E, N) and (S', E', N') are configurations of $\mathcal{P}$,
2) $e_{\text{in}} = \langle a, f, m \rangle \in E$ is an event,
3) $p \in \mathcal{P}$ is a process,
4) E_{out} is a sequence (term) of events

such that there exists

1) a sequence (term) $E_{\text{out}}^\nu \subseteq 2^{\mathcal{E}^\nu \langle\rangle}$ of protoevents,
2) a term $s^\nu \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
3) a sequence $(\nu_1, \nu_2, \dots, \nu_i)$ of all placeholders appearing in E_{out}^ν (ordered lexicographically),
4) a sequence $N^\nu = (\eta_1, \eta_2, \dots, \eta_i)$ of the first i elements in N

with

1) $((e_{\text{in}}, S(p)), (E_{\text{out}}^\nu, s^\nu)) \in R^p$ and $a \in I^p$,
2) $E_{\text{out}} = E_{\text{out}}^\nu[m_1/\nu_1, \dots, m_i/\nu_i]$
3) $S'(p) = s^\nu[m_1/\nu_1, \dots, m_i/\nu_i]$ and $S'(p') = S(p')$ for all $p' \neq p$
4) $E' = E_{\text{out}} \cdot (E \setminus \{e_{\text{in}}\})$
5) $N' = N \setminus N^\nu$

We may omit the superscript and/or subscript of the arrow.

Intuitively, for a processing step, we select one of the processes in $\mathcal{P}$, and call it with one of the events in the list of waiting events E. In its output (new state and output events), we replace any occurences of placeholders ν_x by "fresh" nonces from N (which we then remove from N). The output events are then prepended to the list of waiting events, and the state of the process is reflected in the new configuration.

Definition 20 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run* ρ *of a system* $\mathcal{P}$ *initiated by* E^0 *with nonces* N^0 is a finite sequence of configurations $((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \dots)$ such that $S^0(p) = s_0^p$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

We denote the state $S^n(p)$ of a process p at the end of a run ρ by $\rho(p)$.

Usually, we will initiate runs with a set E^0 containing infinite trigger events of the form $\langle a, a, \mathtt{TRIGGER} \rangle$ for each $a \in \mathsf{IPs}$, interleaved by address.

Atomic Dolev-Yao Processes. We next define atomic Dolev-Yao processes, for which we require that the messages and states that they output can be computed (more formally, derived) from the current input event and state. For this purpose, we first define what it means to derive a message from given messages.

[12] Here: Not in the sense of terms as defined earlier.

Definition 21 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with placeholders V* if there exist $n \geq 0$, $m_1, \dots, m_n \in M$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \dots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \dots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.

For example, $a \in d_{\{\}}(\{\mathsf{enc_a}(\langle a,b,c\rangle, \mathsf{pub}(k)), k\})$.

Definition 22 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ such that (I^p, Z^p, R^p, s_0^p) is an atomic process and (1) $Z^p \subseteq \mathcal{T}_\mathcal{N}$ (and hence, $s_0^p \in \mathcal{T}_\mathcal{N}$), and (2) for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_\mathcal{N}$, $s' \in \mathcal{T}_\mathcal{N}(V_\text{process})$, with $((e,s),(E,s')) \in R^p$ it holds true that E, $s' \in d_{V_\text{process}}(\{e,s\})$.

Definition 23 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$* is an atomic DY process $p = (I,Z,R,s_0)$ such that for all events e, and $s \in \mathcal{T}_\mathcal{N}$ we have that $((e,s),(E,s')) \in R$ iff $s' = \langle e, E, s\rangle$ and $E = \langle\langle a_1, f_1, m_1\rangle, \dots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \dots, a_n \in \mathsf{IPs}$, $f_0, \dots, f_n \in A$, $m_1, \dots, m_n \in d_{V_\text{process}}(\{e,s\})$.

B. Scripts

We define scripts, which model client-side scripting technologies, such as JavaScript. Scripts are defined similarly to DY processes.

Definition 24 (Placeholders for Scripts). By $V_\text{script} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 25 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_\mathcal{N} \times \mathcal{T}_\mathcal{N}(V_\text{script})$ such that for all $s \in \mathcal{T}_\mathcal{N}$, $s' \in \mathcal{T}_\mathcal{N}(V_\text{script})$ with $(s,s') \in R$ it follows that $s' \in d_{V_\text{script}}(s)$.

A script is called by the browser which provides it with state information (such as the script's last state and limited information about the browser's state) s. The script then outputs a term s', which represents the new internal state and some command which is interpreted by the browser. The term s' may contain variables $\lambda_1, \dots$ which the browser will replace by (otherwise unused) placeholders $\nu_1, \dots$ which will be replaced by nonces once the browser DY process finishes (effectively providing the script with a way to get "fresh" nonces).

Similarly to an attacker process, we define the *attacker script R^att*:

Definition 26 (Attacker Script). The attacker script R^att outputs everything that is derivable from the input, i.e., $R^\text{att} = \{(s,s') \mid s \in \mathcal{T}_\mathcal{N}, s' \in d_{V_\text{script}}(s)\}$.

C. Web System

The web infrastructure and web applications are formalized by what is called a web system. A web system contains, among others, a (possibly infinite) set of DY processes, modeling web browsers, web servers, DNS servers, and attackers (which may corrupt other entities, such as browsers).

Definition 27. A *web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a tuple with its components defined as follows:

The first component, $\mathcal{W}$, denotes a system (a set of DY processes) and is partitioned into the sets Hon, Web, and Net of honest, web attacker, and network attacker processes, respectively.

Every $p \in \mathsf{Web} \cup \mathsf{Net}$ is an attacker process for some set of sender addresses $A \subseteq \mathsf{IPs}$. For a web attacker $p \in \mathsf{Web}$, we require its set of addresses I^p to be disjoint from the set of addresses of all other web attackers and honest processes, i.e., $I^p \cap I^{p'} = \emptyset$ for all $p' \in \mathsf{Hon} \cup \mathsf{Web}$. Hence, a web attacker cannot listen to traffic intended for other processes. Also, we require that $A = I^p$, i.e., a web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on I^p) and may spoof all addresses (i.e., the set A may be IPs).

Every $p \in \mathsf{Hon}$ is a DY process which models either a *web server*, a *web browser*, or a *DNS server*, as further described in the following subsections. Just as for web attackers, we require that p does not spoof sender addresses and that its set of addresses I^p is disjoint from those of other honest processes and the web attackers.

The second component, $\mathcal{S}$, is a finite set of scripts such that $R^\text{att} \in \mathcal{S}$. The third component, script, is an injective mapping from $\mathcal{S}$ to $\mathbb{S}$, i.e., by script every $s \in \mathcal{S}$ is assigned its string representation $\mathsf{script}(s)$.

Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \mathtt{TRIGGER}\rangle$ for every $a \in \bigcup_{p \in \mathcal{W}} I^p$.

A *run* of $\mathcal{WS}$ is a run of $\mathcal{W}$ initiated by E^0.

Appendix C
Message and Data Formats

We now provide some more details about data and message formats that are needed for the formal treatment of the web model and the analysis presented in the following.

A. URLs

Definition 28. A *URL* is a term of the form

$$\langle \mathtt{URL}, \mathit{protocol}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{fragment} \rangle$$

with $\mathit{protocol} \in \{\mathtt{P}, \mathtt{S}\}$ (for **p**lain (HTTP) and **s**ecure (HTTPS)), $\mathit{host} \in \mathsf{Doms}$, $\mathit{path} \in \mathbb{S}$, $\mathit{parameters} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, and $\mathit{fragment} \in \mathcal{T}_{\mathcal{N}}$. The set of all valid URLs is URLs.

The *fragment* part of a URL can be omitted when writing the URL. Its value is then defined to be $\bot$.

Example 4. For the URL $u = \langle \mathtt{URL}, a, b, c, d \rangle$, $u.\mathtt{protocol} = a$. If, in the algorithm described later, we say $u.\mathtt{path} := e$ then $u = \langle \mathtt{URL}, a, b, c, e \rangle$ afterwards.

B. Origins

Definition 29. An *origin* is a term of the form $\langle \mathit{host}, \mathit{protocol} \rangle$ with $\mathit{host} \in \mathsf{Doms}$ and $\mathit{protocol} \in \{\mathtt{P}, \mathtt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins.

Example 5. For example, $\langle \mathtt{FOO}, \mathtt{S} \rangle$ is the HTTPS origin for the domain $\mathtt{FOO}$, while $\langle \mathtt{BAR}, \mathtt{P} \rangle$ is the HTTP origin for the domain $\mathtt{BAR}$.

C. Cookies

Definition 30. A *cookie* is a term of the form $\langle \mathit{name}, \mathit{content} \rangle$ where $\mathit{name} \in \mathcal{T}_{\mathcal{N}}$, and *content* is a term of the form $\langle \mathit{value}, \mathit{secure}, \mathit{session}, \mathit{httpOnly} \rangle$ where $\mathit{value} \in \mathcal{T}_{\mathcal{N}}$, *secure*, *session*, $\mathit{httpOnly} \in \{\top, \bot\}$. We write $\mathsf{Cookies}$ for the set of all cookies and $\mathsf{Cookies}^V$ for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

If the *secure* attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections. If the *session* flag is set, this cookie will be deleted as soon as the browser is closed. The *httpOnly* attribute controls whether JavaScript has access to this cookie.

Note that cookies of the form described here are only contained in HTTP(S) requests. In responses, only the components *name* and *value* are transferred as a pairing of the form $\langle \mathit{name}, \mathit{value} \rangle$.

D. HTTP Messages

Definition 31. An *HTTP request* is a term of the form shown in (9). An *HTTP response* is a term of the form shown in (10).

$$\langle \mathtt{HTTPReq}, \mathit{nonce}, \mathit{method}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body} \rangle \tag{9}$$

$$\langle \mathtt{HTTPResp}, \mathit{nonce}, \mathit{status}, \mathit{headers}, \mathit{body} \rangle \tag{10}$$

The components are defined as follows:

- $\mathit{nonce} \in \mathcal{N}$ serves to map each response to the corresponding request
- $\mathit{method} \in \mathsf{Methods}$ is one of the HTTP methods.
- $\mathit{host} \in \mathsf{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $\mathit{path} \in \mathbb{S}$ is a string indicating the requested resource at the server side
- $\mathit{status} \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard)
- $\mathit{parameters} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains URL parameters
- $\mathit{headers} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, containing request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \mathtt{Origin}, o \rangle$ where o is an origin,
  - $\langle \mathtt{Set\text{-}Cookie}, c \rangle$ where c is a sequence of cookies,
  - $\langle \mathtt{Cookie}, c \rangle$ where $c \in, \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ (note that in this header, only names and values of cookies are transferred),
  - $\langle \mathtt{Location}, l \rangle$ where $l \in \mathsf{URLs}$,
  - $\langle \mathtt{Referer}, r \rangle$ where $r \in \mathsf{URLs}$,
  - $\langle \mathtt{Strict\text{-}Transport\text{-}Security}, \top \rangle$,
  - $\langle \mathtt{Authorization}, \langle \mathit{username}, \mathit{password} \rangle \rangle$ where *username*, $\mathit{password} \in \mathbb{S}$,
  - $\langle \mathtt{ReferrerPolicy}, p \rangle$ where $p \in \{\mathtt{noreferrer}, \mathtt{origin}\}$
- $\mathit{body} \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

We write $\mathsf{HTTPRequests}$/$\mathsf{HTTPResponses}$ for the set of all HTTP requests or responses, respectively.

Example 6 (HTTP Request and Response).

$$\begin{aligned} r :=& \langle \mathtt{HTTPReq}, n_1, \mathtt{POST}, \mathtt{example.com}, \mathtt{/show}, \langle\langle \mathtt{index}, 1\rangle\rangle, \\ & [\mathtt{Origin}: \langle \mathtt{example.com}, \mathtt{S}\rangle], \langle \mathtt{foo}, \mathtt{bar}\rangle\rangle \qquad (11) \\ s :=& \langle \mathtt{HTTPResp}, n_1, 200, \langle\langle \mathtt{Set\text{-}Cookie}, \langle\langle \mathtt{SID}, \langle n_2, \bot, \bot, \top\rangle\rangle\rangle\rangle\rangle, \langle \mathtt{somescript}, x\rangle\rangle \qquad (12) \end{aligned}$$

An HTTP $\mathtt{GET}$ request for the URL http://example.com/show?index=1 is shown in (11), with an Origin header and a body that contains $\langle \mathtt{foo}, \mathtt{bar}\rangle$. A possible response is shown in (12), which contains an httpOnly cookie with name $\mathtt{SID}$ and value n_2 as well as the string representation $\mathtt{somescript}$ of the script $\mathsf{script}^{-1}(\mathtt{somescript})$ (which should be an element of $\mathcal{S}$) and its initial state x.

Encrypted HTTP Messages. For HTTPS, requests are encrypted using the public key of the server. Such a request contains an (ephemeral) symmetric key chosen by the client that issued the request. The server is supported to encrypt the response using the symmetric key.

Definition 32. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k'\rangle, k)$, where $k \in \mathit{terms}$, $k' \in \mathcal{N}$, and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc_s}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

We say that an HTTP(S) response matches or corresponds to an HTTP(S) request if both terms contain the same nonce.

Example 7.

$$\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}})) \qquad (13)$$

$$\mathsf{enc_s}(s, k') \qquad (14)$$

The term (13) shows an encrypted request (with r as in (11)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (14) is a response (with s as in (12)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (13).

E. DNS Messages

Definition 33. A *DNS request* is a term of the form $\langle \mathtt{DNSResolve}, \mathit{domain}, n\rangle$ where $\mathit{domain} \in \mathsf{Doms}$, $n \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$.

Definition 34. A *DNS response* is a term of the form $\langle \mathtt{DNSResolved}, \mathit{domain}, \mathit{result}, n\rangle$ with $\mathit{domain} \in \mathsf{Doms}$, $\mathit{result} \in \mathsf{IPs}$, $n \in \mathcal{N}$. We call the set of all DNS responses $\mathsf{DNSResponses}$.

DNS servers are supposed to include the nonce they received in a DNS request in the DNS response that they send back so that the party which issued the request can match it with the request.

F. DNS Servers

Here, we consider a flat DNS model in which DNS queries are answered directly by one DNS server and always with the same address for a domain. A full (hierarchical) DNS system with recursive DNS resolution, DNS caches, etc. could also be modeled to cover certain attacks on the DNS system itself.

Definition 35. A *DNS server d* (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form

$$s_0^d = \langle\langle \mathtt{domain}_1, a_1\rangle, \langle \mathtt{domain}_2, a_2\rangle, \ldots\rangle \ .$$

DNS queries are answered according to this table (otherwise ignored).

APPENDIX D
DETAILED DESCRIPTION OF THE BROWSER MODEL

Following the informal description of the browser model in Section IV, we now present a formal model. We start by introducing some notation and terminology.

A. Notation and Terminology (Web Browser State)

Before we can define the state of a web browser, we first have to define windows and documents.

Definition 36. A *window* is a term of the form $w = \langle nonce, documents, opener\rangle$ with $nonce \in \mathcal{N}$, $documents \subset^{\langle\rangle} \mathsf{Documents}$ (defined below), $opener \in \mathcal{N} \cup \{\bot\}$ where $d.\mathtt{active} = \top$ for exactly one $d \in^{\langle\rangle} documents$ if *documents* is not empty (we then call d the *active document of* w). We write $\mathsf{Windows}$ for the set of all windows. We write $w.\mathtt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else.

We will refer to the window nonce as *(window) reference*.

The documents contained in a window term to the left of the active document are the previously viewed documents (available to the user via the “back” button) and the documents in the window term to the right of the currently active document are documents available via the “forward” button.

A window a may have opened a top-level window b (i.e., a window term which is not a subterm of a document term). In this case, the *opener* part of the term b is the nonce of a, i.e., $b.\mathtt{opener} = a.\mathtt{nonce}$.

Definition 37. A *document* d is a term of the form

$$\langle nonce, location, headers, referrer, script, scriptstate, scriptinputs, subwindows, active\rangle$$

where $nonce \in \mathcal{N}$, $location \in \mathsf{URLs}$, $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $referrer \in \mathsf{URLs} \cup \{\bot\}$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $subwindows \subset^{\langle\rangle} \mathsf{Windows}$, $active \in \{\top, \bot\}$. A *limited document* is a term of the form $\langle nonce, subwindows\rangle$ with *nonce*, *subwindows* as above. A window $w \in^{\langle\rangle} subwindows$ is called a *subwindow* (of d). We write $\mathsf{Documents}$ for the set of all documents. For a document term d we write $d.\mathtt{origin}$ to denote the origin of the document, i.e., the term $\langle d.\mathtt{location.host}, d.\mathtt{location.protocol}\rangle \in \mathsf{Origins}$.

We will refer to the document nonce as *(document) reference*.

Definition 38. For two window terms w and w' we write $w \xrightarrow{\text{childof}} w'$ if $w \in^{\langle\rangle} w'.\mathtt{activedocument.subwindows}$. We write $\xrightarrow{\text{childof}^+}$ for the transitive closure.

B. Web Browser State

We can now define the set of states of web browsers. Note that we use the dictionary notation that we introduced in Definition 13.

Definition 39. The *set of states $Z_{webbrowser}$ of a web browser atomic Dolev-Yao process* consists of the terms of the form

$$\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping,$$
$$sts, DNSaddress, pendingDNS, pendingRequests, isCorrupted\rangle$$

where

- $windows \subset^{\langle\rangle} \mathsf{Windows}$,
- $ids \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$,
- $secrets \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$,
- *cookies* is a dictionary over Doms and sequences of $\mathsf{Cookies}$,
- $localStorage \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$,
- $sessionStorage \in \left[OR \times \mathcal{T}_{\mathcal{N}}\right]$ for $OR := \{\langle o, r\rangle \mid o \in \mathsf{Origins}, r \in \mathcal{N}\}$,
- $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$,
- $sts \subset^{\langle\rangle} \mathsf{Doms}$,
- $DNSaddress \in \mathsf{IPs}$,
- $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$,
- $pendingRequests \in \mathcal{T}_{\mathcal{N}}$,
- and $isCorrupted \in \{\bot, \mathtt{FULLCORRUPT}, \mathtt{CLOSECORRUPT}\}$.

C. Description of the Web Browser Relation

We will now define the relation $R_{\text{webbrowser}}$ of a standard HTTP browser. We first introduce some notations and then describe the functions that are used for defining the browser main algorithm. We then define the browser relation.

Helper Functions. In the following description of the web browser relation $R_{\text{webbrowser}}$ we use the helper functions $\mathsf{Subwindows}$, Docs, Clean, $\mathsf{CookieMerge}$ and $\mathsf{AddCookie}$.

Subwindows. Given a browser state s, $\mathsf{Subwindows}(s)$ denotes the set of all pointers[13] to windows in the window list $s.\texttt{windows}$, their active documents, and (recursively) the subwindows of these documents. We exclude subwindows of inactive documents and their subwindows. With $\mathsf{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\mathsf{Subwindows}(s)$.

Definition 40. For a browser state s we denote by $\mathsf{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\texttt{windows}$ there is a $\overline{p} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p} = w$. (2) For all $\overline{p} \in \mathsf{Subwindows}(s)$, the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p'} = w$.

Given a browser state s, the set $\mathsf{Docs}(s)$ of pointers to active documents is the minimal set such that for every $\overline{p} \in \mathsf{Subwindows}(s)$, there is a pointer $\overline{p'} \in \mathsf{Docs}(s)$ with $s.\overline{p'} = s.\overline{p}.\texttt{activedocument}$.

By $\mathsf{Subwindows}^{+}(s)$ and $\mathsf{Docs}^{+}(s)$ we denote the respective sets that also include the inactive documents and their subwindows.

Clean. The function Clean will be used to determine which information about windows and documents the script running in the document d has access to.

Definition 41. Let s be a browser state and d a document. By $\mathsf{Clean}(s,d)$ we denote the term that equals $s.\texttt{windows}$ but with (1) all inactive documents removed (including their subwindows etc.), (2) all subterms that represent non-same-origin documents w.r.t. d replaced by a limited document d' with the same nonce and the same subwindow list, and (3) the values of the subterms $\texttt{headers}$ for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.)

CookieMerge. The function $\mathsf{CookieMerge}$ merges two sequences of cookies together: When used in the browser, *oldcookies* is the sequence of existing cookies for some origin, *newcookies* is a sequence of new cookies that was output by some script. The sequences are merged into a set of cookies using an algorithm that is based on the *Storage Mechanism* algorithm described in RFC6265.

Definition 42. For a sequence of cookies (with pairwise different names) *oldcookies* and a sequence of cookies *newcookies*, the set $\mathsf{CookieMerge}(\mathit{oldcookies},\mathit{newcookies})$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\texttt{content.httpOnly} \equiv \top$. For any c, $c' \in^{\langle\rangle} \mathit{newcookies}$, $c.\texttt{name} \equiv c'.\texttt{name}$, remove the cookie that appears left of the other in *newcookies*. Let m be the set of cookies that have a name that either appears in *oldcookies* or in *newcookies*, but not in both. For all pairs of cookies $(c_{\text{old}}, c_{\text{new}})$ with $c_{\text{old}} \in^{\langle\rangle} \mathit{oldcookies}$, $c_{\text{new}} \in^{\langle\rangle} \mathit{newcookies}$, $c_{\text{old}}.\texttt{name} \equiv c_{\text{new}}.\texttt{name}$, add c_{new} to m if $c_{\text{old}}.\texttt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(\mathit{oldcookies},\mathit{newcookies})$ is m.

AddCookie. The function $\mathsf{AddCookie}$ adds a cookie c received in an HTTP response to the sequence of cookies contained in the sequence *oldcookies*. It is again based on the algorithm described in RFC6265 but simplified for the use in the browser model.

Definition 43. For a sequence of cookies (with pairwise different names) *oldcookies* and a cookie c, the sequence $\mathsf{AddCookie}(\mathit{oldcookies},c)$ is defined by the following algorithm: Let $m := \mathit{oldcookies}$. Remove any c' from m that has $c.\texttt{name} \equiv c'.\texttt{name}$. Append c to m and return m.

NavigableWindows. The function $\mathsf{NavigableWindows}$ returns a set of windows that a document is allowed to navigate. We closely follow [29], Section 5.1.4 for this definition.

Definition 44. The set $\mathsf{NavigableWindows}(\overline{w},s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

- If $s'.\overline{w'}.\texttt{activedocument.origin} \equiv s'.\overline{w}.\texttt{activedocument.origin}$ (i.e., the active documents in $\overline{w}$ and $\overline{w'}$ are same-origin), then $\overline{w'} \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w'} \wedge \nexists \overline{w''} \in \mathsf{Subwindows}(s')$ with $s'.\overline{w'} \xrightarrow{\text{childof}^*} s'.\overline{w''}$ ($\overline{w'}$ is a top-level window and $\overline{w}$ is an ancestor window of $\overline{w'}$), then $\overline{w'} \in \overline{W}$, and

[13] Recall the definition of a pointer in Definition 14.

- If $\exists \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'} \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\ s'.\overline{p}.\texttt{activedocument.origin} = s'.\overline{w}.\texttt{activedocument.origin}$ ($\overline{w'}$ is not a top-level window but there is an ancestor window $\overline{p}$ of $\overline{w'}$ with an active document that has the same origin as the active document in $\overline{w}$), then $\overline{w'} \in \overline{W}$, and
- If $\exists \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'}.\texttt{opener} = s'.\overline{p}.\texttt{nonce} \wedge \overline{p} \in \overline{W}$ ($\overline{w'}$ is a top-level window—it has an opener—and $\overline{w}$ is allowed to navigate the opener window of $\overline{w'}$, $\overline{p}$), then $\overline{w'} \in \overline{W}$.

Notations for Functions and Algorithms. We use the following notations to describe the browser algorithms:

Non-deterministic chosing and iteration. The notation **let** $n \leftarrow N$ is used to describe that n is chosen non-deterministically from the set N. We write **for each** $s \in M$ **do** to denote that the following commands (until **end for**) are repeated for every element in M, where the variable s is the current element. The order in which the elements are processed is chosen non-deterministically. We write, for example,

let x, y **such that** $\langle \texttt{Constant}, x, y \rangle \equiv t$ **if possible; otherwise** doSomethingElse

for some variables x, y, a string $\texttt{Constant}$, and some term t to express that $x := \pi_2(t)$, and $y := \pi_3(t)$ if $\texttt{Constant} \equiv \pi_1(t)$ and if $|\langle \texttt{Constant}, x, y \rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

Stop without output. We write **stop** (without further parameters) to denote that there is no output and no change in the state.

Placeholders. In several places throughout the algorithms presented next we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 6). Figure 9 shows a list of all placeholders used.

Functions. In the description of the following functions, we use a, f, m, and s as read-only global input variables. All other variables are local variables or arguments.

- The function $\mathsf{GETNAVIGABLEWINDOW}$ (Algorithm 1) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When it is given a window reference (nonce) *window*, this function returns a pointer to a selected window term in s':
  - If *window* is the string $\texttt{_BLANK}$, a new window is created and a pointer to that window is returned.
  - If *window* is a nonce (reference) and there is a window term with a reference of that value in the windows in s', a pointer $\overline{w'}$ to that window term is returned, as long as the window is navigable by the current window's document (as defined by $\mathsf{NavigableWindows}$ above).

  In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).
- The function $\mathsf{GETWINDOW}$ (Algorithm 2) takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.
- The function $\mathsf{CANCELNAV}$ (Algorithm 3) is used to stop any pending requests for a specific window. From the pending requests and pending DNS requests it removes any requests with the given window reference n.
- The function $\mathsf{HTTP\_SEND}$ (Algorithm 4) takes an HTTP request *message* as input, adds cookie and origin headers to the message, creates a DNS request for the hostname given in the request and stores the request in $s'.\texttt{pendingDNS}$ until the DNS resolution finishes. For normal HTTP requests, *reference* is a window reference. For XHRs, *reference* is a value of the form $\langle \mathit{document}, \mathit{nonce} \rangle$ where *document* is a document reference and *nonce* is some nonce that was chosen by the script that initiated the request. *url* contains the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). *origin* is the origin header value that is to be added to the HTTP request.

Placeholder	Usage
ν_1	Algorithm 9, new window nonces
ν_2	Algorithm 9, new HTTP request nonce
ν_3	Algorithm 9, lookup key for pending HTTP requests entry
ν_4	Algorithm 7, new HTTP request nonce (multiple lines)
ν_5	Algorithm 7, new subwindow nonce
ν_6	Algorithm 8, new HTTP request nonce
ν_7	Algorithm 8, new document nonce
ν_8	Algorithm 4, lookup key for pending DNS entry
ν_9	Algorithm 1, new window nonce
$\nu_{10}, \ldots$	Algorithm 7, replacement for placeholders in script output

Figure 9. List of placeholders used in browser algorithms.

Algorithm 1 Web Browser Model: Determine window for navigation.

```
function GETNAVIGABLEWINDOW($\overline{w}$, $window$, $noreferrer$, $s'$)
    if $window \equiv$ _BLANK then    → Open a new window when _BLANK is used
        if $noreferrer \equiv \top$ then
            let $w' := \langle \nu_9, \langle\rangle, \bot \rangle$
        else
            let $w' := \langle \nu_9, \langle\rangle, s'.\overline{w}.\mathtt{nonce} \rangle$
        let $s'.\mathtt{windows} := s'.\mathtt{windows} +^{\langle\rangle} w'$
          ↪ and let $\overline{w}'$ be a pointer to this new element in $s'$
        return $\overline{w}'$
    let $\overline{w}' \leftarrow$ NavigableWindows($\overline{w}, s'$) such that $s'.\overline{w}'.\mathtt{nonce} \equiv window$
      ↪ if possible; otherwise return $\overline{w}$
    return $\overline{w}'$
```

Algorithm 2 Web Browser Model: Determine same-origin window.

```
function GETWINDOW($\overline{w}$, $window$, $s'$)
    let $\overline{w}' \leftarrow$ Subwindows($s'$) such that $s'.\overline{w}'.\mathtt{nonce} \equiv window$
      ↪ if possible; otherwise return $\overline{w}$
    if $s'.\overline{w}'.\mathtt{activedocument.origin} \equiv s'.\overline{w}.\mathtt{activedocument.origin}$ then
        return $\overline{w}'$
    return $\overline{w}$
```

Algorithm 3 Web Browser Model: Cancel pending requests for given window.

```
function CANCELNAV($n$, $s'$)
    remove all $\langle n, req, key, f \rangle$ from $s'.\mathtt{pendingRequests}$ for any $req$, $key$, $f$
    remove all $\langle x, \langle n, message, url \rangle\rangle$ from $s'.\mathtt{pendingDNS}$
      ↪ for any $x$, $message$, $url$
    return $s'$
```

Algorithm 4 Web Browser Model: Prepare headers, do DNS resolution, save message.

```
function HTTP_SEND($reference$, $message$, $url$, $origin$, $referrer$, $referrerPolicy$, $s'$)
    if $message.\mathtt{host} \in^{\langle\rangle} s'.\mathtt{sts}$ then
        let $url.\mathtt{protocol} := \mathtt{S}$
    let $cookies := \langle \{ \langle c.\mathtt{name}, c.\mathtt{content.value} \rangle | c \in^{\langle\rangle} s'.\mathtt{cookies}[message.\mathtt{host}]$
      ↪ $\wedge (c.\mathtt{content.secure} \implies (url.\mathtt{protocol} = \mathtt{S})) \} \rangle$
    let $message.\mathtt{headers}[\mathtt{Cookie}] := cookies$
    if $origin \not\equiv \bot$ then
        let $message.\mathtt{headers}[\mathtt{Origin}] := origin$
    if $referrerPolicy \equiv \mathtt{noreferrer}$ then
        let $referrer := \bot$
    if $referrer \not\equiv \bot$ then
        if $referrerPolicy \equiv \mathtt{origin}$ then
            let $referrer := \langle \mathtt{URL}, referrer.\mathtt{protocol}, referrer.\mathtt{host}, /, \langle\rangle, \bot \rangle$    → Referrer stripped down to origin.
        let $referrer.\mathtt{fragment} := \bot$    → Browsers do not send fragment identifiers in the Referer header.
        let $message.\mathtt{headers}[\mathtt{Referer}] := referrer$
    let $s'.\mathtt{pendingDNS}[\nu_8] := \langle reference, message, url \rangle$
    stop $\langle \langle s'.\mathtt{DNSaddress}, a, \langle \mathtt{DNSResolve}, message.\mathtt{host}, \nu_8 \rangle \rangle \rangle$, $s'$
```

Algorithm 5 Web Browser Model: Navigate a window backward.

```
function NAVBACK($\overline{w}$, $s'$)
    if $\exists \overline{j} \in \mathbb{N}, \overline{j} > 1$ such that $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} \equiv \top$ then
        let $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} := \bot$
        let $s'.\overline{w'}.\mathtt{documents}.(\overline{j} - 1).\mathtt{active} := \top$
        let $s' :=$ CANCELNAV($s'.\overline{w'}.\mathtt{nonce}, s'$)
```

Algorithm 6 Web Browser Model: Navigate a window forward.

```
function NAVFORWARD($\overline{w}$, $s'$)
    if $\exists\, \overline{j} \in \mathbb{N}$ such that $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} \equiv \top$
            $\hookrightarrow\ \wedge\ s'.\overline{w'}.\mathtt{documents}.(\overline{j}+1) \in \mathsf{Documents}$ then
        let $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} := \bot$
        let $s'.\overline{w'}.\mathtt{documents}.(\overline{j}+1).\mathtt{active} := \top$
        let $s' := \mathsf{CANCELNAV}(s'.\overline{w}'.\mathtt{nonce}, s')$
```

- The functions NAVBACK (Algorithm 5) and NAVFORWARD (Algorithm 6), navigate a window forward or backward. More precisely, they deactivate one document and activate that document's succeeding document or preceding document, respectively. If no such successor/predecessor exists, the functions do not change the state.
- The function RUNSCRIPT (Algorithm 7) performs a script execution step of the script in the document $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted.
- The function PROCESSRESPONSE (Algorithm 8) is responsible for processing an HTTP response (*response*) that was received as the response to a request (*request*) that was sent earlier. In *reference*, either a window or a document reference is given (see explanation for Algorithm 4 above). *requestUrl* contains the URL used when retrieving the document.
  The function first saves any cookies that were contained in the response to the browser state, then checks whether a redirection is requested (Location header). If that is not the case, the function creates a new document (for normal requests) or delivers the contents of the response to the respective receiver (for XHR responses).

Definition. We can now define the *relation $R_{webbrowser}$ of a web browser atomic process* as follows:

Definition 45. The pair $((\langle\langle a,f,m\rangle\rangle,s),(M,s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 9 (or any of the functions called therein), when given $(\langle a,f,m\rangle,s)$ as input, terminates with **stop** M, s', i.e., with output M and s'.

Recall that $\langle a,f,m\rangle$ is an (input) event and s is a (browser) state, M is a sequence of (output) protoevents, and s' is a new (browser) state (potentially with placeholders for nonces).

D. Definition of Web Browsers

Finally, we define web browser atomic Dolev-Yao processes as follows:

Definition 46 (Web Browser atomic Dolev-Yao Process). A web browser atomic Dolev-Yao process is an atomic Dolev-Yao process of the form $p = (I^p, Z_{\text{webbrowser}}, R_{\text{webbrowser}}, {s_o}^p)$ for a set I^p of addresses, $Z_{\text{webbrowser}}$ and $R_{\text{webbrowser}}$ as defined above, and an initial state ${s_0}^p \in Z_{\text{webbrowser}}$.

Algorithm 7 Web Browser Model: Execute a script.

```
function RUNSCRIPT($\overline{w}$, $\overline{d}$, $s'$)
    let $tree$ := $\mathsf{Clean}(s', s'.\overline{d})$
    let $cookies$ := $\langle\{\langle c.\mathtt{name}, c.\mathtt{content.value}\rangle | c \in^{\langle\rangle} s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right]$
      ↪ $\wedge c.\mathtt{content.httpOnly} = \bot$
      ↪ $\wedge \left(c.\mathtt{content.secure} \implies \left(s'.\overline{d}.\mathtt{origin.protocol} \equiv \mathtt{S}\right)\right)\}\rangle$
    let $tlw \leftarrow s'.\mathtt{windows}$ such that $tlw$ is the top-level window containing $\overline{d}$
    let $sessionStorage$ := $s'.\mathtt{sessionStorage}\left[\langle s'.\overline{d}.\mathtt{origin}, tlw.\mathtt{nonce}\rangle\right]$
    let $localStorage$ := $s'.\mathtt{localStorage}\left[s'.\overline{d}.\mathtt{origin}\right]$
    let $secrets$ := $s'.\mathtt{secrets}\left[s'.\overline{d}.\mathtt{origin}\right]$
    let $R \leftarrow \mathsf{script}^{-1}(s'.\overline{d}.\mathtt{script})$
    let $in$ := $\langle tree$, $s'.\overline{d}.\mathtt{nonce}, s'.\overline{d}.\mathtt{scriptstate}$, $s'.\overline{d}.\mathtt{scriptinputs}$, $cookies$,
      ↪ $localStorage$, $sessionStorage$, $s'.\mathtt{ids}$, $secrets\rangle$
    let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
      ↪ $cookies' \leftarrow \mathsf{Cookies}^{\nu}$,
      ↪ $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
      ↪ $sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
      ↪ $command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
      ↪ $out^{\lambda} := \langle state', cookies', localStorage', sessionStorage', command\rangle$
      ↪ such that $(in, out^{\lambda}) \in R$
    let $out$ := $out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$
    let $s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right]$
      ↪ := $\langle \mathsf{CookieMerge}(s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right]$, $cookies')\rangle$
    let $s'.\mathtt{localStorage}\left[s'.\overline{d}.\mathtt{origin}\right]$ := $localStorage'$
    let $s'.\mathtt{sessionStorage}\left[\langle s'.\overline{d}.\mathtt{origin}, tlw.\mathtt{nonce}\rangle\right]$ := $sessionStorage'$
    let $s'.\overline{d}.\mathtt{scriptstate}$ := $state'$
    switch $command$ do
        case $\langle\mathtt{HREF}, url, hrefwindow, noreferrer\rangle$
            let $\overline{w}'$ := GETNAVIGABLEWINDOW($\overline{w}$, $hrefwindow$, $noreferrer$, $s'$)
            let $req$ := $\langle\mathtt{HTTPReq}, \nu_4, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, url.\mathtt{parameters}, \langle\rangle\rangle$
            if $noreferrer \equiv \top$ then
                let $referrerPolicy$ := $\mathtt{noreferrer}$
            else
                let $referrerPolicy$ := $s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
            let $s'$ := CANCELNAV($s'.\overline{w}'.\mathtt{nonce}, s'$)
            call HTTP_SEND($s'.\overline{w}'.\mathtt{nonce}$, $req$, $url$, $\bot$, $referrer$, $referrerPolicy$, $s'$)
        case $\langle\mathtt{IFRAME}, url, window\rangle$
            let $\overline{w}'$ := GETWINDOW($\overline{w}, window, s'$)
            let $req$ := $\langle\mathtt{HTTPReq}, \nu_4, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, url.\mathtt{parameters}, \langle\rangle\rangle$
            let $referrer$ := $s'.\overline{w}'.\mathtt{activedocument.location}$
            let $referrerPolicy$ := $s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
            let $w'$ := $\langle\nu_5, \langle\rangle, \bot\rangle$
            let $s'.\overline{w}'.\mathtt{activedocument.subwindows}$
              ↪ := $s'.\overline{w}'.\mathtt{activedocument.subwindows} +^{\langle\rangle} w'$
            call HTTP_SEND($\nu_5$, $req$, $url$, $\bot$, $referrer$, $referrerPolicy$, $s'$)
        case $\langle\mathtt{FORM}, url, method, data, hrefwindow\rangle$
            if $method \notin \{\mathtt{GET}, \mathtt{POST}\}$ then [14]
                stop
            let $\overline{w}'$ := GETNAVIGABLEWINDOW($\overline{w}$, $hrefwindow$, $\bot$, $s'$)
            if $method = \mathtt{GET}$ then
                let $body$ := $\langle\rangle$
                let $parameters$ := $data$
                let $origin$ := $\bot$
            else
                let $body$ := $data$
                let $parameters$ := $url.\mathtt{parameters}$
                let $origin$ := $s'.\overline{d}.\mathtt{origin}$
            let $req$ := $\langle\mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, parameters, body\rangle$
            let $referrer$ := $s'.\overline{d}.\mathtt{location}$
            let $referrerPolicy$ := $s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
            let $s'$ := CANCELNAV($s'.\overline{w}'.\mathtt{nonce}, s'$)
            call HTTP_SEND($s'.\overline{w}'.\mathtt{nonce}$, $req$, $url$, $origin$, $referrer$, $referrerPolicy$, $s'$)
    case $\langle \texttt{SETSCRIPT}, window, script \rangle$
        let $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
        let $s'.\overline{w}'.\texttt{activedocument}.\texttt{script} := script$
        stop $\langle\rangle$, $s'$
    case $\langle \texttt{SETSCRIPTSTATE}, window, scriptstate \rangle$
        let $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
        let $s'.\overline{w}'.\texttt{activedocument}.\texttt{scriptstate} := scriptstate$
        stop $\langle\rangle$, $s'$
    case $\langle \texttt{XMLHTTPREQUEST}, url, method, data, xhrreference \rangle$
        if $method \in \{\texttt{CONNECT}, \texttt{TRACE}, \texttt{TRACK}\} \wedge xhrreference \not\in \{\mathcal{N}, \bot\}$ then
            stop
        if $url.\texttt{host} \not\equiv s'.\overline{d}.\texttt{origin}.\texttt{host}$
         $\hookrightarrow$ $\vee\ url \not\equiv s'.\overline{d}.\texttt{origin}.\texttt{protocol}$ then
            stop
        if $method \in \{\texttt{GET}, \texttt{HEAD}\}$ then
            let $data := \langle\rangle$
            let $origin := \bot$
        else
            let $origin := s'.\overline{d}.\texttt{origin}$
        let $req := \langle \texttt{HTTPReq}, \nu_4, method, url.\texttt{host}, url.\texttt{path}, , url.\texttt{parameters}, data \rangle$
        let $referrer := s'.\overline{d}.\texttt{location}$
        let $referrerPolicy := s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
        call $\mathsf{HTTP\_SEND}(\langle s'.\overline{d}.\texttt{nonce}, xhrreference \rangle, req, url, origin, referrer, referrerPolicy, s')$
    case $\langle \texttt{BACK}, window \rangle$ [15]
        let $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, window, \bot, s')$
        $\mathsf{NAVBACK}(\overline{w}, s')$
        stop $\langle\rangle$, $s'$
    case $\langle \texttt{FORWARD}, window \rangle$
        let $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, window, \bot, s')$
        $\mathsf{NAVFORWARD}(\overline{w}, s')$
        stop $\langle\rangle$, $s'$
    case $\langle \texttt{CLOSE}, window \rangle$
        let $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, window, \bot, s')$
        remove $s'.\overline{w'}$ from the sequence containing it
        stop $\langle\rangle$, $s'$
    case $\langle \texttt{POSTMESSAGE}, window, message, origin \rangle$
        let $\overline{w}' \leftarrow \mathsf{Subwindows}(s')$ such that $s'.\overline{w}'.\texttt{nonce} \equiv window$
        if $\exists \overline{j} \in \mathbb{N}$ such that $s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{active} \equiv \top$
         $\hookrightarrow$ $\wedge (origin \not\equiv \bot \implies s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{origin} \equiv origin)$ then
            let $s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
             $\hookrightarrow$ $:= s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
             $\hookrightarrow$ $+^{\langle\rangle} \langle \texttt{POSTMESSAGE}, s'.\overline{w}.\texttt{nonce}, s'.\overline{d}.\texttt{origin}, message \rangle$
        stop $\langle\rangle$, $s'$
    case else
        stop
```

Algorithm 8 Web Browser Model: Process an HTTP response.

```
function PROCESSRESPONSE($response$, $reference$, $request$, $requestUrl$, $s'$)
    if Set-Cookie $\in$ $response$.headers then
        for each $c \in^{\langle\rangle}$ $response$.headers[Set-Cookie], $c \in$ Cookies do
            let $s'$.cookies[$request$.host]
              $\hookrightarrow$ := AddCookie($s'$.cookies[$request$.host], $c$)
    if Strict-Transport-Security $\in$ $response$.headers $\wedge$ $requestUrl$.protocol $\equiv$ S then
        let $s'$.sts := $s'$.sts $+^{\langle\rangle}$ $request$.host
    if Referer $\in$ $request$.headers then
        let $referrer$ := $request$.headers[Referer]
    else
        let $referrer$ := $\perp$
    if Location $\in$ $response$.headers $\wedge$ $response$.status $\in \{303, 307\}$ then
        let $url$ := $response$.headers[Location]
        if $url$.fragment $\equiv \perp$ then
            let $url$.fragment := $requestUrl$.fragment
        let $method'$ := $request$.method
        let $body'$ := $request$.body
        if Origin $\in$ $request$.headers then
            let $origin$ := $\langle$$request$.headers[Origin], $\langle$$request$.host, $url$.protocol$\rangle\rangle$
        else
            let $origin$ := $\perp$
        if $response$.status $\equiv 303 \wedge$ $request$.method $\not\in$ {GET, HEAD} then
            let $method'$ := GET
            let $body'$ := $\langle\rangle$
        if $\exists \overline{w} \in$ Subwindows($s'$) such that $s'.\overline{w}$.nonce $\equiv$ $reference$ then    $\rightarrow$ Do not redirect XHRs.
            let $req$ := $\langle$HTTPReq, $\nu_6$, $method'$, $url$.host, $url$.path, $\langle\rangle$, $url$.parameters, $body'\rangle$
            let $referrerPolicy$ := $response$.headers[ReferrerPolicy]
            call HTTP_SEND($reference$, $req$, $url$, $origin$, $referrer$, $referrerPolicy$, $s'$)
    if $\exists \overline{w} \in$ Subwindows($s'$) such that $s'.\overline{w}$.nonce $\equiv$ $reference$ then    $\rightarrow$ normal response
        if $response$.body $\not\sim \langle *, * \rangle$ then
            stop $\{\}$, $s'$
        let $script$ := $\pi_1$($response$.body)
        let $scriptstate$ := $\pi_2$($response$.body)
        let $referrer$ := $request$.headers[Referer]
        let $d$ := $\langle \nu_7$, $requestUrl$, $response$.headers, $referrer$, $script$, $scriptstate$, $\langle\rangle, \langle\rangle, \top\rangle$
        if $s'.\overline{w}$.documents $\equiv \langle\rangle$ then
            let $s'.\overline{w}$.documents := $\langle d \rangle$
        else
            let $\overline{i} \leftarrow \mathbb{N}$ such that $s'.\overline{w}$.documents.$\overline{i}$.active $\equiv \top$
            let $s'.\overline{w}$.documents.$\overline{i}$.active := $\perp$
            remove $s'.\overline{w}$.documents.$(\overline{i}+1)$ and all following documents
              $\hookrightarrow$ from $s'.\overline{w}$.documents
            let $s'.\overline{w}$.documents := $s'.\overline{w}$.documents $+^{\langle\rangle}$ $d$
        stop $\{\}$, $s'$
    else if $\exists \overline{w} \in$ Subwindows($s'$), $\overline{d}$ such that $s'.\overline{d}$.nonce $\equiv \pi_1$($reference$)
      $\hookrightarrow$ $\wedge$ $s'.\overline{d} = s'.\overline{w}$.activedocument then    $\rightarrow$ process XHR response
        let $headers$ := $response$.headers $-$ Set-Cookie
        let $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs $+^{\langle\rangle}$
            $\langle$XMLHTTPREQUEST, $headers$, $response$.body, $\pi_2$($reference$)$\rangle$
```

Algorithm 9 Web Browser Model: Main Algorithm

```
Input: $\langle a, f, m\rangle, s$
let $s' := s$
if $s.\mathtt{isCorrupted} \not\equiv \bot$ then
    let $s'.\mathtt{pendingRequests} := \langle m, s.\mathtt{pendingRequests}\rangle$    $\rightarrow$ Collect incoming messages
    let $m' \leftarrow d_V(s')$
    let $a' \leftarrow \mathsf{IPs}$
    stop $\langle\langle a', a, m'\rangle\rangle$, $s'$
if $m \equiv \mathtt{TRIGGER}$ then    $\rightarrow$ A special trigger message.
    let $switch \leftarrow \{\mathtt{script}, \mathtt{urlbar}, \mathtt{reload}, \mathtt{forward}, \mathtt{back}\}$
    let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ such that $s'.\overline{w}.\mathtt{documents} \neq \langle\rangle$
        $\hookrightarrow$ if possible; otherwise stop    $\rightarrow$ Pointer to some window.
    let $\overline{tlw} \leftarrow \mathbb{N}$ such that $s'.\overline{tlw}.\mathtt{documents} \neq \langle\rangle$
        $\hookrightarrow$ if possible; otherwise stop    $\rightarrow$ Pointer to some top-level window.
    if $switch \equiv \mathtt{script}$ then    $\rightarrow$ Run some script.
        let $\overline{d} := \overline{w} +^{\langle\rangle} \mathtt{activedocument}$
        call $\mathsf{RUNSCRIPT}(\overline{w}, \overline{d}, s')$
    else if $switch \equiv \mathtt{urlbar}$ then    $\rightarrow$ Create some new request.
        let $newwindow \leftarrow \{\top, \bot\}$
        if $newwindow \equiv \top$ then    $\rightarrow$ Create a new window.
            let $windownonce := \nu_1$
            let $w' := \langle windownonce, \langle\rangle, \bot\rangle$
            let $s'.\mathtt{windows} := s'.\mathtt{windows} +^{\langle\rangle} w'$
        else    $\rightarrow$ Use existing top-level window.
            let $windownonce := s'.\overline{tlw}.nonce$
        let $protocol \leftarrow \{\mathtt{P}, \mathtt{S}\}$
        let $host \leftarrow \mathsf{Doms}$
        let $path \leftarrow \mathbb{S}$
        let $fragment \leftarrow \mathbb{S}$
        let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
        let $url := \langle \mathtt{URL}, protocol, host, path, parameters, fragment\rangle$
        let $req := \langle \mathtt{HTTPReq}, \nu_2, \mathtt{GET}, host, path, \langle\rangle, parameters, \langle\rangle\rangle$
        call $\mathsf{HTTP\_SEND}(windownonce, req, url, \bot, \bot, \bot, s')$
    else if $switch \equiv \mathtt{reload}$ then    $\rightarrow$ Reload some document.
        let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ such that $s'.\overline{w}.\mathtt{documents} \neq \langle\rangle$
            $\hookrightarrow$ if possible; otherwise stop
        let $url := s'.\overline{w}.\mathtt{activedocument}.\mathtt{location}$
        let $req := \langle \mathtt{HTTPReq}, \nu_2, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, url.\mathtt{parameters}, \langle\rangle\rangle$
        let $referrer := s'.\overline{w}.\mathtt{activedocument}.\mathtt{referrer}$
        let $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\mathtt{nonce}, s')$
        call $\mathsf{HTTP\_SEND}(s'.\overline{w}.\mathtt{nonce}, req, url, \bot, referrer, \bot, s')$
    else if $switch \equiv \mathtt{forward}$ then
        $\mathsf{NAVFORWARD}(\overline{w}, s')$
    else if $switch \equiv \mathtt{back}$ then
        $\mathsf{NAVBACK}(\overline{w}, s')$
else if $m \equiv \mathtt{FULLCORRUPT}$ then    $\rightarrow$ Request to corrupt browser
    let $s'.\mathtt{isCorrupted} := \mathtt{FULLCORRUPT}$
    stop $\langle\rangle$, $s'$
else if $m \equiv \mathtt{CLOSECORRUPT}$ then    $\rightarrow$ Close the browser
    let $s'.\mathtt{secrets} := \langle\rangle$
    let $s'.\mathtt{windows} := \langle\rangle$
    let $s'.\mathtt{pendingDNS} := \langle\rangle$
    let $s'.\mathtt{pendingRequests} := \langle\rangle$
    let $s'.\mathtt{sessionStorage} := \langle\rangle$
    let $s'.\mathtt{cookies} \subset^{\langle\rangle} \mathsf{Cookies}$ such that
      $\hookrightarrow$ $(c \in^{\langle\rangle} s'.\mathtt{cookies}) \iff (c \in^{\langle\rangle} s.\mathtt{cookies} \wedge c.\mathtt{content}.\mathtt{session} \equiv \bot)$
    let $s'.\mathtt{isCorrupted} := \mathtt{CLOSECORRUPT}$
    stop $\langle\rangle$, $s'$
```

```
else if $\exists \langle reference, request, url, key, f \rangle \in^{\langle\rangle} s'.\mathtt{pendingRequests}$
  ↪ such that $\pi_1(\mathsf{dec_s}(m, key)) \equiv \mathtt{HTTPResp}$ then    → Encrypted HTTP response
    let $m' := \mathsf{dec_s}(m, key)$
    if $m'.\mathtt{nonce} \not\equiv request.\mathtt{nonce}$ then
        stop
    remove $\langle reference, request, url, key, f \rangle$ from $s'.\mathtt{pendingRequests}$
    call $\mathsf{PROCESSRESPONSE}(m', reference, request, url, s')$
else if $\pi_1(m) \equiv \mathtt{HTTPResp} \wedge \exists \langle reference, request, url, \bot, f \rangle \in^{\langle\rangle} s'.\mathtt{pendingRequests}$
  ↪ such that $m'.\mathtt{nonce} \equiv request.\mathtt{key}$ then
    remove $\langle reference, request, url, \bot, f \rangle$ from $s'.\mathtt{pendingRequests}$
    call $\mathsf{PROCESSRESPONSE}(m, reference, request, url, s')$
else if $m \in \mathsf{DNSResponses}$ then    → Successful DNS response
    if $m.\mathtt{nonce} \not\in s.\mathtt{pendingDNS} \vee m.\mathtt{result} \not\in \mathsf{IPs} \vee m.\mathtt{domain} \not\equiv \pi_2(s.\mathtt{pendingDNS}).\mathtt{host}$ then
        stop
    let $\langle reference, message, url \rangle := s.\mathtt{pendingDNS}[m.\mathtt{nonce}]$
    if $url.\mathtt{protocol} \equiv \mathsf{S}$ then
        let $s'.\mathtt{pendingRequests} := s'.\mathtt{pendingRequests}$
          ↪ $+^{\langle\rangle} \langle reference, message, url, v_3, m.\mathtt{result} \rangle$
        let $message := \mathsf{enc_a}(\langle message, v_3 \rangle, s'.\mathtt{keyMapping}[message.\mathtt{host}])$
    else
        let $s'.\mathtt{pendingRequests} := s'.\mathtt{pendingRequests}$
          ↪ $+^{\langle\rangle} \langle reference, message, url, \bot, m.\mathtt{result} \rangle$
    let $s'.\mathtt{pendingDNS} := s'.\mathtt{pendingDNS} - m.\mathtt{nonce}$
    stop $\langle \langle m.\mathtt{result}, a, message \rangle \rangle$, $s'$
else    → Some other message
    call $\mathsf{PROCESS\_OTHER}(m, a, f, s')$
stop
```

Appendix E
Generic HTTPS Server Model

This model will be used as the base for all servers in the following. It makes use of placeholder algorithms that are later superseded by more detailed algorithms to describe a concrete relation for an HTTPS server.

Definition 47 (Base state for an HTTPS server.). The state of each HTTPS server that is an instantiation of this relation must contain at least the following subterms: $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{pendingRequests} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ (both containing arbitrary terms), $\mathit{DNSaddress} \in \mathsf{IPs}$ (containing the IP address of a DNS server), $\mathit{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (containing a mapping from domains to public keys), $\mathit{tlskeys} \in [\mathsf{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$ (either $\perp$ if the server is not corrupted, or an arbitrary term otherwise).

We note that in concrete instantiations of the generic HTTPS server model, there is no need to extract information from these subterms or alter these subterms.

Let ν_{n0} and ν_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. We now define the default functions of the generic web server in Algorithms 10–14, and the main relation in Algorithm 15.

Algorithm 10 Generic HTTPS Server Model: Sending a DNS message (in preparation for sending an HTTPS message).

```
function $\mathsf{HTTPS\_SIMPLE\_SEND}(\mathit{reference}, \mathit{message}, s')$
    let $s'.\mathtt{pendingDNS}[\nu_{n0}] := \langle \mathit{reference}, \mathit{message} \rangle$
    stop $\langle\langle s'.\mathtt{DNSaddress}, a, \langle \mathtt{DNSResolve}, \mathit{message}.\mathtt{host}, \nu_{n0} \rangle\rangle\rangle$, $s'$
```

Algorithm 11 Generic HTTPS Server Model: Default HTTPS response handler.

```
function $\mathsf{PROCESS\_HTTPS\_RESPONSE}(m, \mathit{reference}, \mathit{request}, \mathit{key}, a, f, s')$
    stop
```

Algorithm 12 Generic HTTPS Server Model: Default trigger event handler.

```
function $\mathsf{TRIGGER}(s')$
    stop
```

Algorithm 13 Generic HTTPS Server Model: Default HTTPS request handler.

```
function $\mathsf{PROCESS\_HTTPS\_REQUEST}(m, k, a, f, s')$
    stop
```

Algorithm 14 Generic HTTPS Server Model: Default handler for other messages.

```
function $\mathsf{PROCESS\_OTHER}(m, a, f, s')$
    stop
```

Algorithm 15 Generic HTTPS Server Model: Main relation of a generic HTTPS server

```
Input: $\langle a, f, m\rangle, s$
if $s'.\mathtt{corrupt} \not\equiv \bot \vee m \equiv \mathtt{CORRUPT}$ then
    let $s'.\mathtt{corrupt} := \langle\langle a, f, m\rangle, s'.\mathtt{corrupt}\rangle$
    let $m' \leftarrow d_V(s')$
    let $a' \leftarrow \mathsf{IPs}$
    stop $\langle\langle a', a, m'\rangle\rangle$, $s'$
if $\exists m_{\text{dec}}$, $k$, $k'$, $\mathit{inDomain}$ such that $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\mathtt{tlskeys}$ then
    let $n$, $\mathit{method}$, $\mathit{path}$, $\mathit{parameters}$, $\mathit{headers}$, $\mathit{body}$ such that
  $\hookrightarrow$ $\langle \mathtt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
  $\hookrightarrow$ if possible; otherwise stop
    call PROCESS_HTTPS_REQUEST($m_{\text{dec}}$, $k$, $a$, $f$, $s'$)
else if $m \in \mathsf{DNSResponses}$ then    $\rightarrow$ Successful DNS response
    if $m.\mathtt{nonce} \not\in s.\mathtt{pendingDNS} \vee m.\mathtt{result} \not\in \mathsf{IPs} \vee m.\mathtt{domain} \not\equiv s.\mathtt{pendingDNS}[m.\mathtt{nonce}].2.\mathtt{host}$ then
        stop
    let $\langle \mathit{reference}, \mathit{request}\rangle := s.\mathtt{pendingDNS}[m.\mathtt{nonce}]$
    let $s'.\mathtt{pendingRequests} := s'.\mathtt{pendingRequests}$
        $\hookrightarrow$ $+^{\langle\rangle} \langle \mathit{reference}, \mathit{request}, \nu_{n1}, m.\mathtt{result}\rangle$
    let $\mathit{message} := \mathsf{enc_a}(\langle \mathit{request}, \nu_{n1}\rangle, s'.\mathtt{keyMapping}[\mathit{request}.\mathsf{host}])$
    let $s'.\mathtt{pendingDNS} := s'.\mathtt{pendingDNS} - m.\mathtt{nonce}$
    stop $\langle\langle m.\mathtt{result}, a, \mathit{message}\rangle\rangle$, $s'$
else if $\exists \langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\mathtt{pendingRequests}$
  $\hookrightarrow$ such that $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \mathtt{HTTPResp}$ then    $\rightarrow$ Encrypted HTTP response
    let $m' := \mathsf{dec_s}(m, \mathit{key})$
    if $m'.\mathtt{nonce} \not\equiv \mathit{request}.\mathtt{nonce}$ then
        stop
    remove $\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle$ from $s'.\mathtt{pendingRequests}$
    call PROCESS_HTTPS_RESPONSE($m'$, $\mathit{reference}$, $\mathit{request}$, $\mathit{key}$, $a$, $f$, $s'$)
    stop
else if $m \equiv \mathtt{TRIGGER}$ then    $\rightarrow$ Process was triggered
    call PROCESS_TRIGGER($s'$)
stop
```

Appendix F
Formal Model of OpenID Connect with a Network Attacker

We here present the full details of our formal model of OIDC which we use to analyze the authentication and authorization properties. This model contains a network attacker. We will later derive from this model a model where the network attacker is replaced by a web attacker. We use this modified model for the session integrity properties.

We model OIDC as a web system (in the sense of Appendix B-C). We call a web system $\mathcal{OIDC}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ an *OIDC web system with a network attacker* if it is of the form described in what follows.

A. Outline

The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker process (in Net), a finite set B of web browsers, a finite set RP of web servers for the relying parties, a finite set IDP of web servers for the identity providers, with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP}$. More details on the processes in $\mathcal{W}$ are provided below. We do not model DNS servers, as they are subsumed by the network attacker. Figure 10 shows the set of scripts $\mathcal{S}$ and their respective string representations that are defined by the mapping script. The set E^0 contains only the trigger events as specified in Appendix B-C.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
script_rp_index	`script_rp_index`
script_rp_get_fragment	`script_get_fragment`
script_idp_form	`script_idp_form`

Figure 10. List of scripts in $\mathcal{S}$ and their respective string representations.

This outlines $\mathcal{OIDC}^n$. We will now define the DY processes in $\mathcal{OIDC}^n$ and their addresses, domain names, and secrets in more detail.

B. Addresses and Domain Names

The set IPs contains for the network attacker in Net, every relying party in RP, every identity provider in IDP, and every browser in B a finite set of addresses each. By addr we denote the corresponding assignment from a process to its address. The set Doms contains a finite set of domains for every relying party in RP, every identity provider in IDP, and the network attacker in Net. Browsers (in B) do not have a domain.

By addr and dom we denote the assignments from atomic processes to sets of IPs and Doms, respectively.

C. Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into five sets, an infinite sequence N, an finite set K_{TLS}, an finite set K_{sign}, and a finite set $\mathsf{Passwords}$. We therefore have $\mathcal{N} = \underbrace{N}_{\text{infinite sequence}} \dot\cup \underbrace{K_{\text{TLS}}}_{\text{finite}} \dot\cup \underbrace{K_{\text{sign}}}_{\text{finite}} \dot\cup \underbrace{\mathsf{Passwords}}_{\text{finite}}$.

These sets are used as follows:

- The set N contains the nonces that are available for each DY process in $\mathcal{W}$ (it can be used to create a run of $\mathcal{W}$).
- The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $tlskeys^p = \langle \{ \langle d, \mathsf{tlskey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle$.
- The set K_{sign} contains the keys that will be used by IdPs for signing id tokens. Let $\mathsf{signkey}\colon \mathsf{IDP} \to K_{\text{sign}}$ be an injective mapping that assigns a (different) signing key to every IdP.
- The set $\mathsf{Passwords}$ is the set of passwords (secrets) the browsers share with the identity providers. These are the passwords the users use to log in at the IdPs.

D. Identities and Passwords

Identites consist, similar to email addresses, of a user name and a domain part. For our model, this is defined as follows:

Definition 48. An *identity* (email address) i is a term of the form $\langle name, domain \rangle$ with $name \in \mathbb{S}$ and $domain \in \mathsf{Doms}$.

Let ID be the finite set of identities. We say that an ID is *governed* by the DY process to which the domain of the ID belongs. Formally, we define the mapping $\mathsf{governor}: \mathsf{ID} \to \mathcal{W}$, $\langle name, domain \rangle \mapsto \mathsf{dom}^{-1}(domain)$. By ID^y we denote the set $\mathsf{governor}^{-1}(y)$.

The governor of an ID will usually be an IdP, but could also be the attacker. Besides governor, we define the following mappings:

- By $\mathsf{secretOfID} : \mathsf{ID} \to \mathsf{Passwords}$ we denote the bijective mapping that assigns secrets to all identities.
- Let $\mathsf{ownerOfSecret} : \mathsf{Passwords} \to \mathsf{B}$ denote the mapping that assigns to each secret a browser that *owns* this secret. Now, we define the mapping $\mathsf{ownerOfID} : \mathsf{ID} \to \mathsf{B}$, $i \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which assigns to each identity the browser that owns this identity (we say that the identity belongs to the browser).

E. Corruption

RPs and IdPs can become corrupted: If they receive the message $\mathtt{CORRUPT}$, they start collecting all incoming messages in their state and (upon triggering) send out all messages that are derivable from their state and collected input messages, just like the attacker process. We say that an RP or an IdP is *honest* if the according part of their state ($s.\mathtt{corrupt}$) is $\perp$, and that they are corrupted otherwise.

We are now ready to define the processes in $\mathcal{W}$ as well as the scripts in $\mathcal{S}$ in more detail.

F. Network Attackers

As mentioned, the network attacker na is modeled to be a network attacker as specified in Appendix B-C. We allow it to listen to/spoof all available IP addresses, and hence, define $I^{na} = \mathsf{IPs}$. The initial state is $s_0^{na} = \langle attdoms, tlskeys, signkeys \rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker *na*, *tlskeys* is a sequence of all domains and the corresponding public keys, and *signkeys* is a sequence containing all public signing keys for all IdPs.

G. Browsers

Each $b \in \mathsf{B}$ is a web browser atomic Dolev-Yao process as defined in Definition 46, with $I^b := \mathsf{addr}(b)$ being its addresses.

To define the inital state, first let $\mathsf{ID}_b := \mathsf{ownerOfID}^{-1}(b)$ be the set of all IDs of b. We then define the set of passwords that a browser b gives to an origin o: If the origin belongs to an IdP, then the user's passwords of this IdP are contained in the set. To define this mapping in the initial state, we first define for some process p

$$\mathsf{Secrets}^{b,p} = \left\{ s \;\middle|\; b = \mathsf{ownerOfSecret}(s) \wedge (\exists\, i : s = \mathsf{secretOfID}(i) \wedge i \in \mathsf{ID}^p) \right\}.$$

Then, the initial state s_0^b is defined as follows: the key mapping maps every domain to its public (TLS) key, according to the mapping tlskey; the DNS address is an address of the network attacker; the list of secrets contains an entry $\langle \langle d, \mathsf{S} \rangle, \langle \mathsf{Secrets}^{b,p} \rangle \rangle$ for each $p \in \mathsf{RP} \cup \mathsf{IDP}$ and $d \in \mathsf{dom}(p)$; *ids* is $\langle \mathsf{ID}_b \rangle$; *sts* is empty.

H. Relying Parties

A relying party $r \in \mathsf{RP}$ is a web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the addresses $I^r := \mathsf{addr}(r)$. Next, we define the set Z^r of states of r and the initial state s_0^r of r.

Definition 49. A *state* $s \in Z^r$ *of an RP* r is a term of the form $\langle$*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*, *sessions*, *issuerCache*, *oidcConfigCache*, *jwksCache*, *clientCredentialsCache*$\rangle$ with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $tlskeys \in [\mathsf{Doms} \times K_{\text{TLS}}]$ (all former components as in Definition 47), $sessions \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $issuerCache \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $oidcConfigCache \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, and $jwksCache \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$.

An *initial state* s_0^r *of* r is a state of r with $s_0^r.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^r.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^r.\mathtt{corrupt} \equiv \perp$, $s_0^r.\mathtt{keyMapping}$ being the same as the keymapping for browsers above, $s_0^r.\mathtt{tlskeys} \equiv tlskeys^r$, $s_0^r.\mathtt{sessions} \equiv \langle\rangle$, $s_0^r.\mathtt{issuerCache} \equiv \langle\rangle$, $s_0^r.\mathtt{oidcConfigCache} \equiv \langle\rangle$, $s_0^r.\mathtt{jwksCache} \equiv \langle\rangle$, and $s_0^r.\mathtt{clientCredentialsCache} \equiv \langle\rangle$.

We now specify the relation R^r: This relation is based on our model of generic HTTPS servers (see Appendix E). Hence we only need to specify algorithms that differ from or do not exist in the generic server model. These algorithms are defined in Algorithms 16–21. (Note that in several places throughout these algorithms we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 6). Figure 11 shows a list of all placeholders used.)

The scripts that are used by the RP are described in Algorithms 22 and 23. In these scripts, to extract the current URL of a document, the function $\mathsf{GETURL}(tree, docnonce)$ is used. We define this function as follows: It searches for the document with the identifier *docnonce* in the (cleaned) tree *tree* of the browser's windows and documents. It then returns the URL u of that document. If no document with nonce *docnonce* is found in the tree *tree*, $\diamond$ is returned.

Algorithm 16 Relation of a Relying Party R^r – Processing HTTPS Responses

```
function PROCESS_HTTPS_RESPONSE($m$, $reference$, $request$, $key$, $a$, $f$, $s'$)
    let $session := s'.\texttt{sessions}[reference[\texttt{session}]]$
    let $id := session[\texttt{identity}]$
    let $issuer := s'.\texttt{issuerCache}[id]$
    if $reference[\texttt{responseTo}] \equiv \texttt{WEBFINGER}$ then
        let $wf := m.\texttt{body}$
        if $wf[\texttt{subject}] \not\equiv id$ then
            stop
        if $wf[\texttt{links}][\texttt{rel}] \not\equiv \texttt{OIDC\_issuer}$ then
            stop
        let $s'.\texttt{issuerCache}[id] := wf[\texttt{links}][\texttt{href}]$
        call START_LOGIN_FLOW($reference[\texttt{session}]$, $s'$)
    else if $reference[\texttt{responseTo}] \equiv \texttt{CONFIG}$ then
        let $oidcc := m.\texttt{body}$
        if $oidcc[\texttt{issuer}] \not\equiv issuer$ then
            stop
        let $s'.\texttt{oidcConfigCache}[issuer] := oidcc$
        call START_LOGIN_FLOW($reference[\texttt{session}]$, $s'$)
    else if $reference[\texttt{responseTo}] \equiv \texttt{JWKS}$ then
        let $s'.\texttt{jwksCache}[issuer] := m.\texttt{body}$
        call START_LOGIN_FLOW($reference[\texttt{session}]$, $s'$)
    else if $reference[\texttt{responseTo}] \equiv \texttt{REGISTRATION}$ then
        let $s'.\texttt{clientCredentialsCache}[issuer] := m.\texttt{body}$
        call START_LOGIN_FLOW($reference[\texttt{session}]$, $s'$)
    else if $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$ then
        if $\texttt{token} \in^{\langle\rangle} session[\texttt{response\_type}] \wedge useAccessTokenNow \equiv \top$ then
            call USE_ACCESS_TOKEN($reference[\texttt{session}]$, $m.\texttt{body}[\texttt{access\_token}]$, $s'$)
        call CHECK_ID_TOKEN($reference[\texttt{session}]$, $m.\texttt{body}[\texttt{id\_token}]$, $s'$)
    stop
```

Placeholder	Usage
v_1	new login session id
v_2	new HTTP request nonce
v_3	new HTTP request nonce
v_4	new service session id
v_5	new HTTP request nonce
v_6	new state value
v_7	new *nonce* value (for the implicit flow)

Figure 11. List of placeholders used in the relying party algorithm.

Algorithm 17 Relation of a Relying Party R^r – Processing HTTPS Requests

```
function PROCESS_HTTPS_REQUEST($m$, $k$, $a$, $f$, $s'$)   → Process an incoming HTTPS request. Other message types are handled in separate functions. $m$ is the incoming message, $k$ is the encryption key for the response, $a$ is the receiver, $f$ the sender of the message. $s'$ is the current state of the atomic DY process $r$.
    if $m$.path $\equiv$ / then   → Serve index page.
        let $headers := [$ReferrerPolicy:origin$]$   → Set the Referrer Policy for the index page of the RP (cf. Section III-A).
        let $m' := \mathsf{enc_s}(\langle$HTTPResp, $m$.nonce, 200, $headers$, $\langle$script_rp_index, $\langle\rangle\rangle\rangle, k)$   → Send script_rp_index in HTTP response.
        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
    else if $m$.path $\equiv$ /startLogin $\wedge$ $m$.method $\equiv$ POST then   → Serve the request to start a new login.
        if $m$.headers[Origin] $\not\equiv \langle m$.host, S$\rangle$ then
            stop   → Check the Origin header for CSRF protection.
        let $id := m$.body
        let $sessionId := \nu_1$   → Session id is a freshly chosen nonce.
        let $s'$.sessions[$sessionId$] := [startRequest:[message:$m$, key:$k$, receiver:$a$, sender:$f$], identity : $id$]   → Create new session record.
        call START_LOGIN_FLOW($sessionId$, $s'$)   → Call the function that starts or proceeds with a login flow. Runs discovery/registration/etc.
    else if $m$.path $\equiv$ /redirect_ep then   → User is being redirected after authentication to the IdP.
        let $sessionId := m$.headers[Cookie][sessionId]
        if $sessionId \notin s'$.sessions then
            stop
        let $session := s'$.sessions[$sessionId$]   → Retrieve session data.
        let $id := session$[id]
        let $issuer := s'$.issuerCache[$identity$]   → Issuer cache contains mappings from identites to issuers. Caches are being filled during discovery/registration in the function START_LOGIN_FLOW.
        if $m$.parameters[iss] $\not\equiv issuer$ then
            stop   → Check issuer parameter (cf. Section III-A).
        let $oidcConfig := s'$.oidcConfigCache[$issuer$]   → Retrieve OIDC configuration for issuer.
        let $responseType := session$[response_type]
        → Determines the OIDC flow to use, e.g., code id_token token for a hybrid flow.
        if $responseType \equiv \langle$code$\rangle$ then   → Authorization code mode: Take data from URL parameters.
            let $data := m$.parameters
        else   → Hybrid or implicit mode: Send the script script_rp_get_fragment to the browser to retrieve data from URL fragment
            if $m$.method $\equiv$ GET then
                let $headers := \langle\langle$ReferrerPolicy, origin$\rangle\rangle$
                let $m' := \mathsf{enc_s}(\langle$HTTPResp, $m$.nonce, 200, $headers$, $\langle$script_rp_get_fragment, $\bot\rangle\rangle, k)$
                stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
            else   → If this is a POST request, the script script_rp_get_fragment is sending the data from URL fragment.
                let $data := m$.body
        if $data$[state] $\not\equiv session$[state] then
            stop   → Check $state$ value.
        let $s'$.sessions[$sessionId$][redirectEpRequest] :=
        ↪ [message:$m$, key:$k$, receiver:$a$, sender:$f$]   → Store incoming request for later use in CHECK_ID_TOKEN (Algorithm 20).
        if id_token $\in^{\langle\rangle} responseType$ then   → Check if the chosen response type contains id_token.
            if code $\in^{\langle\rangle} responseType$ then   → In hybrid mode, only one of the two id tokens must be checked(cf. Appendix A).
                let $checkIdTokenNow \leftarrow \{\top, \bot\}$   → Non-deterministically decide whether to check first or second id token to capture all potential choices in real-world implementations (cf. Appendix A).
            else
                let $checkIdTokenNow := \top$   → In non-hybrid modes, the id token is always checked.
            if $checkIdTokenNow \equiv \top$ then[16]
                call CHECK_ID_TOKEN($sessionId$, $data$[id_token], $s'$)   → Check the id token and (if successful) log the user in. See Algorithm 20.
        let $useAccessTokenNow \leftarrow \{\top, \bot\}$   → Non-det. decide whether to use access token (authorization) or not.
        if token $\in^{\langle\rangle} responseType \wedge useAccessTokenNow \equiv \top$ then
            call USE_ACCESS_TOKEN($sessionId$, $m$.body[access_token], $s'$)   → Use the access token at the IdP.
        if code $\in^{\langle\rangle} responseType$ then
            call SEND_TOKEN_REQUEST($sessionId$, $m$.body[code], $s'$)   → Retrieve a token from the token endpoint of the IdP.
    stop
```

Algorithm 18 Relying Party R^r: Request to token endpoint.

```
function SEND_TOKEN_REQUEST($sessionId$, $code$, $s'$)
    let $session := s'.\mathtt{sessions}[sessionId]$
    let $identity := session[\mathtt{identity}]$
    let $issuer := s'.\mathtt{issuerCache}[identity]$
    let $credentials := s'.\mathtt{clientCredentialsCache}[issuer]$
    let $headers := []$
    let $body := [\mathtt{grant\_type}{:}\mathtt{authorization\_code}, \mathtt{code}{:}code, \mathtt{redirect\_uri}{:}session[\mathtt{redirect\_uri}]]$
    let $clientId := credentials[\mathtt{client\_id}]$
    let $clientSecret := credentials[\mathtt{client\_secret}]$
    if $clientSecret \equiv \langle\rangle$ then
        let $body[\mathtt{client\_id}] := clientId$
    else
        let $headers[\mathtt{Authorization}] := \langle clientId, clientSecret\rangle$
    let $url := s'.\mathtt{oidcConfigCache}[issuer][\mathtt{token\_ep}]$
    let $message := \langle \mathtt{HTTPReq}, \nu_2, \mathtt{POST}, url.\mathtt{domain}, url.\mathtt{path}, url.\mathtt{parameters}, headers, body\rangle$
    call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}{:}\mathtt{TOKEN}, \mathtt{session}{:}sessionId]$, $message$, $s'$)
```

Algorithm 19 Relying Party R^r: Using the access token (no response expected).

```
function USE_ACCESS_TOKEN($sessionId$, $token$, $s'$)
    let $session := s'.\mathtt{sessions}[sessionId]$
    let $identity := session[\mathtt{identity}]$
    let $issuer := s'.\mathtt{issuerCache}[identity]$
    let $headers := [\mathtt{Authorization}: \langle \mathtt{Bearer}, token\rangle]$
    let $url := s'.\mathtt{oidcConfigCache}[issuer][\mathtt{token\_ep}]$
    let $url.\mathtt{path} \leftarrow \mathbb{S}$
    let $message := \langle \mathtt{HTTPReq}, \nu_3, \mathtt{POST}, url.\mathtt{domain}, url.\mathtt{path}, url.\mathtt{parameters}, headers, \langle\rangle\rangle$
    call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}{:}\mathtt{RESOURCE\_USAGE}, \mathtt{session}{:}sessionId]$, $message$, $s'$)
```

Algorithm 20 Relying Party R^r: Check id token.

```
function CHECK_ID_TOKEN($sessionId$, $id\_token$, $s'$)    → Check id token validity and create service session.
    let $session := s'.\mathtt{sessions}[sessionId]$    → Retrieve session data.
    let $identity := session[\mathtt{identity}]$
    let $issuer := s'.\mathtt{issuerCache}[identity]$    → Retrieve issuer.
    let $oidcConfig := s'.\mathtt{oidcConfigCache}[issuer]$    → Retrieve OIDC configuration for that issuer.
    let $credentials := s'.\mathtt{clientCredentialsCache}[issuer]$    → Retrieve OIDC credentials for issuer.
    let $jwks := s'.\mathtt{jwksCache}[issuer]$    → Retrieve signing keys for issuer.
    let $data := \mathsf{extractmsg}(id\_token)$    → Extract contents of signed id token.
    if $data[\mathtt{iss}] \not\equiv issuer$ then
        stop    → Check the issuer.
    if $data[\mathtt{aud}] \not\equiv credentials[\mathtt{client\_id}]$ then
        stop    → Check the audience against own client id.
    if $\mathsf{checksig}(id\_token, jwks) \not\equiv \top$ then
        stop    → Check the signature of the id token.
    if $nonce \in session \wedge data[\mathtt{nonce}] \not\equiv session[\mathtt{nonce}]$ then
        stop    → If a nonce was used, check its value.
    let $s'.\mathtt{sessions}[sessionId][\mathtt{loggedInAs}] := \langle issuer, data[\mathtt{sub}]\rangle$    → User is now logged in. Store user identity and issuer.
    let $s'.\mathtt{sessions}[sessionId][\mathtt{serviceSessionId}] := \nu_4$    → Choose a new service session id.
    let $request := session[\mathtt{redirectEpRequest}]$    → Retrieve stored meta data of the request from the browser to the redir. endpoint in order to respond to it now. The request's meta data was stored in PROCESS_HTTPS_REQUEST (Algorithm 17).
    let $headers := [\mathtt{ReferrerPolicy}{:}\mathtt{origin}]$
    let $headers[\mathtt{Set\text{-}Cookie}] := [\mathtt{serviceSessionId}{:}\langle \nu_4, \top, \top, \top\rangle]$    → Create a cookie containing the service session id.
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, request[\mathtt{message}].\mathtt{nonce}, 200, headers, \mathtt{ok}\rangle, request[\mathtt{key}])$    → Respond to browser's request to the redirection endpoint.
    stop $\langle\langle request[\mathtt{sender}], request[\mathtt{receiver}], m'\rangle\rangle$, $s'$
```

Algorithm 21 Relying Party R^r: Continuing in the login flow.

```
function START_LOGIN_FLOW($sessionId$, $s'$)
    let $redirectUris := \{\langle \mathtt{URL}, \mathtt{S}, d, \mathtt{/redirect\_ep}, \langle\rangle, \langle\rangle \rangle | d \in \mathsf{dom}(r)\}$    → Set of redirect URIs for all domains.
    let $session := s'.\mathtt{sessions}[sessionId]$
    let $identity := session[\mathtt{identity}]$
    if $identity \not\in s'.\mathtt{issuerCache}$ then
        let $host := identity.\mathtt{domain}$
        let $path := \mathtt{/.wk/webfinger}$
        let $parameters := [\mathtt{resource}: identity]$
        let $message := \langle \mathtt{HTTPReq}, \nu_5, \mathtt{GET}, host, path, parameters, \langle\rangle, \langle\rangle \rangle$
        call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}{:}\mathtt{WEBFINGER}, \mathtt{session}{:}sessionId]$, $message$, $s'$)
    let $issuer := s'.\mathtt{issuerCache}[identity]$
    if $issuer \not\in s'.\mathtt{oidcConfigCache}$ then
        let $host := issuer$
        let $path := \mathtt{/.wk/openid\text{-}configuration}$
        let $message := \langle \mathtt{HTTPReq}, \nu_5, \mathtt{GET}, host, path, [], \langle\rangle, \langle\rangle \rangle$
        call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}{:}\mathtt{CONFIG}, \mathtt{session}{:}sessionId]$, $message$, $s'$)
    let $oidcConfig := s'.\mathtt{oidcConfigCache}[issuer]$
    if $issuer \not\in s'.\mathtt{jwksCache}$ then
        let $url := oidcConfig[\mathtt{jwks\_uri}]$
        let $message := \langle \mathtt{HTTPReq}, \nu_5, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, [], \langle\rangle, \langle\rangle \rangle$
        call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}{:}\mathtt{JWKS}, \mathtt{session}{:}sessionId]$, $message$, $s'$)
    if $issuer \not\in s'.\mathtt{clientCredentialsCache}$ then
        let $url := oidcConfig[\mathtt{reg\_ep}]$
        let $message := \langle \mathtt{HTTPReq}, \nu_5, \mathtt{POST}, url.\mathtt{host}, url.\mathtt{path}, [], \langle\rangle, [\mathtt{redirect\_uris}: \langle redirectUris \rangle] \rangle$
        call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}{:}\mathtt{REGISTRATION}, \mathtt{session}: sessionId]$, $message$, $s'$)
    let $credentials := s'.\mathtt{clientCredentialsCache}[issuer]$
    let $responseType \leftarrow \{\langle \mathtt{code} \rangle, \langle \mathtt{id\_token} \rangle, \langle \mathtt{id\_token}, \mathtt{token} \rangle, \langle \mathtt{code}, \mathtt{id\_token} \rangle,$
              ↪ $\langle \mathtt{code}, \mathtt{token} \rangle, \langle \mathtt{code}, \mathtt{id\_token}, \mathtt{token} \rangle\}$
    let $redirectUri \leftarrow redirectUris$
    let $data := [\mathtt{response\_type}{:}responseType, \mathtt{redirect\_uri}{:}redirectUri,$
              ↪ $\mathtt{client\_id}{:}credentials[\mathtt{client\_id}], \mathtt{state}{:}\nu_6]$
    if $\mathtt{code} \not\in^{\langle\rangle} responseType$ then    → Implicit flow requires nonce.
        let $data[\mathtt{nonce}] := \nu_7$
    let $s'.\mathtt{sessions}[sessionId] := s'.\mathtt{sessions}[sessionId] \cup data$
    let $authEndpoint := oidcConfig[\mathtt{auth\_ep}]$
    let $authEndpoint.\mathtt{parameters} := authEndpoint.\mathtt{parameters} \cup data$
    let $headers := [\mathtt{Location}{:}authEndpoint, \mathtt{ReferrerPolicy}{:}\mathtt{origin}]$
    let $headers[\mathtt{Set\text{-}Cookie}] := [\mathtt{sessionId}{:}\langle sessionId, \top, \top, \top \rangle]$
    let $request := s'.\mathtt{sessions}[sessionId][\mathtt{startRequest}]$
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, request[\mathtt{message}].\mathtt{nonce}, 303, headers, \bot \rangle, request[\mathtt{key}])$
    stop $\langle \langle request[\mathtt{sender}], request[\mathtt{receiver}], m' \rangle \rangle$, $s'$
```

Algorithm 22 Relation of *script_rp_index*

```
Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$ → Script that models the index page of a relying party. Users can initiate the login flow or follow arbitrary links. The script receives various information about the current browser state, filtered according to the access rules (same origin policy and others) in the browser.
let $switch \leftarrow \{\mathtt{auth}, \mathtt{link}\}$   → Non-deterministically decide whether to start a login flow or to follow some link.
if $switch \equiv \mathtt{auth}$ then   → Start login flow.
    let $url := \mathsf{GETURL}(tree, docnonce)$   → Retrieve own URL.
    let $id \leftarrow ids$   → Retrieve one of user's identities.
    let $url' := \langle \mathtt{URL}, \mathtt{S}, url.\mathtt{host}, \mathtt{/startLogin}, \langle\rangle, \langle\rangle\rangle$   → Assemble URL.
    let $command := \langle \mathtt{FORM}, url', \mathtt{POST}, id, \bot\rangle$
        → Post a form including the identity to the RP.
    stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$   → Finish script's run and instruct the browser to follow the command (form post).
else   → Follow link.
    let $protocol \leftarrow \{\mathtt{P}, \mathtt{S}\}$   → Non-deterministically select protocol (HTTP or HTTPS).
    let $host \leftarrow \mathsf{Doms}$   → Non-det. select host.
    let $path \leftarrow \mathbb{S}$   → Non-det. select path.
    let $fragment \leftarrow \mathbb{S}$   → Non-det. select fragment part.
    let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$   → Non-det. select parameters.
    let $url := \langle \mathtt{URL}, protocol, host, path, parameters, fragment\rangle$   → Assemble URL.
    let $command := \langle \mathtt{HREF}, url, \bot, \bot\rangle$   → Follow link to the selected URL.
    stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$   → Finish script's run and instruct the browser to follow the command (follow link).
```

Algorithm 23 Relation of *script_rp_get_fragment*

```
Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$
let $url := \mathsf{GETURL}(tree, docnonce)$
let $url' := \langle \mathtt{URL}, \mathtt{S}, url.\mathtt{host}, \mathtt{/redirect\_ep}, [\mathtt{iss}: url.\mathtt{parameters}[\mathtt{iss}]], \langle\rangle\rangle$
let $command := \langle \mathtt{FORM}, url', \mathtt{POST}, url.\mathtt{fragment}, \bot\rangle$
stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$
```

I. Identity Providers

An identity provider $i \in \mathsf{IDP}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s^i_0) with the addresses $I^i := \mathsf{addr}(i)$. Next, we define the set Z^i of states of i and the initial state s^i_0 of i.

Definition 50. A *state $s \in Z^i$ of an IdP i* is a term of the form $\langle$*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*, *registrationRequests*, (sequence of terms) *clients*, (dict from nonces to terms) *records*, (sequence of terms) *jwk*$\rangle$ (signing key (only one)) with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $tlskeys \in [\mathsf{Doms} \times K_{\text{TLS}}]$ (all former components as in Definition 47), $registrationRequests \in \mathcal{T}_{\mathcal{N}}$, $clients \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $records \in \mathcal{T}_{\mathcal{N}}$, and $jwk \in K_{\text{sign}}$.

An *initial state s^i_0 of i* is a state of i with $s^i_0.\mathtt{pendingDNS} \equiv \langle\rangle$, $s^i_0.\mathtt{pendingRequests} \equiv \langle\rangle$, $s^i_0.\mathtt{corrupt} \equiv \bot$, $s^i_0.\mathtt{keyMapping}$ being the same as the keymapping for browsers above, $s^i_0.\mathtt{tlskeys} \equiv tlskeys^i$, $s^i_0.\mathtt{registrationRequests} \equiv \langle\rangle$, $s^i_0.\mathtt{clients} \equiv an$, $s^i_0.\mathtt{records} \equiv \langle\rangle$, and $s^i_0.\mathtt{jwk} \equiv \mathsf{signkey}(i)$.

We now specify the relation R^i: As for the RPs above, this relation is based on our model of generic HTTPS servers (see Appendix E). We specify algorithms that differ from or do not exist in the generic server model in Algorithms 24 and 25. As above, Figure 12 shows a list of all placeholders used. Algorithm 26 shows the script *script_idp_form* that is used by IdPs.

Placeholder	Usage
ν_1	new authorization code
ν_2, ν_3	new access tokens
ν_4	new client secret

Figure 12. List of placeholders used in the identity provider algorithm.

Algorithm 24 Relation of IdP R^i – Processing HTTPS Requests

```
function PROCESS_HTTPS_REQUEST($m$, $k$, $a$, $f$, $s'$)
    if $m.\mathtt{path} \equiv /.\mathtt{wk}/\mathtt{webfinger}$ then
        let $user$, $domain$ such that $\langle user, domain\rangle \equiv m.\mathtt{parameters}[\mathtt{resource}] \wedge \langle user, domain\rangle \in \mathsf{ID}^i$ if possible; otherwise stop
        let $descriptor := [\mathtt{subject}: \langle user, domain\rangle, \mathtt{links}: [\mathtt{rel}: \mathtt{OIDC\_issuer}, \mathtt{href}: m.\mathtt{host}]]$
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\rangle, descriptor\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
    else if $m.\mathtt{path} \equiv /.\mathtt{wk}/\mathtt{openid\text{-}configuration}$ then
        let $metaData := [\mathtt{issuer}: m.\mathtt{host}]$[17]
        let $metaData[\mathtt{auth\_ep}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, /\mathtt{auth}, \langle\rangle, \langle\rangle\rangle$
        let $metaData[\mathtt{token\_ep}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, /\mathtt{token}, \langle\rangle, \langle\rangle\rangle$
        let $metaData[\mathtt{jwks\_uri}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, /\mathtt{jwks}, \langle\rangle, \langle\rangle\rangle$
        let $metaData[\mathtt{reg\_ep}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, /\mathtt{reg}, \langle\rangle, \langle\rangle\rangle$
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\rangle, metaData\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
    else if $m.\mathtt{path} \equiv /\mathtt{jwks}$ then
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 201, \langle\rangle, \mathsf{pub}(s'.\mathtt{jwk})\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
    else if $m.\mathtt{path} \equiv /\mathtt{reg} \wedge m.\mathtt{method} \equiv \mathtt{POST}$ then
        let $s'.\mathtt{registrationRequests} := s'.\mathtt{registrationRequests} +^{\langle\rangle} \langle m, k, a, f\rangle$
        stop    → Stop here to let attacker choose the client id.
    else if $m.\mathtt{path} \equiv /\mathtt{auth}$ then
        if $m.\mathtt{method} \equiv \mathtt{GET}$ then
            let $data := m.\mathtt{parameters}$
        else if $m.\mathtt{method} \equiv \mathtt{POST}$ then
            let $data := m.\mathtt{body}$
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\langle \mathtt{ReferrerPolicy}, \mathtt{origin}\rangle\rangle, \langle \mathtt{script\_idp\_form}, data\rangle\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
    else if $m.\mathtt{path} \equiv /\mathtt{auth2} \wedge m.\mathtt{method} \equiv \mathtt{POST} \wedge m.\mathtt{headers}[\mathtt{Origin}] \equiv \langle m.\mathtt{host}, \mathtt{S}\rangle$ then
        let $identity := m.\mathtt{body}[\mathtt{identity}]$
        let $password := m.\mathtt{body}[\mathtt{password}]$
        if $identity.\mathtt{domain} \notin \mathsf{dom}(i)$ then
            stop
        if $password \not\equiv \mathsf{secretOfID}(identity)$ then
            stop
        let $responseType := m.\mathtt{body}[\mathtt{response\_type}]$
        let $clientId := m.\mathtt{body}[\mathtt{client\_id}]$
        let $redirectUri := m.\mathtt{body}[\mathtt{redirect\_uri}]$
        let $state := m.\mathtt{body}[\mathtt{state}]$
        let $nonce := m.\mathtt{body}[\mathtt{nonce}]$
        if $clientId \notin s'.\mathtt{clients}$ then
            stop
        let $clientInfo := s'.\mathtt{clients}[clientId]$
        if $redirectUri \not\in^{\langle\rangle} clientInfo[\mathtt{redirect\_uris}]$ then
            stop
        let $record := [\mathtt{client\_id}: clientId]$
        let $record[\mathtt{redirect\_uri}] := redirectUri$
        let $record[\mathtt{subject}] := identity$
        let $record[\mathtt{issuer}] := m.\mathtt{host}$
        let $record[\mathtt{nonce}] := nonce$
        let $record[\mathtt{code}] := \nu_1$
        let $record[\mathtt{access\_tokens}] := \langle \nu_2, \nu_3\rangle$[18]
        let $s'.\mathtt{records} := s'.\mathtt{records} +^{\langle\rangle} record$
        let $\mathit{responseData} := [\,]$
        if $\mathtt{code} \in^{\langle\rangle} \mathit{responseType}$ then
            let $\mathit{responseData}[\mathtt{code}] := v_1$
        if $\mathtt{token} \in^{\langle\rangle} \mathit{responseType}$ then
            let $\mathit{responseData}[\mathtt{access\_token}] := v_2$
            let $\mathit{responseData}[\mathtt{token\_type}] := \mathtt{bearer}$
        if $\mathtt{id\_token} \in^{\langle\rangle} \mathit{responseType}$ then
            let $\mathit{idTokenBody} := [\mathtt{iss}: \mathit{record}[\mathtt{issuer}], \mathtt{sub}: \mathit{record}[\mathtt{subject}],$
                $\hookrightarrow \mathtt{aud}: \mathit{record}[\mathtt{client\_id}], \mathtt{nonce}: \mathit{record}[\mathtt{nonce}]]$
            let $\mathit{responseData}[\mathtt{id\_token}] := \mathsf{sig}(\mathit{idTokenBody}, s'.\mathtt{jwk})$
        if $\mathit{state} \not\equiv \langle\rangle$ then
            let $\mathit{responseData}[\mathtt{state}] := \mathit{state}$
        if $\mathit{responseType} \equiv \langle\mathtt{code}\rangle$ then    → Authorization Code Mode
            let $\mathit{redirectUri}.\mathtt{parameters} := \mathit{redirectUri}.\mathtt{parameters} \cup \mathit{responseData}$
        else    → Implicit/Hybrid Mode
            if $\mathtt{code} \not\in^{\langle\rangle} \mathit{responseType} \wedge \mathtt{id\_token} \in^{\langle\rangle} \mathit{responseType} \wedge \mathit{nonce} \equiv \langle\rangle$ then
                stop    → Nonce is required in implicit mode.
            let $\mathit{redirectUri}.\mathtt{fragment} := \mathit{redirectUri}.\mathtt{fragment} \cup \mathit{responseData}$
        let $\mathit{redirectUri}.\mathtt{parameters}[\mathtt{iss}] := \mathit{record}[\mathtt{issuer}]$
        let $m' := \mathsf{enc_s}(\langle\mathtt{HTTPResp}, m.\mathtt{nonce}, 303, \langle\langle\mathtt{Location}, \mathit{redirectUri}\rangle\rangle, \langle\rangle\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
    else if $m.\mathtt{path} \equiv /\mathtt{token} \wedge m.\mathtt{method} \equiv \mathtt{POST}$ then
        if $\mathtt{client\_id} \in m.\mathtt{body}$ then    → Only client id is provided, no client secret.
            let $\mathit{clientId} := m.\mathtt{body}[\mathtt{client\_id}]$
            let $\mathit{clientSecret} := \langle\rangle$
        else
            let $\mathit{clientId} := m.\mathtt{headers}[\mathtt{Authorization}].\mathtt{username}$
            let $\mathit{clientSecret} := m.\mathtt{headers}[\mathtt{Authorization}].\mathtt{password}$
        let $\mathit{clientInfo} := s'.\mathtt{clients}[\mathit{clientId}]$
        if $\mathit{clientInfo} \equiv \langle\rangle \vee \mathit{clientInfo}[\mathtt{client\_secret}] \not\equiv \mathit{clientSecret}$ then
            stop
        let $\mathit{code} := m.\mathtt{body}[\mathtt{code}]$
        let $\mathit{record}$, $\mathit{ptr}$ such that $\mathit{record} \equiv s'.\mathtt{records}.\mathit{ptr} \wedge \mathit{record}[\mathtt{code}] \equiv \mathit{code} \wedge \mathit{code} \not\equiv \bot$ if possible; otherwise stop
        if $\mathit{record}[\mathtt{client\_id}] \not\equiv \mathit{clientId}$ then
            stop
        if not $(\mathit{record}[\mathtt{redirect\_uri}] \equiv m.\mathtt{body}[\mathtt{redirect\_uri}] \vee (|\mathit{clientInfo}[\mathtt{redirect\_uris}]| = 1 \wedge \mathtt{redirect\_uri} \not\in m.\mathtt{body}))$ then
            stop    → If only one redirect URI is registered, it can be omitted.
        let $s'.\mathtt{records}.\mathit{ptr}[\mathtt{code}] := \bot$    → Invalidate code
        let $\mathit{accessTokenChoice} \leftarrow \{1, 2\}$
        let $\mathit{accessToken} := \mathit{record}[\mathtt{access\_tokens}].\mathit{accessTokenChoice}$
        let $\mathit{idTokenBody} := [\mathtt{iss}: \mathit{record}[\mathtt{issuer}]]$
        let $\mathit{idTokenBody}[\mathit{sub}] := \mathit{record}[\mathtt{subject}]$
        let $\mathit{idTokenBody}[\mathit{aud}] := \mathit{record}[\mathtt{client\_id}]$
        let $\mathit{idTokenBody}[\mathit{nonce}] := \mathit{record}[\mathtt{nonce}]$
        let $\mathit{id\_token} := \mathsf{sig}(\mathit{idTokenBody}, s'.\mathtt{jwk})$
        let $m' := \mathsf{enc_s}(\langle\mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\rangle, [\mathtt{access\_token}{:}\mathit{accessToken}, \mathtt{token\_type}{:}\mathtt{bearer}, \mathtt{id\_token}{:}\mathit{id\_token}]\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
```

Algorithm 25 Relation of IdP R^i – Processing other messages.

```
function PROCESS_OTHER($m$, $a$, $f$, $s'$)
    let $clientId := m$    $\rightarrow$ $m$ is client id chosen by and sent by an attacker process.
    if $clientId \in s'.\texttt{clients}$ then
        stop
    let $m$, $k$, $a$, $f$ such that $\langle m,k,a,f\rangle \in^{\langle\rangle} s'.\texttt{registrationRequests}$ if possible; otherwise stop
    remove $\langle m,k,a,f\rangle$ from $s'.\texttt{registrationRequests}$
    let $redirectUris := m.\texttt{body}[\texttt{redirect\_uris}]$
    let $regResponse := [\texttt{client\_id}: clientId]$
    let $issueSecret \leftarrow \{\top,\bot\}$
    if $issueSecret \equiv \top$ then
        let $clientSecret := \nu_4$
        let $regResponse[\texttt{client\_secret}] := clientSecret$
    let $clientInfo := regResponse$
    let $clientInfo[\texttt{redirect\_uris}] := redirectUris$
    let $s'.\texttt{clients}[clientId] := clientInfo$
    let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 201, \langle\rangle, regResponse\rangle, k)$
    stop $\langle\langle f,a,m'\rangle\rangle$, $s'$
```

Algorithm 26 Relation of *script_idp_form*

```
Input: $\langle tree$, $docnonce$, $scriptstate$, $scriptinputs$, $cookies$, $localStorage$, $sessionStorage$, $ids$, $secrets\rangle$
let $url := \mathsf{GETURL}(tree, docnonce)$
let $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/auth2}, \langle\rangle, \langle\rangle\rangle$
let $formData := scriptstate$
let $identity \leftarrow ids$
let $secret \leftarrow secrets$
let $formData[\texttt{identity}] := identity$
let $formData[\texttt{password}] := secret$
let $command := \langle \texttt{FORM}, url', \texttt{POST}, formData, \bot\rangle$
stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$
```

Appendix G
Formal Model of OpenID Connect with Web Attackers

We now derive $\mathcal{OIDC}^w$ (an *OIDC web system with web attackers*) from $\mathcal{OIDC}^n$ by replacing the network attacker with a finite set of web attackers. (Note that we more generally speak of an *OIDC web system* if it is not important what kind of attacker the web system contains.)

Definition 51. An *OIDC web system with web attackers*, $\mathcal{OIDC}^w$, is an OIDC web system $\mathcal{OIDC}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ with the following changes:

- We have $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Web}$, in particular, there is no network attacker. The set Web contains a finite number of web attacker processes. The set Hon is as described above, and additionally contains a DNS server d as defined below.
- The set of IP addresses IPs contains no IP addresses for the network attacker, but instead a finite set of IP addresses for each web attacker.
- The set of Domains Doms contains no domains for the network attacker, but instead a finite set of domains for each web attacker.
- All honest parties use the DNS server d as their DNS server.

A. DNS Server

The DNS server d is a DNS server as defined in Definition 35. Its initial state s_0^d contains only pairings $\langle D, i\rangle$ such that $i \in \mathsf{addr}(\mathsf{dom}^{-1}(D))$, i.e., any domain is resolved to an IP address belonging to the owner of that domain (as defined in Appendix F-B).

B. Web Attackers

Web attackers, as opposed to network attackers, can only use their own IP addresses for listening to and sending messages. Therefore, for any web attacker process w we have that $I^w = \mathsf{addr}(w)$. The inital states of web attackers are defined parallel to those of network attackers, i.e., the initial state for a web attacker process w is $s_0^w = \langle \mathit{attdoms}^w, \mathit{tlskeys}, \mathit{signkeys}\rangle$, where $\mathit{attdoms}^w$ is a sequence of all domains along with the corresponding private keys owned by the attacker w, $\mathit{tlskeys}$ is a sequence of all domains and the corresponding public keys, and $\mathit{signkeys}$ is a sequence containing all public signing keys for all IdPs.

Appendix H
Formal Security Properties

The security properties for OIDC are formally defined as follows.

A. Authentication

Intuitively, authentication for $\mathcal{OIDC}^n$ means that an attacker should not be able to login at an (honest) RP under the identity of a user unless certain parties involved in the login process are corrupted. As explained above, being logged in at an RP under some user identity means to have obtained a service token for this identity from the RP.

Definition 52 (Service Sessions). We say that there is a *service session identified by a nonce n for an identity id at some RP r* in a configuration (S,E,N) of a run ρ of an OIDC web system iff there exists some session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ such that $S(r).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d, id\rangle$ and $S(r).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$.

Definition 53 (Authentication Property). Let $\mathcal{OIDC}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDC}^n$ is secure w.r.t. authentication* iff for every run ρ of $\mathcal{OIDC}^n$, every configuration (S,E,N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id)$ being an honest IdP, every service session identified by some nonce n for id at r, n is not derivable from the attackers knowledge in S (i.e., $n \not\in d_{\emptyset}(S(\mathsf{attacker}))$).

B. Authorization

Intuitively, authorization for $\mathcal{OIDC}^n$ means that an attacker should not be able to obtain or use a protected resource available to some honest RP at an IdP for some user unless certain parties involved in the authorization process are corrupted.

Definition 54. We say that a client id *c has been issued to r by i* iff i has sent a response to a registration request from r in Line 17 of Algorithm 25 and this response contains c in its body under the dictionary key `client_id`.

Definition 55 (Authorization Property). Let $\mathcal{OIDC}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDC}^n$ is secure w.r.t. authorization* iff for every run ρ of $\mathcal{OIDC}^n$, every configuration (S,E,N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $i \in \mathsf{IdP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}^i$ owned by b, every nonce n, every term $x \in^{\langle\rangle} S(i).\mathtt{records}$ with $x[\mathtt{subject}] \equiv id$, $n \in^{\langle\rangle} x[\mathtt{access\_tokens}]$, and the client id $x[\mathtt{client\_id}]$ has been issued by i to r, we have that n is not derivable from the attackers knowledge in S (i.e., $n \not\in d_{\emptyset}(S(\mathsf{attacker}))$).

C. Session Integrity for Authentication and Authorization

The two session integrity properties capture that an attacker should be unable to forcefully log a user in to some RP. This includes attacks such as CSRF and session swapping.

Session Integrity Property for Authentication. This security property captures that (a) a user should only be logged in when the user actually expressed the wish to start an OIDC flow before, and (b) if a user expressed the wish to start an OIDC flow using some honest identity provider and a specific identity, then user is not logged in under a different identity.

We first need to define notations for the processing steps that represent important events during a flow of an OIDC web system.

Definition 56 (User is logged in). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDC}^w$ we say that a browser b was authenticated to an RP r using an IdP i and an identity u in a login session identified by a nonce $lsid$ in processing step Q in ρ with

$$Q = (S,E,N) \xrightarrow[r \rightarrow E_{\text{out}}]{} (S',E',N')$$

(for some S, S', E, E', N, N') and some event $\langle y, y', m\rangle \in E_{\text{out}}$ such that m is an HTTPS response matching an HTTPS request sent by b to r and we have that in the headers of m there is a header of the form $\langle \mathtt{Set\text{-}Cookie}, [\mathtt{serviceSessionId}{:}\langle ssid, \top, \top, \top\rangle]\rangle$ for some nonce $ssid$ and we have that there is a term g such that $S(r).\mathtt{sessions}[lsid] \equiv g$, $g[\mathtt{serviceSessionId}] \equiv ssid$, and $g[\mathtt{loggedInAs}] \equiv \langle d, u\rangle$ with $d \in \mathsf{dom}(i)$. We then write $\mathsf{loggedIn}^Q_\rho(b, r, u, i, lsid)$.

Definition 57 (User started a login flow). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDC}^w$ we say that the user of the browser b started a login session identified by a nonce $lsid$ at the RP r in a processing step Q in ρ if (1) in that processing step, the browser b was triggered, selected a document loaded from an origin of r, executed the script *script_rp_index* in that document, and in that script, executed the Line 7 of Algorithm 22, and (2) r sends an HTTPS response corresponding to the HTTPS request sent by b in Q and in that response, there is a header of the form $\langle \mathtt{Set\text{-}Cookie}, [\mathtt{sessionId}{:}\langle lsid, \top, \top, \top\rangle]\rangle$. We then write $\mathsf{started}^Q_\rho(b, r, lsid)$.

Definition 58 (User authenticated at an IdP). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDC}^w$ we say that the user of the browser b authenticated to an IdP i using an identity u for a login session identified by a nonce $lsid$ at the RP r if there is a processing step Q in ρ with

$$Q = (S,E,N) \to (S',E',N')$$

(for some S, S', E, E', N, N') in which the browser b was triggered, selected a document loaded from an origin of i, executed the script *script_idp_form* in that document, and in that script, (1) in Line 4 of Algorithm 26, selected the identity u, and (2) we have that the *scriptstate* of that document, when triggered, contains a nonce s such that $\mathit{scriptstate}[\mathtt{state}] \equiv s$ and $S(r).\mathtt{sessions}[\mathit{lsid}][\mathtt{state}] \equiv s$. We then write $\mathsf{authenticated}_\rho^Q(b,r,u,i,\mathit{lsid})$.

Definition 59 (RP uses an access token). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDC}^w$ we say that the RP r uses some access token t in a login session identified by the nonce $lsid$ established with the browser b at an IdP i if there is a processing step Q in ρ with

$$Q = (S,E,N) \to (S',E',N')$$

(for some S, S', E, E', N, N') in which (1) r calls the function $\mathsf{USE\_ACCESS\_TOKEN}$ with the first two parameters being $lsid$ and t, (2) $S(r).\mathtt{issuerCache}[S(r).\mathtt{sessions}[\mathit{lsid}][\mathtt{identity}]] \in \mathsf{dom}(i)$, and (3) $\langle \mathtt{sessionid}, \langle \mathit{lsid}, y, z, z' \rangle\rangle \in^{\langle\rangle} S(b).\mathtt{cookies}[d]$ for $d \in \mathsf{dom}(r)$, $y,z,z' \in \mathcal{T}_{\mathcal{N}}$. We then write $\mathsf{usedAuthorization}_\rho^Q(b,r,i,\mathit{lsid})$.

Definition 60 (RP acts on the user's behalf). For a run ρ of an OIDC web system with web attacker $\mathcal{OIDC}^w$ we say that the RP r acts on behalf of the user with the identity u at an honest IdP i in a login session identified by the nonce $lsid$ established with the browser b if there is a processing step Q in ρ with

$$Q = (S,E,N) \to (S',E',N')$$

(for some S, S', E, E', N, N') in which (1) r calls the function $\mathsf{USE\_ACCESS\_TOKEN}$ with the first two parameters being $lsid$ and t, (2) we have that there is a term g such that $g \in^{\langle\rangle} S(i).\mathtt{records}$ with $t \in^{\langle\rangle} g[\mathtt{access\_tokens}]$ and $g[\mathtt{subject}] \equiv u$, and (3) $\langle \mathtt{sessionid}, \langle \mathit{lsid}, y, z, z' \rangle\rangle \in^{\langle\rangle} S(b).\mathtt{cookies}[d]$ for $d \in \mathsf{dom}(r)$, $y,z,z' \in \mathcal{T}_{\mathcal{N}}$. We then write $\mathsf{actsOnUsersBehalf}_\rho^Q(b,r,u,i,\mathit{lsid})$.

For session integrity for authentication we say that a user that is logged in at some RP must have expressed her wish to be logged in to that RP in the beginning of the login flow. If the IdP is honest, then the user must also have authenticated herself at the IdP with the same user account that RP uses for her identification. This excludes, for example, cases where (1) the user is forcefully logged in to an RP by an attacker that plays the role of an IdP, and (2) where an attacker can force a user to be logged in at some RP under a false identity issued by an honest IdP.

Definition 61 (Session Integrity for Authentication). Let $\mathcal{OIDC}^w$ be an OIDC web system with web attackers. We say that *$\mathcal{OIDC}^w$ is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDC}^w$, every processing step Q in ρ with

$$Q = (S,E,N) \to (S',E',N')$$

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{IdP}$, every identity u that is owned by b, every $r \in \mathsf{RP}$ that is honest in S, every nonce $lsid$, and $\mathsf{loggedIn}_\rho^Q(b,r,u,i,\mathit{lsid})$ we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b,r,\mathit{lsid})$, and (2) if i is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b,r,u,i,\mathit{lsid})$.

For session integrity for authorization we say that if an RP uses some access token at some IdP in a session with a user, then that user expressed her wish to authorize the RP to interact with some IdP. If the IdP is honest, and the RP acts on the user's behalf at the IdP (i.e., the access token is bound to the user's identity), then the user authenticated to the IdP using that identity.

Definition 62 (Session Integrity for Authorization). Let $\mathcal{OIDC}^w$ be an OIDC web system with web attackers. We say that *$\mathcal{OIDC}^w$ is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDC}^w$, every processing step Q in ρ with

$$Q = (S,E,N) \to (S',E',N')$$

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{IdP}$, every identity u that is owned by b, every $r \in \mathsf{RP}$ that is honest in S, every nonce $lsid$, we have that (1) if $\mathsf{usedAuthorization}_\rho^Q(b,r,i,\mathit{lsid})$ then there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b,r,\mathit{lsid})$, and (2) if i is honest in S and $\mathsf{actsOnUsersBehalf}_\rho^Q(b,r,u,i,\mathit{lsid})$ then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b,r,u,i,\mathit{lsid})$.

Appendix I
Proof of Theorem 1

Before we prove Theorem 1, in order to provide a quick overview, we first provide a proof sketch. We then show some general properties of OIDC web systems with a network attacker, and then proceed to prove the authentication, authorization, and session integrity properties separately.

A. Proof Sketch

For authentication and authorization, we first show that the secondary security properties from Section V-C hold true (see Lemmas 1–6 below). We then assume that the authentication/authorization properties do not hold, i.e., that there is a run ρ of $\mathcal{OIDC}^n$ that does not satisfy authentication or authorization, respectively. Using Lemmas 1–6, it then only requires a few steps to lead the respective assumption to a contradication and thereby show that $\mathcal{OIDC}^n$ enjoys authentication/authorization.

For the session integrity properties, we follow a similar scheme. We first show Lemma 9, which essentially says that a web attacker is unable to get hold of the *state* value that is used in a session between an honest browser b, an honest RP r, and an honest IdP i. (Recall that the *state* value is essential for session integrity.) We then show session integrity for authentication/authorization by starting from the latest "known" processing steps in the respective flows (e.g., for authentication, $\mathsf{loggedIn}_\rho^Q(b,r,u,i,lsid)$) and tracking through the OIDC flows to show the existence of the earlier processing steps (e.g., $\mathsf{started}_\rho^{Q'}(b,r,lsid)$) and their respective properties.

B. Properties of $\mathcal{OIDC}^n$

Let $\mathcal{OIDC}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be an OIDC web system with a network attacker. Let ρ be a run of $\mathcal{OIDC}^n$. We write $s_x = (S^x, E^x, N^x)$ for the states in ρ.

Definition 63. We say that a term t *is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \to s_{i+1}$ of a run $\rho = (s_0, s_1, \dots)$)* if t is derivable from t' with the knowledge available to P, i.e.,

$$t \in d_\emptyset(\{t'\} \cup \bigcup_{p \in P} S^{i+1}(p))$$

Definition 64. We say that *a set of processes P leaks a term t (in a processing step $s_i \to s_{i+1}$) to a set of processes P'* if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.

Definition 65. We say that an DY process p *created* a message m (at some point) in a run if m is derivably contained in a message emitted by p in some processing step and if there is no earlier processing step where m is derivably contained in a message emitted by some DY process p'.

Definition 66. We say that *a browser b accepted* a message (as a response to some request) if the browser decrypted the message (if it was an HTTPS message) and called the function PROCESSRESPONSE, passing the message and the request (see Algorithm 8).

Definition 67. We say that an atomic DY process p *knows a term t* in some state $s = (S, E, N)$ of a run if it can derive the term from its knowledge, i.e., $t \in d_\emptyset(S(p))$.

Definition 68. We say that a *script initiated a request r* if a browser triggered the script (in Line 10 of Algorithm 7) and the first component of the *command* output of the script relation is either `HREF`, `IFRAME`, `FORM`, or `XMLHTTPREQUEST` such that the browser issues the request r in the same step as a result.

The following lemma captures properties of RP when it uses HTTPS. For example, the lemma says that other parties cannot decrypt messages encrypted by RP.

C. Proof of Authentication

We here want to show that every OIDC web system is secure w.r.t. authentication, and therefore assume that there exists an OIDC web system that is not secure w.r.t. authentication. We then lead this to a contradiction, thereby showing that all OIDC web systems are secure w.r.t. authentication. In detail, we assume:

Lemma 1 (Integrity of Issuer Cache). For any run ρ of an OIDC web system $\mathcal{OIDC}^n$ with a network attacker or an OIDC web system $\mathcal{OIDC}^w$ with web attackers, every configuration (S, E, N) in ρ, every IdP i that is honest in S, every identity $id \in \mathsf{ID}^i$, every relying party r that is honest in S, we have that $S(r).\mathtt{issuerCache}[id] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{issuerCache}[id] \in \mathsf{dom}(i)$.

Proof. Initially, the issuer cache of an honest relying party is empty (according to Definition 49). This issuer cache can only be modified in Line 11 of Algorithm 16. There, the value of $S^l(r).\mathtt{issuerCache}[id']$ (for some $l < j$) is taken from an HTTPS response. The value of id' is taken from session data (Line 3) which is identified by a session id that is taken from the internal reference data of the incoming message. This internal reference data must have been created previously in Algorithm 10 (HTTPS_SIMPLE_SEND) which must have been called in Line 10 of Algorithm 21 (since this is the only place where the reference data for a webfinger request is created). In this algorithm, it is easy to see that the request to which the request is sent (see Line 6) is the domain part of the identity. We therefore have that a webfinger request must have been sent (using HTTPS) to the IdP i. (Note that an attacker can neither decrypt any information from this request, nor spoof a response to this request. The request must therefore have been responded to by the honest IdP.)

Since the path of this request is $/\mathtt{.wk/webfinger}$, the IdP can respond to this request only in Lines 3ff. of Algorithm 24. Since the IdP there chooses an issuer value that is one of its own domains (see Line 4), we finally have that $S(r).\mathtt{issuerCache}[id] \equiv \langle\rangle$ (if the response is blocked or the webfinger request was never sent) or we have that $S(r).\mathtt{issuerCache}[id] \in \mathsf{dom}(i)$, which proves the lemma. ■

Lemma 2 (Integrity of oidcConfigCache). For any run ρ of an OIDC web system $\mathcal{OIDC}^n$ with a network attacker or an OIDC web system $\mathcal{OIDC}^w$ with web attackers, every configuration (S,E,N) in ρ, every IdP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, $l \in \{1,2,3,4\}$ we have that $S(r).\mathtt{oidcConfigCache}[d] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{oidcConfigCache}[d] \equiv [\mathtt{issuer}: d, \mathtt{auth\_ep}: u_1, \mathtt{token\_ep}: u_2, \mathtt{jwks\_ep}: u_3, \mathtt{reg\_ep}: u_4]$ with u_l being URLs, $u_l.\mathtt{host} \in \mathsf{dom}(i)$, and $u_l.\mathtt{protocol} \equiv \mathtt{S}$.

PROOF. This proof proceeds analog to the one for Lemma 1 with the following changes: First, the OIDC configuration cache is filled only in Line 17 of Algorithm 16. It requires a request that was created in Line 16 of Algorithm 21. This request was not sent to the domain contained in an ID (as above) but instead to the issuer (in this case, d). The issuer responds to this request in Lines 8ff. of Algorithm 24. There, the issuer only choses the redirection endpoint URIs such that the host is the domain of the incoming request and the protocol is HTTPS ($\mathtt{S}$). This proves the lemma. ■

Lemma 3 (Integrity of JWKS Cache). For any run ρ of an OIDC web system $\mathcal{OIDC}^n$ with a network attacker or an OIDC web system $\mathcal{OIDC}^w$ with web attackers, every configuration (S,E,N) in ρ, every IdP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, we have that $S(r).\mathtt{jwksCache}[d] \equiv \langle\rangle$ (not set) or $S(r).\mathtt{jwksCache}[d] \equiv \mathsf{pub}(S(i).\mathtt{jwks})$.

PROOF. This proof proceeds analog to the one for Lemma 2. The relevant HTTPS request by r is created in Line 21 of Algorithm 21, and responded to by the IdP i in Lines 15ff. of Algorithm 24. There, the IdP chooses its own signature verification key to send in the response. This proves the lemma. ■

Lemma 4 (Integrity of Client Registration). For any run ρ of an OIDC web system $\mathcal{OIDC}^n$ with a network attacker or an OIDC web system $\mathcal{OIDC}^w$ with web attackers, every configuration (S,E,N) in ρ, every IdP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every relying party r that is honest in S, every client id c that has been issued to r by i, every URL $u \in^{\langle\rangle} S(i).\mathtt{clients}[c][\mathtt{redirect\_uris}]$ we have that $u.\mathtt{host} \in \mathsf{dom}(r)$ and $u.\mathtt{protocol} \equiv \mathtt{S}$.

PROOF. From Definition 54 it follows that an HTTPS request must have been sent from r to i in Lines 23ff. of Algorithm 21. This request must have been processed by i in Lines 18ff. of Algorithm 24, and, after receiving the client id from some other party (usually the attacker), in Algorithm 25. From the latter algorithm it is easy to see that the redirection endpoint data must have been taken from r's initial registration request to create the dictionary stored in $S(i).\mathtt{clients}[c]$. This data, however, was chosen by r in Line 2 of Algorithm 21 such that $u.\mathtt{host} \in \mathsf{dom}(r)$ and $u.\mathtt{protocol} \equiv \mathtt{S}$ for every $u \in^{\langle\rangle} S(i).\mathtt{clients}[c][\mathtt{redirect\_uris}]$.■

Lemma 5 (Other parties do not learn passwords). For any run ρ of an OIDC web system $\mathcal{OIDC}^n$ with a network attacker or an OIDC web system $\mathcal{OIDC}^w$ with web attackers, every configuration (S,E,N) in ρ, every IdP i that is honest in S, every identity $id \in \mathsf{ID}^i$, every browser b with $b = \mathsf{ownerOfID}(id)$ that is honest in S, every $p \in \mathcal{W} \setminus \{b,i\}$ we have that $\mathsf{secretOfID}(id) \notin d_{\emptyset}(S^l(p))$.

PROOF. Let $s := \mathsf{secretOfID}(id)$. Initially, in S^0, s is only contained in $S^0(b).\mathtt{secrets}[\langle d, \mathtt{S}\rangle]$ with $d \in \mathsf{dom}(i)$ and in no other states of any atomic processes (or in any waiting events). By the definition of the browser, we can see that only scripts loaded from the origins $\langle d, \mathtt{S}\rangle$ can access s. We know that i is an honest IdP. Now, the only script that an honest IdP sends to the browser is *script_idp_form*. This scripts sends the form data only to its own origin, which means, that the form data is sent over HTTPS and to the honest IdP. In this request, the script uses the path $/\mathtt{auth2}$. There, identity and password are checked, but not used otherwise. Therefore, the form data cannot leak from the honest IdP. It could, however, leak from the browser itself. The form data is sent via POST, and therefore, not used in any referer headers. A redirection response from the server contains the status code 303, which implies that the browser does not send the form data again when following the redirection. Since there are also no other scripts from the same origin running in the browser which could access the form data, the password s cannot leak from the browser either. This proves Lemma 5. ■

Lemma 6 (Attacker does not Learn ID Tokens). For any run ρ of an OIDC web system $\mathcal{OIDC}^n$ with a network attacker or an OIDC web system $\mathcal{OIDC}^w$ with web attackers, every configuration (S,E,N) in ρ, every IdP i that is honest in S, every domain $d \in \mathsf{dom}(i)$, every identity $id \in \mathsf{ID}^i$ with $b = \mathsf{ownerOfID}(id)$ being an honest browser (in S), every relying party r that is honest in S, every client id c that has been issued to r by i, every term y, every id token $t = \mathsf{sig}([\mathtt{iss}: d, \mathtt{sub}: id, \mathtt{aud}: c, \mathtt{nonce}: y], k)$ with $k = S(i).\mathtt{jwks}$, every attacker process a we have that $t \not\in d_\emptyset(S(a))$.

PROOF. The signing key k is only known to i initially and at least up to S (since i is honest). Therefore, only i can create t. There are two places where an honest IdP can create such a token in Algorithm 24: In Line 60 (immediately after receiving the user credentials) and in Lines 92ff. (after receiving an access token).

We now distinguish between these two cases to show that in either case, the attacker cannot get hold of an id token. We start with the first case.

ID token was created in Line 60.

To create t, the IdP i must have received a request to the path $/\mathtt{auth2}$ in Lines 28ff. of Algorithm 24. It is clear that i sends the response to this request to the sender of the request, and, if that sender is honest, the response cannot be read by an attacker. The request must contain $\mathsf{secretOfId}(id)$. Only b and i know this secret (per Lemma 5). Since i does not send requests to itself, the request must have been sent from b. Since the origin header in the request must be a domain of i, we know that the request was not initiated by a script other than i's own scripts, in particular, it must have been initiated by *script_idp_form*.

Now it is easy to see that this script does not use the token t in any way after the token was returned from i, since the script uses a form post to transmit the credentials to i, and the window is subsequently navigated away. Instead, i provides an empty script in its response to b. This response contains a location redirect header. It is now crucial to check that this location redirect does not cause the id token to be leaked to the attacker: With Lemma 4 we have that the redirection URIs that are registered at i for the client id c only point to domains of r (and use HTTPS).

We therefore know that b will send an HTTPS request (say m) containing t to r. We have to check whether r or a script delivered by r to b will leak t. Algorithm 17 processes all HTTPS requests delivered to r. As i redirected b using the 303 status code, the request to r must be a GET request. Hence, r does not process this request in Lines 6ff. of Algorithm 17. Lines 2ff. do only respond with a script and do not use t in any way. We are left with Lines 13ff. to be analyzed.

As in m the id token t is always contained in a dictionary under the key $\mathtt{id\_token}$ and this dictionary is either in the parameters, the fragment, or the body of m, it is now easy to see that r does not store or send out t in any way. We now have to check if a script delivered by r to b leads to t being leaked. First note that r always sets the header $\mathtt{ReferrerPolicy}$ to $\mathtt{origin}$ in every HTTP(S) response r sends out. Hence, t can never leak using the Referer header.

There are only two scripts that r may deliver: (1) The script *script_rp_index* either issues a $\mathtt{FORM}$ command to the browser, which does not contain t, or this script issues a $\mathtt{HREF}$ command to the browser for some URL, which also does not contain t. (2) The script *script_rp_get_fragment* takes the fragment of the current URL (which may be a dictionary that contains t under the key $\mathtt{id\_token}$) and the $\mathtt{iss}$ parameter and issues an HTTPS request to r for the path $/\mathtt{redirect\_ep}$, which will be processed by r in Lines 13ff. of Algorithm 17. Now, the same reasoning as above applies.

ID token was created in Lines 92ff.

In this case, the id token is created by i only when an HTTPS request was received by i that matches the following criteria: (a) it must be for the path $/\mathtt{token}$, (b) it must contain the client id c in the body (under the key $\mathtt{client\_id}$), and (c) it must contain a authorization code in the body (under the key $\mathtt{code}$) that occurs in one of i's internal records with a matching subject, issuer, and nonce. To be more precise, the request must contain a code *code* such that there is a record *rec* with $rec \in^{\langle\rangle} S(i).\mathtt{records}$ and $rec[\mathtt{issuer}] \equiv d$, $rec[\mathtt{subject}] \equiv id$, $rec[\mathtt{client\_id}] \equiv c$, and $rec[\mathtt{code}] \equiv c$. Such a record can only be created and the authorization code *code* issued under exactly the same circumstances that allow an id token (of the above form) to be created in Line 92. With exactly the same reasoning as above, this time for the code instead of the id token, we can follow that *code* does not leak to the attacker.

We have therefore shown that no attacker process can get hold of the id token t. This proves the lemma. ■

Assumption 1. There exists an OIDC web system $\mathcal{OIDC}^n$ with a network attacker such that there exists a run ρ of $\mathcal{OIDC}^n$, a configuration (S,E,N) in ρ, some $r \in \mathsf{RP}$ that is honest in S, some identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id)$ being an honest IdP (in S) and $\mathsf{ownerOfID}(id)$ being an honest browser (in S), some service session identified by some nonce n for id at r, and n is derivable from the attackers knowledge in S (i.e., $n \not\in d_\emptyset(S(\mathsf{attacker}))$).

Lemma 7. Assumption 1 is a contradiction.

PROOF. We first recall how the service session identified by some nonce n for id at r is defined. It means that there is some session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ with $S(r).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d, id \rangle$ and $S(r).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$. Now the assumption is that n is derivable from the attacker's knowledge. Since we have that $S(r).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$, we can check where and how, in general, service session ids can be created. It is easy to see that this can only happen in Algorithm 20, where, in Line 18, the RP chooses a fresh nonce as the value for the service session id, in this case x. In the line before, it sets the value for $S(r).\mathtt{sessions}[x][\mathtt{loggedInAs}]$, in this case $\langle d, id \rangle$. In the Lines 9ff., r performs several checks to ensure the integrity and authenticity of the id token.

The function function CHECK_ID_TOKEN can be called in either (a) Line 42 of Algorithm 17 or (b) in Line 28 of Algorithm 16.

We can now distinguish between these two cases.

Case (a).

In this case, we can easily see that the same party that finally receives the service session id x, must have provided, in an HTTPS request, an id token (say, t') with the following properties (for some $l < j$):

$$\mathsf{extractmsg}(t')[\mathtt{iss}] \equiv d$$

$$\mathsf{extractmsg}(t')[\mathtt{sub}] \equiv id$$

$$\mathsf{extractmsg}(t')[\mathtt{aud}] \equiv S^l(r).\mathtt{clientCredentialsCache}[d][\mathtt{client\_id}]$$

$$\mathsf{checksig}(t', \mathsf{pub}(S^l(i).\mathtt{jwks})) \equiv \top \ .$$

The attacker (and, by extension, any other party except for i, b, and r), however, cannot know such an id token (see Lemma 6). Since r and i do not send requests to r, the id token must have been sent by b to r. As the service session id x is only contained in a set-cookie header with the httpOnly and secure flags set, b will only ever send the service session id x to r (contained in a cookie header). As b does not leak x in any other way and as r does not leak information sent in cookie headers, the service session id x does not leak.

Case (b).

Otherwise, the party that finally receives the service session id x needs to provide a code c such that, when this code is sent to the token endpoint of i (Algorithm 18), i responds with an id token matching the criteria listed in Case (a). This, however, would mean that an attacker, knowing this code, could do the same, violating Lemma 6. (Note that for every run where a client secret is associated with the client id there is also a run where the client secret is not used; the client secret does not prevent the attacker from requesting an id token at the token endpoint for a valid code.)

We therefore have shown that the attacker cannot know x, proving the lemma and showing that Assumption 1 is, in fact, a contradiction. ∎

D. Proof of Authorization

As above, we assume that there exists an OIDC web system that is not secure w.r.t. authorization and lead this to a contradiction.

Assumption 2. There exists an OIDC web system with a network attacker $\mathcal{OIDC}^n$, a run ρ of $\mathcal{OIDC}^n$, a state (S^j, E^j, N^j) in ρ, a relying party $r \in \mathsf{RP}$ that is honest in S^j, an identity provider $i \in \mathsf{IdP}$ that is honest in S^j, a browser b that is honest in S^j, an identity $id \in \mathsf{ID}^i$ owned by b, a nonce n, a term $x \in^{\langle\rangle} S^j(i).\mathtt{records}$ with $x[\mathtt{subject}] \equiv id$, $n \in^{\langle\rangle} x[\mathtt{access\_tokens}]$, and the client id $x[\mathtt{client\_id}]$ has been issued by i to r, and n is derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$).

Lemma 8. Assumption 2 is a contridiction.

PROOF. We have that $n \in d_\emptyset(S^j(\mathsf{attacker}))$ and therefore, there must have been a message from a third party to attacker (or any other corrupted party, which could have forwarded n to the attacker) that contained n. We can now distinguish between the parties that could have sent n to the attacker (or to the corrupted party):

The access token n was sent by the browser b:

We now track different cases in which the access token n can get into b's knowledge. We will omit the cases in which b learns n from any dishonest party as in such a case there is a different run ρ' of $\mathcal{OIDC}^n$ in which this dishonest party immediately sends n to the attacker.

(I) First, we analyze the case in which b has learned n from an honest (in S^j) identity provider, say i'. In this case, b must have received an HTTPS response from i' (honest identity providers do not send out unencrypted HTTP responses). Honest identity providers send out HTTPS responses in Lines 6, 14, 17, 27, 72, and 98 of Algorithm 24 and Line 17 of Algorithm 25. It is easy to see that i' does not send out n in Lines 6, 14, 17, and 27 of Algorithm 24 and Line 17 of Algorithm 25 (given that the attacker does not know n), leaving Lines 72, and 98 of Algorithm 24 to analyze.

(a) If i' sends out n in Line 72 of Algorithm 24, b must have sent an HTTPS POST request bearing an Origin header for one of the domains of i' to i'. As i' only delivers the script *script_idp_form*, only this script could have caused this request (using a FORM command). Hence, b will navigate the corresponding window to the location indicated in the Location header of the HTTPS response assembled in Lines 28ff. of Algorithm 24. The body of this response can consist of an authorization code (a fresh nonce), an access token (a fresh nonce), and an id token consisting of one domain of i, a valid user name for i', a client id, and a nonce (say n') from the request.

We now reason why i' must be i, and the access token in the response must be n. In the id token, only the client id and the nonce n' could be n. As the client id is always set by the attacker during registration, the client id cannot be n. The nonce n' originates from the request sent by b on the command of *script_idp_form*. In this request, the nonce n' must be contained in the URL, which is the URL from which the script was loaded before. Hence, the browser must have been navigated to this URL. As the attacker does not know n at this point, only honest scripts or honest web servers could have navigated the browser to such an URL (containing n). Honest relying parties only populate the parameter nonce (bearing n') in such a redirect with a fresh nonce, honest identity providers do not populate such an URL parameter by themselves, but could have used this parameter in a redirect based on a registered redirect URL. As honest parties never register such a redirect URL, n' cannot be n. Hence, only the access token in the response above can be n. As the access token is a fresh nonce, we must have that i' is i and that i creates the term $x \in^{\langle\rangle} S^j(i).\mathtt{records}$ with $x[\mathtt{subject}] \equiv id$, $n \in^{\langle\rangle} x[\mathtt{access\_tokens}]$ (i will never create such a term at any other time), and the client id $x[\mathtt{client\_id}]$ has been issued by i to r. Hence, the location redirect issued by i must point to an URL of r with the path /redirect_ep (see Lemma 4) and this URL contains the parameter iss with a domain of i. The access token n is only contained in the fragment of this URL under the key access_token.

Now, b sends an HTTPS request to r. This request does not contain n (as it is placed in the fragment part of the URL). The relying party r can (regardless of the path) send out only the scripts *script_rp_index* and *script_rp_get_fragment* as a response to such a request. The script *script_rp_index* ignores the fragment of its URL. The script *script_rp_get_fragement* takes the fragment of the URL and uses it as the body of a POST request to its own origin (which is r) with path /redirect_uri. When r processes this POST request, r only ever uses n in Line 45 of Algorithm 17. There, the access token n and the value of the parameter iss (a domain of i) is processed by Algorithm 19. From Lemma 2, we know that r will only send n to the token endpoint of i in an HTTPS request. This request is then processed by i in Lines 73ff. of Algorithm 24. There, i only checks n, but does not send out n.

If b sends out a response, the same reasoning as above applies. Hence, we have that n does not leak to the attacker in this case.

(b) If i' sends out n in Line 98 of Algorithm 24, we have that the response does not contain a script or a redirect. The browser would only interpret such a response if the request was caused by an XMLHTTPREQUEST command of a script. Honest scripts do not issue such a command, leaving only the attacker script as the only possible source for such a request. If i' is not i, it is easy to see that this response cannot contain n. The identity provider i only sends out n (taken from the subterm records from its state) if the request contains a valid authorization code for this access token. With the same reasoning as for the authentication property above, the attacker cannot know a valid id token for any user id owned by b. If the attacker would know a valid authorization code, he could retrieve a valid id token (for such a user id) from i. Hence, the attacker cannot know a valid authorization code. As this reasoning also applies for the attacker script, the attacker script could not have caused a request to i revealing n.

(II) Now, we analyze the case in which b received n from some honest (in S^j) relying party, say r'. In this case, b must have received an HTTPS response from r' (honest relying parties do not send out unencrypted HTTP responses). Honest relying parties only send out such HTTPS responses in Lines 5 and 29 of Algorithm 17, Line 23 of Algorithm 20, and Line 39 of Algorithm 21. In the former three cases, r' only sends out fixed information and fresh nonces (either chosen by r' directly before sending out the message or the HTTPS nonce and key chosen by b when creating the request). In the latter case, r' (besides the pieces of information as before) also adds information from its OpenID Connect configuration cache (i.e., client id and authorization URL). From Lemma 2 and 4 we know that if r' gathered this information from an honest party, this information cannot contain n. As the attacker does not know n at this point, this registration information cannot contain n if r' gathered this information from a dishonest party. Hence, b cannot have learned n from any r'.

(III) b cannot have learned n from a different honest (in S^j) browser as honest browsers do not create messages that can be interpreted by honest browsers.

(IV) b cannot have learned n from the attacker, as the attacker does not know n at this point.

The access token n was sent by the IdP i: We can see that access tokens are sent by the IdP only after a request to the path (endpoints) /auth2 or to the path and /token.

In case of a request to the path /auth2, a pair of access tokens is created and the first access token in the pair is returned from the endpoint. If the attacker would be able to learn n from this endpoint such that there exists a record $x \in^{\langle\rangle} S^j(i).\mathtt{records}$ with $x[\mathtt{subject}] \equiv id$, then the attacker would need to provide the user's credentials to the IdP i. The attacker cannot know these credentials (Lemma 5), therefore the attacker cannot request n from this endpoint.

In case of a request to the path `/token`, the attacker would need to provide an authorization code that is contained in the same record (in this case x) as n. Now, recall that we have that $x[\mathtt{subject}] \equiv id$ and $c := x[\mathtt{client\_id}]$ has been issued to r by i. We can now see that if the attacker would be able to send a request to the endpoint `/token` which would cause a response that contains n, the attacker would also be able to learn an id token of the form shown in Lemma 6 (the issuer is a domain of i, the subject is id, and the audience is c). This would be a contradiction to Lemma 6.

We can conclude that the access token n was not sent by the IdP i.

The access token n was sent by the RP r:

The only place where the (honest) RP uses an access token is in Algorithm 19. There, the access token is sent to the domain of the token endpoint (compare Algorithm 18, where the authorization code is sent to that endpoint). We can now see that the access token is always sent to i: If the access token would be sent to the attacker, so would the authorization code in Algorithm 18, and Lemma 6 would not hold true.

E. Proof of Session Integrity

Before we prove this property, we highlight that in the absence of a network attacker and with the DNS server as defined for $OI\mathcal{DC}^w$, HTTP(S) requests by (honest) parties can only be answered by the owner of the domain the request was sent to, and neither the requests nor the responses can be read or altered by any attacker unless he is the intended receiver.

We further show the following lemma, which says that an attacker (under the assumption above) cannot learn a *state* value that is used in a login session between an honest browser, an honest IdP, and an honest RP.

Lemma 9 (Third parties do not learn state). There exists no run ρ of an OIDC web system with web attackers $OI\mathcal{DC}^w$, no configuration (S,E,N) of ρ, no $r \in \mathsf{RP}$ that is honest in S, no $i \in \mathsf{IDP}$ that is honest in S, no browser b that is honest in S, no nonce $lsid \in \mathcal{N}$, no domain $h \in \mathsf{dom}(r)$ of r, no terms g, x, y, $z \in \mathcal{T}_{\mathcal{N}}$, no cookie $c := \langle \mathtt{sessionId}, \langle lsid, x, y, z \rangle \rangle$, no atomic DY process $p \in \mathcal{W} \setminus \{b,i,r\}$ such that (1) $S(r).\mathtt{sessions}[lsid] \equiv g$, (2) $g[\mathtt{state}] \in d_{\emptyset}(S(p))$, (3) $S(r).\mathtt{issuerCache}[g[\mathtt{identity}]] \in \mathsf{dom}(i)$, and (4) $c \in^{\langle\rangle} S(b).\mathtt{cookies}[h]$.

PROOF. To prove Lemma 9, we track where the login session identified by *lsid* is created and used.

Login session ids are only chosen in Line 10 of Algorithm 17. After the session id was chosen, its value is sent over the network to the party that requested the login (in Line 39 of Algorithm 21). We have that for *lsid*, this party must be b because only r can set the cookie c for the domain h in the state of b[19] and Line 39 of Algorithm 21 is actually the only place where r does so.

Since b is honest, b follows the location redirect contained in the response sent by r. This location redirect contains *state* (as a URL parameter). The redirect points to some domain of i. (This follows from Lemma 2.) The browser therefore sends (among others) *state* in a GET request to i. Of all the endpoints at i where the request can be received, the authorization endpoint is the only endpoint where *state* could potentially leak to another party. (For all other endpoints, the value is dropped.) If the request is received at the authorization endpoint, *state* is only sent back to b in the initial scriptstate of *script_idp_form*. In this case, the script sends *state* back to i in a POST request to the authorization endpoint. Now, i redirects the browser b back to the redirection URI that was passed alongside *state* from r via the browser to i. This redirection URI was chosen in Line 2 of Algorithm 21 and therefore points to one of r's domains. The value *state* is appended to this URI (either as a parameter or in the fragment). The redirection to the redirection URI is then sent to the browser b. Therefore, b now sends a GET request to r.

If *state* is contained in the parameter, then *state* is immediately sent to r where it is compared to the stored login session records but neither stored nor sent out again. In each case, a script is sent back to b. The scripts that r can send out are *script_rp_index* and *script_rp_get_fragment*, none of which cause requests that contain *state* (recall that we are in the case where *state* is contained in the URI parameter, not in the fragment). Also, since both scripts are always delivered with a restrictive Referrer Policy header, any requests that are caused by these scripts (e.g., the start of a new login flow) do not contain *state* in the referer header.[20]

If *state* is contained in the fragment, then *state* is not immediately sent to r, but instead, a request without *state* is sent to r. Since this is a GET request, r either answers with a response that only contains the string `ok` but no script (Lines 23ff. of Algorithm 20), a response containing *script_rp_index* (Lines 2ff. of Algorithm 17), or a response containing *script_rp_get_fragment* (Line 29 of Algorithm 17). In case of the `ok` response, *state* is not used anymore by the browser. In case of *script_rp_index*, the fragment is not used. (As above, there is no other way in which *state* can be sent out, also because the fragment part of an URL is stripped in the referer header.) In the case of *script_rp_get_fragment* being loaded into the browser, the script sends *state* in the body of an HTTPS request to r (using the path `/redirect_ep`). When r receives this request, it does not send out *state* to any party (see Lines 13ff. of Algorithm 17).

This shows that *state* cannot be known to any party except for b, i, and r. ■

[19] Note that we have only web attackers.

[20] We note that, as discussed earlier, without the Referrer Policy, *state* could leak to a malicious IdP or other parties.

Proof of Session Integrity for Authentication. To prove that every OIDC web system with web attackers is secure w.r.t. session integrity for authentication, we assume that there exists an OIDC web system with web attackers which is not secure w.r.t. session integrity for authentication and lead this to a contradiction.

Assumption 3. There exists an $\mathcal{OIDC}^w$ be an OIDC web system with web attackers, a run ρ of $\mathcal{OIDC}^w$, a processing step Q in ρ with

$$Q = (S, E, N) \to (S', E', N')$$

(for some S, S', E, E', N, N'), a browser b that is honest in S, an IdP $i \in \mathsf{IdP}$, an identity u that is owned by b, an RP $r \in \mathsf{RP}$ that is honest in S, a nonce *lsid*, with $\mathsf{loggedIn}_\rho^Q(b, r, u, i, \mathit{lsid})$ and (1) there exists no processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, r, \mathit{lsid})$, or (2) i is honest in S, and there exists no processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, r, u, i, \mathit{lsid})$.

Lemma 10. Assumption 3 is a contradiction.

Proof. **(1)** We have that $\mathsf{loggedIn}_\rho^Q(b, r, u, i, \mathit{lsid})$. With Definition 56 we have that r sent out the service session id belonging to *lsid* to b. (This can only happen when the function CHECK_ID_TOKEN (Algorithm 20) was called with *lsid* as the first parameter.) This means that r must have received a request from b containing a cookie with the name `sessionId` and the value *lsid*: The response by r (which we know was sent to b) was sent in Line 23 in Algorithm 20. There, r looks up the address of b using the login session record under the key `redirectEpRequest`. This key is only ever created in Line 35 of Algorithm 17. This line is only ever called when r receives an HTTPS request from b with the cookie as described.

We can now track how the cookie was stored in b: Since the cookie is stored under a domain of r and we have no network attacker, the cookie must have been set by r. This can only happen in Line 39 in Algorithm 21. Similar to the `redirectEpRequest` session entry above, r sends this cookie as a response to a stored request, in this case, using the key `startRequest` in the session data. This key is only ever created in Lines 6ff. of Algorithm 17. Hence, there must have been a request from b to r containing a POST request for the path `/startLogin` with an origin header for an origin of r. There are only two scripts which could potentially send such a request, *script_rp_index* and *script_rp_get_fragment*. It is easy to see that only the former send requests of the kind described. We therefore have a processing step Q' that happens before Q in ρ with $\mathsf{started}_\rho^{Q'}(b, r, \mathit{lsid})$.

(2) Again, we have that $\mathsf{loggedIn}_\rho^Q(b, r, u, i, \mathit{lsid})$. Now, however, i is honest.

We first highlight that if r receives an HTTPS request, say m, which contains *state* such that $S(r).\mathtt{sessions}[\mathit{lsid}][\mathtt{state}] \equiv \mathit{state}$ and contains a cookie with the name `sessionId` and the value *lsid* then this request must have come from the browser b and be caused by a redirection from i or a script from r. From $\mathsf{loggedIn}_\rho^Q(b, r, u, i, \mathit{lsid})$ it follows that there is a term g such that $S(r).\mathtt{sessions}[\mathit{lsid}] \equiv g$, and $g[\mathtt{loggedInAs}] \equiv \langle d, u\rangle$ with $d \in \mathsf{dom}(i)$. From the Algorithm 20 we have that $S(r).\mathtt{issuerCache}[g[\mathtt{identity}]] \equiv d$. With Lemma 9 we have that only b, r, and i know *state*.

We can now show that m must have been caused by i by means of a Location redirect that was sent to b or by the script *script_rp_get_fragment*. First, neither r nor i send requests that contain cookies. The request must therefore have originated from b. Since no attacker knows *state*, the request cannot have been caused by any attacker scripts or by redirects from parties other than r or i (otherwise, there would be runs where the attacker learns *state*).

Redirects from r can be excluded, since r only sends a redirection in Line 39 in Algorithm 21 but there, a freshly chosen state value is used, hence, there is only one processing step in which r uses *state* for this redirect. This is the processing step where r adds *state* to the session data stored under the key *lsid*. Since this is a session in which the honest IdP i is used, and with Lemma 2, we have that r does not redirect to itself (but to i instead).

The scripts *script_rp_index* and *script_idp_form* do not send requests with the *state* parameter. Therefore, the remaining causes for the request m are either the script *script_rp_get_fragment* or a location redirect from i.

If the request m was caused by *script_rp_get_fragment*, then it is easy to see from the definition of *script_rp_get_fragment* (Algorithm 23) that this script only sends data from the fragment part of its own URI (except for the `iss` parameter) and it sends this data only to its own origin. This script therefore must have been sent to b by r, which only sends this script after receiving HTTPS request to the redirection endpoint (`/redirect_ep`). With the same reasoning as above this must have been caused by a location redirect from i.

For clarity, by m_redir we denote the response by i to the browser b containing this redirection. We now show that for m_redir to take place, there must have been a processing step Q'' (before Q) with $\mathsf{authenticated}_\rho^{Q''}(b, r, u', i, \mathit{lsid})$ for some identity u'.

In the honest IdP i, there is only one place where a redirection happens, namely in Line 72 in Algorithm 24. To reach this point, i must have received the login data for u' in the HTTPS request corresponding to m_redir. This must be a POST request with an origin header containing an origin of i. As i only uses *script_idp_form*, the request must have been caused by this script. Hence, we have $\mathsf{authenticated}_\rho^{Q''}(b, r, u', i, \mathit{lsid})$.

We now only need to show that $u' = u$.

With $\mathsf{loggedIn}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$, we know that r must have called the function $\mathsf{CHECK\_ID\_TOKEN}$ (Algorithm 20). We further have that $S(r).\mathtt{sessions}[\mathit{lsid}][\mathtt{loggedInAs}] \equiv \langle d,u\rangle$. We therefore have that i must have created an id token with the issuer d and the identity u. $\mathsf{CHECK\_ID\_TOKEN}$ can be called in Line 28 in Algorithm 16 and in Line 42 in Algorithm 17. We now distinguish between these two cases.

$\mathsf{CHECK\_ID\_TOKEN}$ *was called in Line 28 in Algorithm 16:* When the function is called in this line, there must have been an HTTPS request reference with the string $\mathtt{TOKEN}$ (cf. generic web server model, Algorithm 10). Such a reference is only created in Line 16 of Algorithm 18. With Lemma 2 we know that this HTTPS request was sent to the token endpoint (path $\mathtt{/token}$) of i (because the issuer, stored in the login session record, is i). Since the token endpoint returned an id token of the form described above, and i is honest, there must have been a record in i, say v, with $v[\mathtt{subject}] \equiv u$. In the request to the token endpoint, r must have sent a nonce c such that $v[\mathtt{code}] \equiv c$. This request, as already mentioned, must have been sent in Line 16 of Algorithm 18. This means, that there must have been an HTTPS request to i containing the session id lsid as a cookie, c, and state. Such a request can only be the request m as shown above, hence there must have been the HTTPS response m_{redir} containing the values c and state. Recall that we have the record v as shown above in the state of i. Such a record is only created in i if $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,\mathit{lsid})$. Therefore, $u = u'$ in this case.

$\mathsf{CHECK\_ID\_TOKEN}$ *was called in Line 42 in Algorithm 17:* In this case, the id token must have been contained in m and m_{redir} as above. Such an id token is only sent out in m_{redir} by i if $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,\mathit{lsid})$. Therefore, $u = u'$ in every case. ■

Proof of Session Integrity for Authorization. To prove that every OIDC web system with web attackers is secure w.r.t. session integrity for authorization, we assume that there exists an OIDC web system with web attackers which is not secure w.r.t. session integrity for authorization and lead this to a contradiction.

Assumption 4. There is a OIDC web system $\mathcal{OIDC}^{w}$ with web attackers, a run ρ of $\mathcal{OIDC}^{w}$, a processing step Q in ρ with

$$Q = (S,E,N) \to (S',E',N')$$

(for some S, S', E, E', N, N') a browser b that is honest in S, an IdP $i \in \mathsf{IdP}$, an identity u that is owned by b, an RP $r \in \mathsf{RP}$ that is honest in S, a nonce lsid, with (1) $\mathsf{usedAuthorization}^{Q}_{\rho}(b,r,i,\mathit{lsid})$ and there exists no processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b,r,\mathit{lsid})$, or (2) i is honest in S and $\mathsf{actsOnUsersBehalf}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$ and there exists no processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,\mathit{lsid})$.

Lemma 11. Assumption 4 is a contradiction.

PROOF. **(1)** We have that $\mathsf{usedAuthorization}^{Q}_{\rho}(b,r,i,\mathit{lsid})$. With Definition 59 we have that r sent out the access token belonging to lsid to i. This can only happen when the function $\mathsf{USE\_ACCESS\_TOKEN}$ (Algorithm 19) was called with lsid. This function is called in Line 45 of Algorithm 17 and in Line 27 of Algorithm 16.

In both cases, there must have been a request, say m, to r containing a cookie with the session id lsid. In the former case, this is the request that is processed in the same processing step as calling the function $\mathsf{USE\_ACCESS\_TOKEN}$. In the latter case, there must have been an HTTPS request reference with the string $\mathtt{TOKEN}$ (cf. generic web server model, Algorithm 10). Such a reference is only created in Line 16 of Algorithm 18. To get to this point in the algorithm, a request as described above must have been received. Since we have web attackers (and no network attacker), it is easy to see that this request must have been sent by b. With the same reasoning as in the proof for session integrity for authentication, we now have that there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b,r,\mathit{lsid})$.

(2) We have that i is honest and $\mathsf{actsOnUsersBehalf}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$. From (1) we know that r must have received a request m from b containing a cookie with the session id lsid. Therefore, we know that m_{redir} exists just as in the proof for Lemma 10 (2). As in that proof, we have that $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u',i,\mathit{lsid})$ for some identity u'. We therefore need to show that $u = u'$.

With $\mathsf{actsOnUsersBehalf}^{Q}_{\rho}(b,r,u,i,\mathit{lsid})$, we know that r must have called the function $\mathsf{USE\_ACCESS\_TOKEN}$ with some access token t (Algorithm 19). We further have that there is a term g such that $g \in^{\langle\rangle} S(i).\mathtt{records}$ with $t \in^{\langle\rangle} g[\mathtt{access\_tokens}]$ and $g[\mathtt{subject}] \equiv u$.

$\mathsf{USE\_ACCESS\_TOKEN}$ can be called in Line 27 in Algorithm 16 and in Line 45 in Algorithm 17. We now distinguish between these two cases.

$\mathsf{USE\_ACCESS\_TOKEN}$ *was called in Line 27 in Algorithm 16:* When the function is called in this line, there must have been an HTTPS request reference with the string $\mathtt{TOKEN}$ (cf. generic web server model, Algorithm 10). Such a reference is only created in Line 16 of Algorithm 18. With Lemma 2 we know that this HTTPS request was sent to the token endpoint (path $\mathtt{/token}$) of i (because the issuer, stored in the login session record, is i). Since the token endpoint returned the access token t, and i is honest, there must have been a record in i, say v, with $v[\mathtt{subject}] \equiv u$. In the request to the token endpoint, r must have sent a nonce c such that $v[\mathtt{code}] \equiv c$. This request, as already mentioned, must have been sent in Line 16 of

Algorithm 18. This means, that there must have been an HTTPS request to i containing the session id $lsid$ as a cookie, c, and $state$. Such a request can only be the request m as shown above, hence there must have been the HTTPS response m_{redir} containing the values c and $state$. Recall that we have the record v as shown above in the state of i. Such a record is only created in i if $\mathsf{authenticated}_{\rho}^{Q''}(b,r,u,i,lsid)$. Therefore, $u = u'$ in this case.

$\mathsf{USE\_ACCESS\_TOKEN}$ *was called in Line 45 in Algorithm 17:* In this case, the access token t must have been contained in m and m_{redir} as above. This access token is only sent out in m_{redir} by i if $\mathsf{authenticated}_{\rho}^{Q''}(b,r,u,i,lsid)$. Therefore, $u = u'$ in every case.

F. *Proof of Theorem 1*

With Lemmas 7, 8, 10, and 11, Theorem 1 follows immediately.