Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Formal Analysis of Self-Issued OpenID Providers

Christina Bauer

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Pedram Hosseyni, M.Sc.
Commenced:	October 4, 2021
Completed:	April 4, 2022

Abstract

The Self-Issued OpenID Provider specification is an extension of OpenID Connect that allows users to authenticate at Relying Parties using an Identity Provider under the local control of the user.

For this, the Self-Issued OpenID Provider specification introduces a notion of an identity that is self-asserted by the user. To supplement the self-asserted claims of the user with verifiable claims by a trusted party, a natural extension for the Self-Issued OpenID Provider specification is the use of Verifiable Credentials. The Self-Issued OpenID Provider specification defines two protocol flows: the same-device flow and the cross-device flow where the latter is known to be vulnerable to authentication request replay.

In this thesis, we analyze the Self-Issued OpenID Provider specification using the Web Infrastructure Model. We model the Self-Issued OpenID Provider same-device protocol flow and uncover an attack on the protocol during formal analysis. After mitigating this attack in the model, we show that the security properties authentication, session integrity, and holder binding for Verifiable Credentials hold under certain assumptions.

We also developed a variant of the cross-device flow that mitigates the request replay attack on the cross-device flow at the cost of introducing a web service associated with the Self-Issued OpenID Provider. For our variant of the cross-device protocol flow, we show that it is secure with respect to the authentication security property under suitable assumptions.

Kurzfassung

Self-Issued OpenID Provider sind eine Erweiterung von OpenID Connect, die es Nutzern erlaubt, sich bei einer Relying Party mithilfe eines Identity Provider auf ihrem Endgerät zu authentifizieren.

Für die Authentifizierung mit Self-Issued OpenID Providern definiert die Spezifikation Identitäten, deren Besitz der Nutzer beweisen kann. Um diese Identitäten mit prüfbaren Angaben einer vertrauenswürdigen Partei zu ergänzen, ist die zusätzliche Verwendung von Verifiable Credentials eine natürliche Erweiterung für Self-Issued OpenID Provider. In der Self-Issued OpenID Provider Spezifikation werden zwei Nachrichtenflüsse definiert, der Same-Device Flow und der Cross-Device Flow, wobei für den letzteren ein Replay Angriff bekannt ist.

Im Rahmen dieser Masterarbeit führen wir eine formale Analyse der Self-Issued OpenID Provider Spezifikation im Web Infrastructure Model durch. Wir modellieren den Self-Issued OpenID Provider Same-Device Flow, beschreiben einen Angriff, den die formale Analyse aufdeckt, und zeigen, wie der Angriff verhindert werden kann. Für das angepasste Protokoll zeigen wir, dass Sicherheit bezüglich der Eigenschaften Authentifizierung, Session Integrity und Holder Binding von Verifiable Credentials erreicht wird.

Wir beschreiben und modellieren zudem eine Variante des Cross-Device Flow, die im Rahmen dieser Arbeit entwickelt wurde. Für den Preis eines zusätzlichen Webservices erschwert die Variante den Replay Angriff auf den Cross-Device Flow erheblich. Für unsere Variante des Cross-Device Flow zeigen wir, dass sie sichere Authentifizierung ermöglicht.

Contents

List of Figures

List of Algorithms

Acronyms

DID Decentralized Identifier. 19

DY Dolev-Yao. 37

JWK JSON Web Key. 19

JWS JSON Web Signature. 17

JWT JSON Web Token. 17

OP OpenID Provider. 15

RP Relying Party. 15

Self-Issued OP Self-Issued OpenID Provider. 15

VDR Verifiable Data Registry. 20

WIM Web Infrastructure Model. 15

1 Introduction

Authentication, i.e., determining the identity of a user, plays a key role in the security of web applications. OpenID Connect [16] is a widely used protocol for delegating authentication. Utilizing OpenID Connect a Relying Party (RP) can authenticate a user using a third party, the OpenID Provider (OP). For this, both the RP and the user need to place great trust in the OP to act benevolently.

If the OP is unavailable or even discontinues its service, the user can no longer utilize its previous identity assured by the OP at the RP. Additionally, the convenience, that logging in via the OP provides, is paid for in additional messages being sent. This might pose efficiency problems in environments with low connectivity to the internet such as in mobile environments. As an approach to mitigate the shortcomings of traditional OpenID Connect for appropriate Use Cases, the OpenID Foundation is developing an extension of OpenID Connect with OPs that are under the user's local control, so-called Self-Issued OpenID Providers (Self-Issued OPs) [24]. Using Self-Issued OPs, trust is negotiated directly between the RP and the Self-Issued OP as the agent of the user.

As users can not in general establish a similar level of trust with an RP as a centralized OP, the use of verifiable claims by a third party is a natural extension for the use of Self-Issued OPs. Verifiable Credentials [18] describe a format that can be utilized to exchange such claims and allow validation of them. The use of OpenID Connect with Verifiable Credentials is described in [21].

The Self-Issued OP specification describes two versions of the Self-Issued OP protocol flow: the same-device flow and the cross-device flow. In the same-device flow, the Self-Issued OP application resides on the same device as the browser with the session the user wishes to authenticate at the RP, while in the cross-device flow these devices may be distinct. By the security considerations of [24], however, the cross-device flow is vulnerable to a request replay attack that breaks authentication. Note that throughout this thesis we refer to [24] as the Self-Issued OpenID Provider specification.

This thesis strives to provide a formal security analysis of the Self-Issued OpenID Provider specification together with the use of Verifiable Credentials. We model the components of the Self-Issued OpenID Provider specification in the Web Infrastructure Model (WIM) [10], define security properties on this model and prove them for our analysis. The WIM is a formal model of web infrastructure that was first used to analyze the BrowserID single sign-on system [7]. Using the WIM, further analyses were conducted on the SPRESSO single sign-on system [9], OAuth 2.0 [8], OpenID Connect [6], the OpenID Financial-Grade API [5], and the W3C Web Payment APIs [4], where each analysis uncovered attacks.

In this thesis, we first give an overview of the Self-Issued OpenID Provider extension of OpenID Connect in Chapter 2. In Chapter 3, we describe the known request replay attack on the cross-device flow and describe a variant of the cross-device flow, that hardens the flow against the known replay attack, that we developed. In Chapter 4, we describe a vulnerability of the Self-Issued OP same-device flow that our analysis uncovered. We provide an overview of the WIM in general

(Chapter 5) and our models of Same-Device Self-Issued OP and our variant of Cross-Device Self-Issued OP (Chapter 6). We give informal descriptions of the security properties we analyze in Chapter 7 and summaries of the proofs of the properties in Chapter 8. The appendix of this thesis contains the formal security analysis. In Appendix A, we introduce the definitions of data structures that we use in the models. We give the full formal models of the Self-Issued OP same-device flow (Appendix B) and our cross-device flow variant (Appendix C). In Appendix D, we provide formal security property definitions and, in Appendix E, we give the proofs of these properties.

2 Self-Issued OpenID Providers

The specification of Self-Issued OpenID Providers [24] extends OpenID Connect with the notion of an OpenID Provider that resides within the control of the user, as an application on the user device. With this so called 'Self-Issued OpenID Provider' the user can (re-)authenticate at the RP without the need for a third party to function as an identity provider for them.

2.1 OpenID Connect

OpenID Connect [16] is a protocol build on top of OAuth 2.0 [11] that allows an RP to authenticate a user via an OP that serves as an Identity Provider. For this, OpenID Connect supports three protocol flows, one of which is the Implicit Flow that the Self-Issued OP specification extends.

2.1.1 ID Token

A core concept of OpenID Connect is the so-called ID token, a collection of claims about the user issued by the OP that the OP provides to an RP. In particular the ID token contains a claim about the identity of the user. The ID token in OpenID Connect is represented as a JWT [13]. In OpenID Connect the ID token is cryptographically signed by the OP using JWS [12]. The RP verifies the signature of each ID token it receives using a public key made available by the OP.

Each ID token issued by an OP for an RP contains an identifier of the OP, an identifier of the user, an intended audience including the RP, information about the timing of the authentication process and possibly other optional claims.

2.1.2 OpenID Provider

The OP is an OAuth 2.0 authorization server that supports authentication of the user for the RP via OpenID Connect.

To enable this the OP provides an endpoint for authentication. When this authentication endpoint receives a request, the OP authenticates the user and negotiates consent with them. Then the OP creates an ID token containing information about the user for the RP.

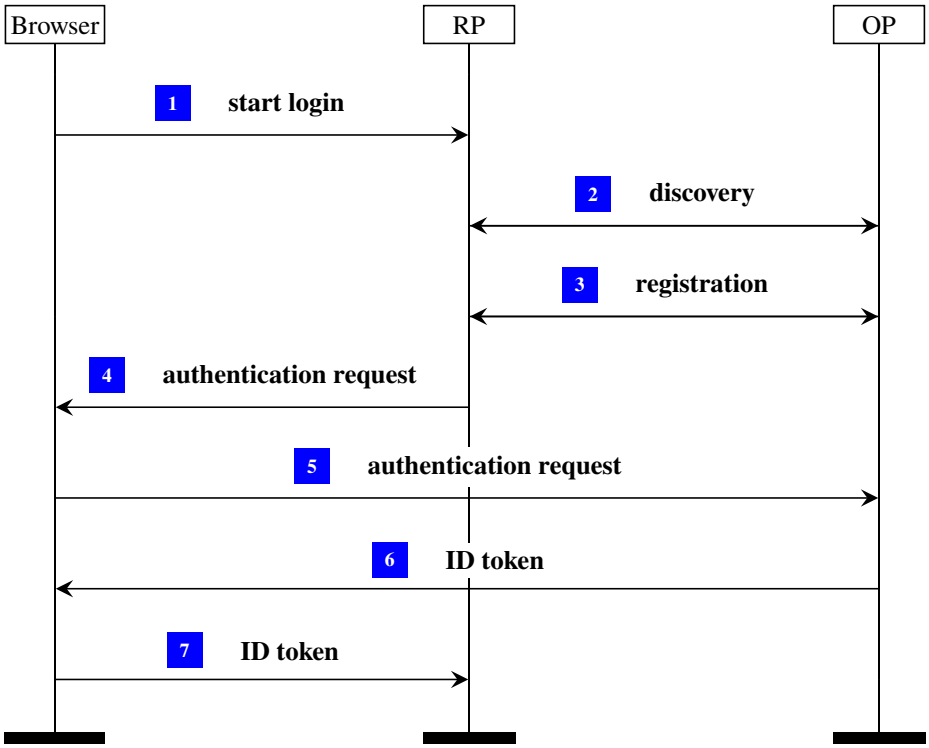

Figure 2.1: OpenID Connect Implicit Flow – Overview

2.1.3 Relying Party

The RP is an OAuth 2.0 client that wishes to authenticate the user and supports authentication using OpenID Connect. For this the RP sends an authentication request to the OP, when the user wishes to log in at the RP. On receiving the ID token from OP the validates its contents and acknowledges the user as authenticated.

2.1.4 Implicit Flow

In the OpenID Connect Implicit Flow the RP redirects the user to the OP authentication endpoint, when the user expresses the wish to be logged in via the OP. The user then authenticates at the OP, OP and user negotiate user consent on the request, and the OP creates an ID token signed with a private key for which the OP has made the public key available to the RP. The OP redirects back to the RP redirection endpoint with the ID token in the fragment component. Now, the RP can retrieve the ID token from the user's browser and validate the ID token signature. If validation by the RP succeeds, the RP acknowledges the user as authenticated by sending a session cookie to the browser. Figure 2.1 gives a high level illustration of the protocol flow.

2.1.5 Dynamic Discovery and Registration

While RP and OP might be pre-configured to interact with each other, OpenID Connect also specifies extensions that allow an RP to dynamically discover information about an OP [17] and allow the RP to dynamically register at an OP [15] before the RP sends an authentication request.

For dynamic discovery the RP queries the path `/.well-known/openid-configuration` of the OP and retrieves information about the endpoint for authentication requests, registration requests, the OP issuer identifier, a URL for retrieving the OP public verification key, supported functionality and algorithms and several other optional metadata.

If an RP wishes to register dynamically to an OP, it sends an HTTPS POST request to the OP containing metadata on the RP such as supported endpoints for redirection, the supported protocol flows, supported algorithms and many more.

2.2 Self-Issued OpenID Providers

This section describes important concepts in the Self-Issued OpenID Provider specification that differ from traditional OpenID Connect.

2.2.1 Self-Issued Identities

A core concept for Self-Issued OP are globally unique identifiers that an entity can prove the possession of an associated private credential for – so called Self-Issued Identities. While in traditional OpenID Connect identities are grouped under the scope of an OP that governs them, similarly to email addresses being grouped under a domain, Self-Issued Identities in general have no such property made explicit. As a result, in contrast to general OpenID Connect identity providers where a key associated with the identity provider is used to sign ID tokens for all identities the identity provider governs, for Self-Issued OP signing keys are directly associated with each Self-Issued Identity.

Self-Issued Identities as used for Self-Issued OpenID Provider can be of two types, called subject syntax types in the Self-Issued OP specification, either JWK Thumbprint URIs or Decentralized Identifiers (DIDs) are used as identifiers.

JWK Thumbprint URIs. The first type of Self-Issued Identity, based on JWK Thumbprints, is in essence a public verification key in form of a JSON Web Key (JWK) from which a stable identifier is derived by applying a hash function on a normalized form of the public key. The resulting hash together with a prefix to mark the identifier as a JWK Thumbprint URI forms the identifier. When using a JWK Thumbprint URI identifier, the underlying public key is provided in the ID token together with the identifier.

For the verification of the ID token such a Self-Issued Identity, the RP computes the JWK Thumbprint of the provided key and verifies that it equals the user identifier presented in the ID token. The RP then uses the key to verify the signature of the ID token.

Uniqueness of the identity here is achieved by relying on cryptographic assumptions such as the collision resistance of the hash function, and the uniqueness of generated public keys. The exact format for such an identifier is described in [23].

Decentralized Identifiers. The second type of Self-Issued Identity are DIDs, as described in [19]. DIDs are identifiers that allow to prove control over a DID using public verification keys that are published in trusted storage. Each DID consists of a prefix, a method and a method-specific identifier.

Every DID is associated with a DID Document that contains metadata for the DID such as public verification keys that are used to prove control over the DID. For storing DID Documents and allowing resolution of a DID to the DID Document a trusted data storage is required, the so called Verifiable Data Registry (VDR). Similarly to the identities governed by a traditional OP, each VDR governs a method and is responsible to ensure global uniqueness of the DIDs within the method it governs.

The exact structure and assurances of a VDR might vary widely depending on the specific method. At the time of the writing of this thesis, the majority of DID method specification drafts use blockchain technology to instantiate the VDR, but also other distributed systems such as DNS may be used [20].

2.2.2 Verifiable Credentials

With the use of a Self-Issued OP and Self-Issued Identifiers any claims about the user in the ID token are self-asserted. While a traditional OP might establish trust with the RP as an institution, a Self-Issued OP is not expected to have this option. As a result a natural addition to Self-Issued Identities is to allow the Self-Issued OP to present additional claims about the user by a trusted third party. For this, claims are presented in the form of so-called Verifiable Credentials. The Verifiable Credential data model is defined in [18]. As an addition, [21] defines how Verifiable Credentials can be used with OpenID Connect and the Self-Issued OP extension.

A Verifiable Credential contains a collection of claims that an issuer attests about a subject. The issuer grants this Verifiable Credential to a holder that typically but not necessarily is the same entity as the subject. The holder can then create a Verifiable Presentation from the Verifiable Credential to present the claims of the issuer to a verifier. Integrity and authorship of the Verifiable Credential and Presentation can be cryptographically verified, e.g., through the use of signatures or means like zero-knowledge proofs.

For example, an RP might need to verify that a user is at least 18 years old for legal reasons. A user could send a claim to the RP that this is true, but a RP cannot trust this, as even users that do not meet this requirement could send such a claim. A trusted third party like the bank of the user that has verified that the user indeed fulfills the claim however, could issue a Verifiable Credential for the user asserting the claim. If the user now presents the Verifiable Credential to the RP, the RP can trust this claim as much as it trusts the issuer. As we expect issuers to be reputable institutions, this improves the level of trust the RP can place in the provided claims.

Holder Binding. Often, a vital aspect to enable trust into the claims in a Verifiable Credential is that the claims are bound to a particular subject or holder, such that a malicious verifier cannot use a Verifiable Credential presented to it at another party. Two major approaches to this are presented in [3].

One option is to include the identifier of an entity in the Verifiable Credential, and present the credential together with a proof of authentication for that identifier to bind the authenticated subject to the Verifiable Credential subject. For OpenID Connect that would mean that the subject of the ID token and the subject of the Verifiable Credential are equal. The other option is to use a so-called 'Link Secret', a secret provided by the holder to the issuer for the Verifiable Credential. When presenting the Verifiable Credential the holder then proves knowledge of the secret in the Verifiable Credential to the verifier. For OpenID Connect a possibility to realize this is for the Self-Issued OP as the holder to generate a fresh signing key pair and give the public key to the issuer. The issuer then includes the public key as the subject in the credential. When presenting the Verifiable Credential, the holder now uses the associated private key to sign the Verifiable Presentation.

2.2.3 Local Application

The Self-Issued OP application runs on a device of the user and is unique to the user, in contrast to a traditional OP that is offered as a web application and typically serves many users. As a result some behaviors of Self-Issued OP and traditional web service OP differ.

A Self-Issued OP application does not in general provide an endpoint to the network. Also, while in the case of traditional OpenID Connect RP and OP can negotiate the parameters of the interaction before the authentication flow, in the case of Self-Issued OP, where each user holds their own Self-Issued OP application some behavior of the protocol needs to be adapted.

Self-Issued OP Discovery. As the Self-Issued OP application on the user's device might not provide an endpoint to the network for discovery, we need an additional endpoint to provide metadata for the Self-Issued OP when dynamic discovery of the Self-Issued OP application is to be supported. A priori, the RP does not know how to invoke the Self-Issued OP and thus the Self-Issued OP application cannot provide this endpoint itself. As a result, we need an additional web service that makes the metadata available for the Self-Issued OP. In this thesis, the web service providing this endpoint for the Self-Issued OP metadata is called Configuration Provider.

If, for example, a company owning the domain `example.com` offers a Self-Issued OP application, they also might provide an endpoint on the `/.well-known/` directory that provides metadata for all users that use an instance of their Self-Issued OP application.

Aside from the function of this web service to provide metadata for local Self-Issued OP applications, the Configuration Provider might also be the result of the redirect to the Self-Issued OP if no Self-Issued OP application is installed on the device of the browser.

Relying Party Registration. With each user holding their own instance of a Self-Issued OP, a pre-registration of the RP at each Self-Issued OP becomes infeasible. As a result the RP sends the registration metadata with each request to the Self-Issued OP, either by value in a parameter, or by reference as a URL from which the Self-Issued OP can retrieve the RP registration metadata.

A major difference here is that the identifier of the RP, the `client_id` is not assigned to the RP by the OP but provided by the RP in the request. If the RP is part of a trust framework that allows to resolve the RP public key, the RP might choose its identifier in the trust framework as its `client_id` and sign the request with the private key for the public key made available. Otherwise, the `client_id` must be equal to the redirection endpoint given in the request – the redirection endpoint then also serves as the identifier of the RP towards the Self-Issued OP.

Invocation of the Self-Issued OP Application. For the Self-Issued OP as a local application, simply using HTTP redirection does not suffice to invoke the Self-Issued OP. In this case other options need to be used. There are two ways to address an application from the browser context considered in the Self-Issued OP specification.

One way are custom URL schemes that many systems support. Any application can register to handle such a custom URL scheme and the browser then passes the requests to this scheme to the application. The Self-Issued OP specification security considerations warn, however, that there is no restriction on which app can register a custom scheme, and thus a malicious application on the user device can register the same scheme as a good one and intercept messages intended for the good one.

To overcome the shortcomings of custom URL schemes a concept of registering the handling of an HTTPS URL securely were developed – these concepts hold names like Android App Links [1] or Universal Links [22]. For these, the owner of the domain makes available metadata on the authorized applications in the `/.well – known/` directory for the HTTPS URL. These concepts allow verifying the caller if the application is authorized to handle the HTTPS URL or prevent registration of unauthorized applications as handlers of HTTPS URLs. However, the lack of standards for these concepts poses a challenge for thorough analysis.

User Interaction. While in the traditional OpenID Connect where both RP and OP are web applications the user interacts with both only in the browser context, for native Self-Issued OP applications the user can directly interact with the Self-Issued OP. In particular, while in traditional OpenID Connect the user authenticates to the OP and gives consent to answer the request via the browser, in the case of Self-Issued OP the user interacts with the Self-Issued OP application directly.

2.3 Same-Device Flow

The Self-Issued OP same-device protocol flow is characterized by the browser and the Self-Issued OP application of the user residing on the same user device. It is an extension of the Implicit Flow of OpenID connect. The Relying Party requests authentication from the local Self-Issued OP application, that replies with an ID token containing the information the RP needs to authenticate the user. The RP and the Self-Issued OP application communicate via the browser of the user. Figure 2.2 illustrates the protocol flow.

In more detail the same-device protocol flow proceeds as follows. The browser starts the flow with the RP by sending a POST request including the Self-Issued OP to use (Message 1) — typically the RP would let the browser display a website where the user chooses their desired Self-Issued OP

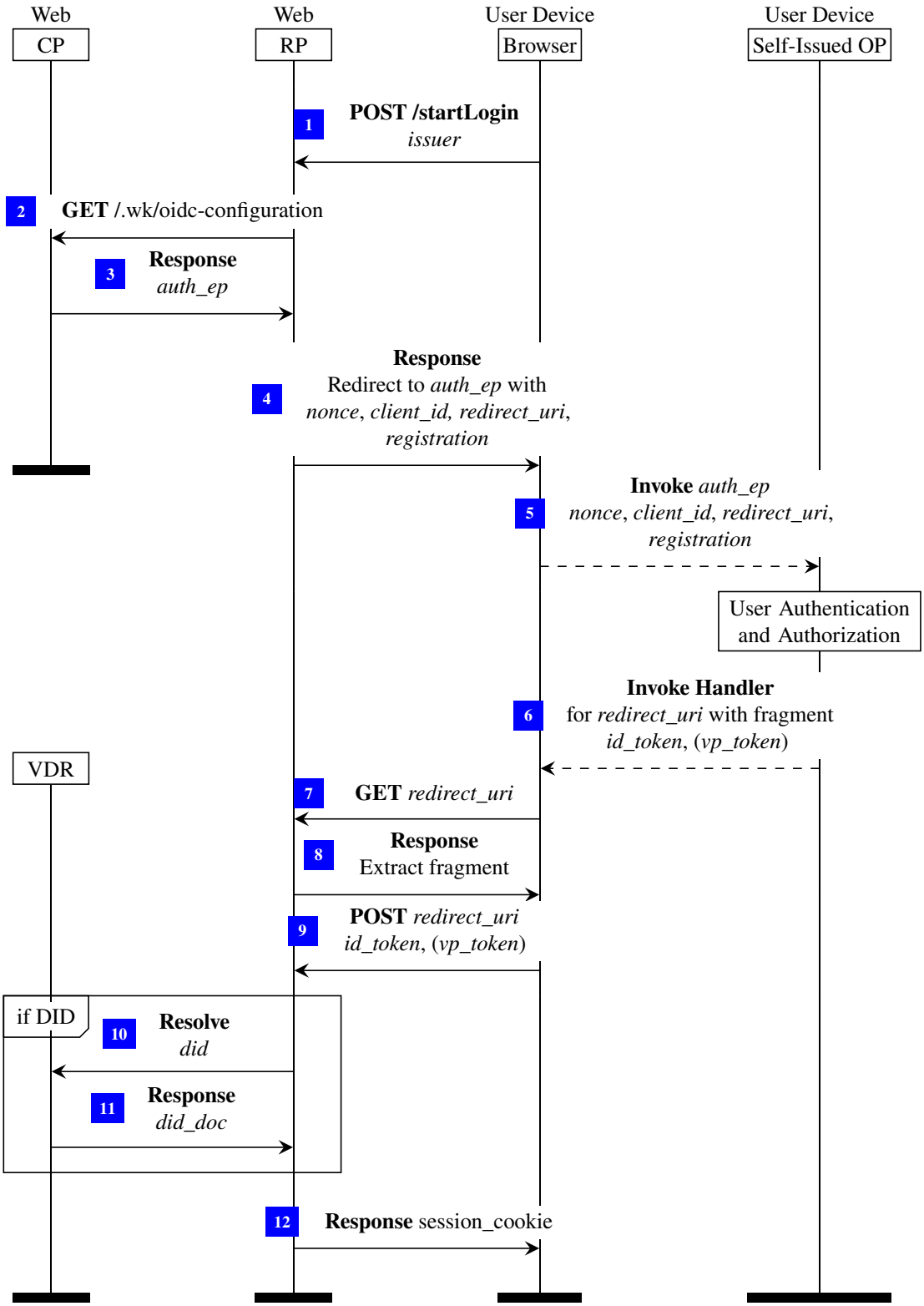

Figure 2.2: Self-Issued OpenID Provider Same-Device Flow

and the user would select one to start the protocol flow. For the chosen Self-Issued OP the RP then sends a GET request for the discovery metadata to the Configuration Provider for the domain of the Self-Issued OP (Message 2) if the RP has not already cached it.

After the discovery of the Self-Issued OP metadata, the RP responds to the browser with a redirect to the authorization endpoint of the local Self-Issued OP application including the request parameters (Message 4). If a Self-Issued OP application is registered for the authentication endpoint in the environment of the browser, the browser invokes this Self-Issued OP application with the request parameters (Message 5). Otherwise, the browser performs a standard HTTP redirect to the website for the domain which results in a request to the Configuration Provider.

When the Self-Issued OP application is invoked, it requests user authentication and consent for processing the request, and asks the user to choose a Self-Issued Identity. The RP might pass the request and the registration metadata by value or by reference. If the RP passed the request and the registration metadata by reference, i.e., as a URL, the Self-Issued OP first retrieves the request or registration values using a GET request. If the RP requested Verifiable Credentials, the Self-Issued OP requests consent from the user to present such a credential and lets the user select a suitable credential for presentation that the Self-Issued OP holds. The Self-Issued OP then constructs an ID token (and possible Verifiable Presentations) for the choice of the user and invokes the local browser with the redirection URL the RP provided in the request and the ID token (and possible Verifiable Presentations) added in the fragment (Message 6).

The browser then sends a GET request to the redirection endpoint of the RP. Note that this GET request does not directly contain the fragment and thus does not contain the ID token (and possible Verifiable Presentations). The RP responds to this request with a script that retrieves the fragment from the browser (Messages 6 to 9).

After retrieving the authentication response, the RP validates the signature and the content of the ID token, as well as the Verifiable Presentations if the RP requested them. If the user chose a DID type Self-Issued Identity then the RP needs to resolve this DID to the associated public key for verifying the signature. For this the RP retrieves the DID Document from the VDR (Messages 10 and 11).

If the RP finds the ID token and the requested Verifiable Presentations to be valid, it responds to the browser with a session cookie for an authenticated session (Message 12).

2.4 Cross-Device Flow

As an alternative to the same-device flow the Self-Issued OP specification defines the so-called cross-device flow. This flow differs from the same-device flow in that the Self-Issued OP application and the browser of the user reside on different devices and thus the RP and the Self-Issued OP cannot utilize redirects through the browser to communicate. We call the first device, where the browser resides, the session device and the device where the Self-Issued OP resides the authentication device.

The cross-device flow proceeds as follows. When the user initiates a login flow, the RP renders the request as a QR code in the browser. The user then scans this QR code with the authentication device, either with the Self-Issued OP application directly, or such that the Self-Issued OP is invoked with the request. Alternatively, the RP might transmit the request to the Self-Issued OP by a similar

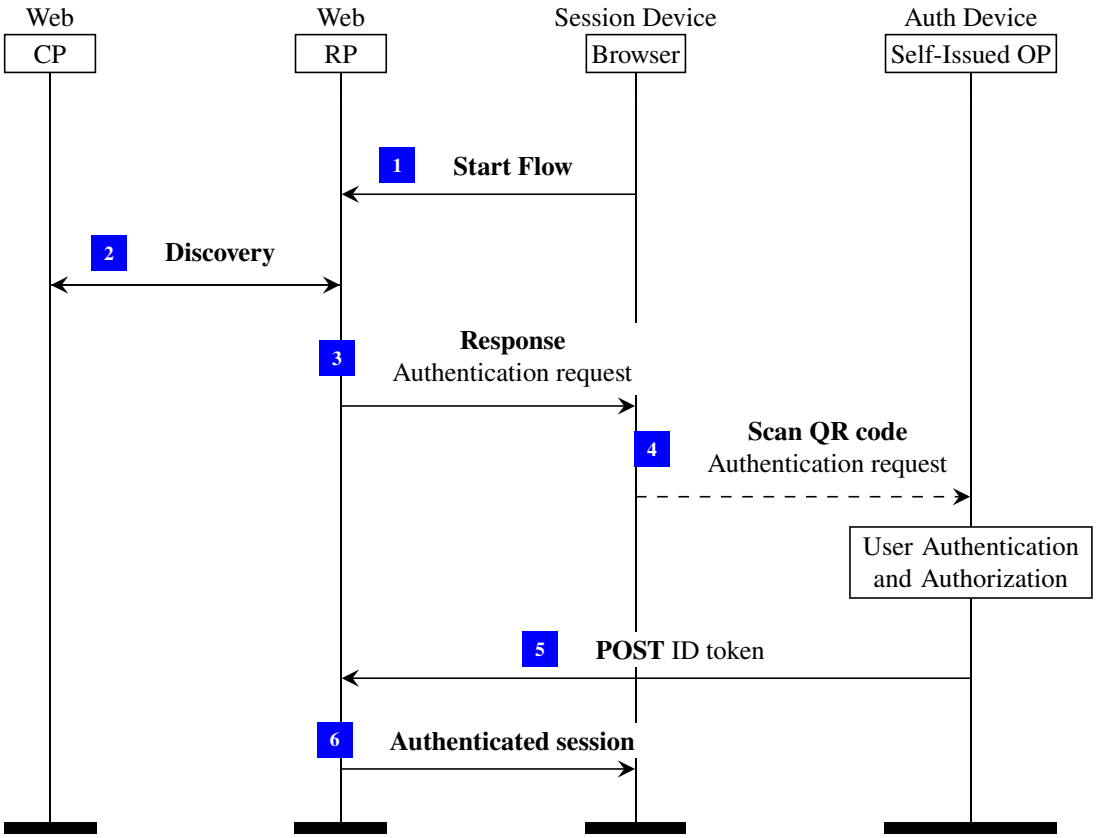

Figure 2.3: Self-Issued OpenID Provider Cross-Device Flow

mechanism. After processing the request, the Self-Issued OP directly sends a POST message which contains the ID token (and possibly Verifiable Presentations) to an endpoint of the RP. A high-level illustration of the cross-device flow is given in Figure 2.3 (note that the optional communication to resolve the DID Document is omitted for simplicity).

3 Cross-Device Flow Attack and Mitigation

This section describes the known request replay attack on the cross-device flow and the mitigation of this attack that we developed for this thesis.

3.1 Request Replay Attack

As the security considerations in section 14.3 of the Self-Issued OP specification caution, there is a known replay attack on the cross-device flow that breaks authentication. Figure 3.1 gives an illustration of the attack.

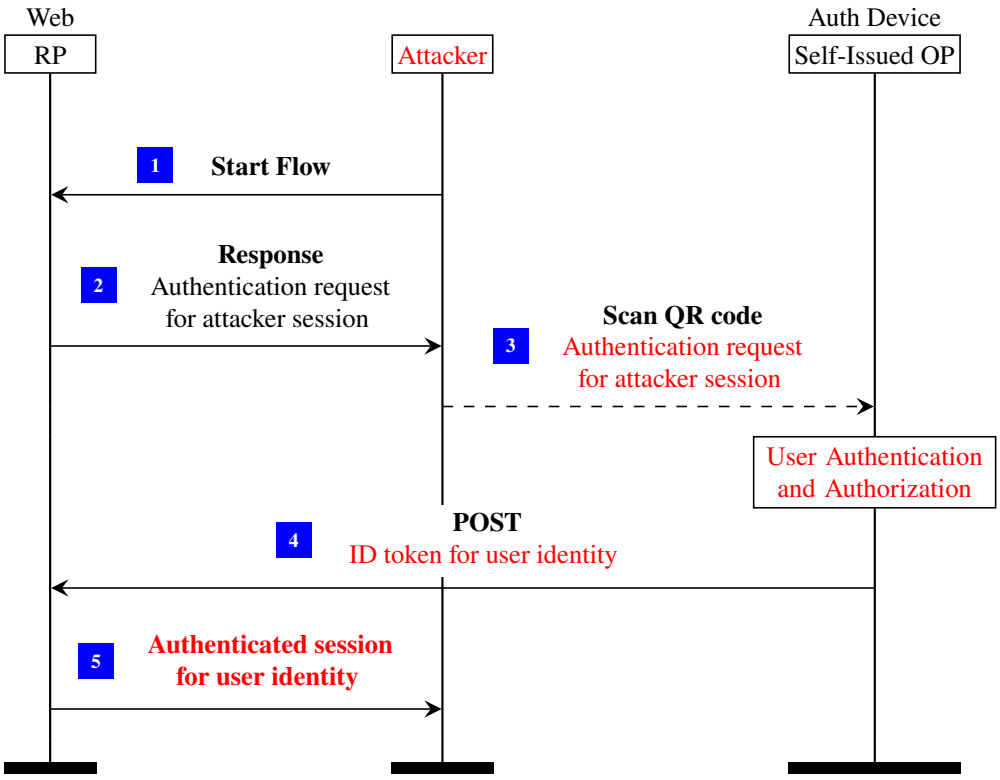

Figure 3.1: Replay Attack on the Cross-Device Flow

The attacker starts the attack by initiating a login flow at the RP to obtain a QR code that contains an authentication request from the RP. Then, the attacker presents this authentication request to the user, for example, when the user wishes to log in at the attacker's website on the user session device. To the Self-Issued OP on the user's authentication device, this authentication request looks like a legitimate request for the honest RP. If the user gives consent at the Self-Issued OP to provide an ID token with a user identity, the Self-Issued OP sends this ID token to the RP. When the RP receives the ID token, the RP associates this ID token with the context for which the RP created the authentication request, i.e., the attacker session, and thus acknowledges the attacker as logged in under the user identity.

3.2 Mitigation Overview

We see that the attack on the cross-device flow results from the attacker presenting an authentication request from an honest RP associated with a session of the attacker to the Self-Issued OP of the user on the authentication device. This Self-Issued OP on the authentication device cannot distinguish between requests from a session of the user and a request from a session of the attacker. If the Self-Issued OP of the user sends an ID token in response to a request for a session of the attacker, the attacker is logged in under an identity of the user at the RP.

The core idea of this variant of the cross-device flow is to have a part of the Self-Issued OP as a web service on the session device, we call this part Cross-Device Stub, and perform some checks on requests to ensure their legitimacy, i.e., that the request for an RP is associated with a session in the user browser with the RP), before the Cross-Device Stub renders the QR code with the request on behalf of the RP. The Self-Issued OP on the authentication device then is required to only accept requests in QR codes from the Cross-Device Stub. A high-level illustration of the proposed attack mitigation is given in Figure 3.2.

The variant of the Cross-Device Flow described in this section can be broken down into two parts. The first is the check that the Cross-Device Stub needs to perform to ensure that a request is legitimate. For this, we describe a protocol flow between Cross-Device Stub, browser, and RP in Section 3.3. The second part is the pairing of the Self-Issued OP and a trusted Cross-Device Stub to ensure that the Self-Issued OP only accepts requests from the Cross-Device Stub and thus only processes the requests for that the checks in the previous parts have succeeded. We describe how we achieve this in Section 3.4.

3.3 Sync Flow

The sync flow serves for the Cross-Device Stub to assert that authentication requests for an RP are associated with a session in the same browser (on the session device) as the one that the Cross-Device Stub is communicating with. Figure 3.3 gives an illustration of the sync flow.

To start the sync flow the RP redirects to the Cross-Device Stub sync start endpoint including the authentication request that the RP would render in the QR code in the standard cross-device flow as a parameter (Messages 2 and 3).

On receiving a request to the sync start endpoint, the Cross-Device Stub stores the request and redirects back to the RP. For this, the Cross-Device Stub extracts the redirection endpoint and the nonce of the request from the parameters. The Cross-Device Stub then adds to the redirection endpoint the request nonce in the parameter `cnonce`, a fresh nonce in the parameter `cstub` that serves to detect CSRF attacks later, a URL for the Cross-Device Stub sync fin endpoint in the parameter `stub_fin`, and a parameter `xdev_sync` set to `request` to make explicit that it is a request of the sync flow. With this modified redirection endpoint URL in the Location header, the Cross-Device Stub redirects to the RP (Messages 4 and 5). Note that in this response the Cross-Device Stub also instructs the browser to set a cookie to later correlate the response of the sync flow.

When the RP receives a request to the redirection endpoint with the parameter `xdev_sync` that marks it as a sync request, the RP retrieves the session from the cookie in the request — if there is no such cookie the RP aborts the flow. The RP then verifies that there is a request with the nonce parameter given in the `cnonce` parameter of the sync request in this session. If this holds, the

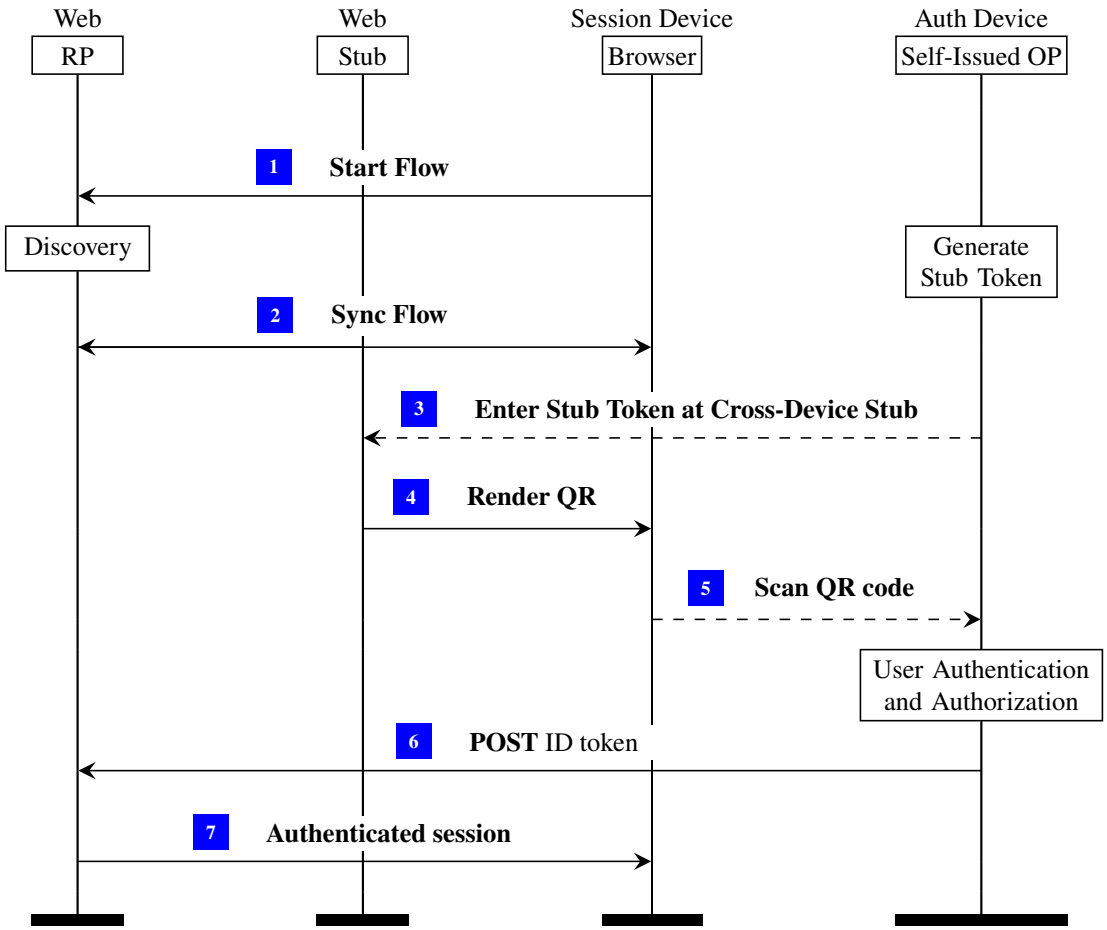

Figure 3.2: Cross-Device Flow Attack Mitigation – Overview

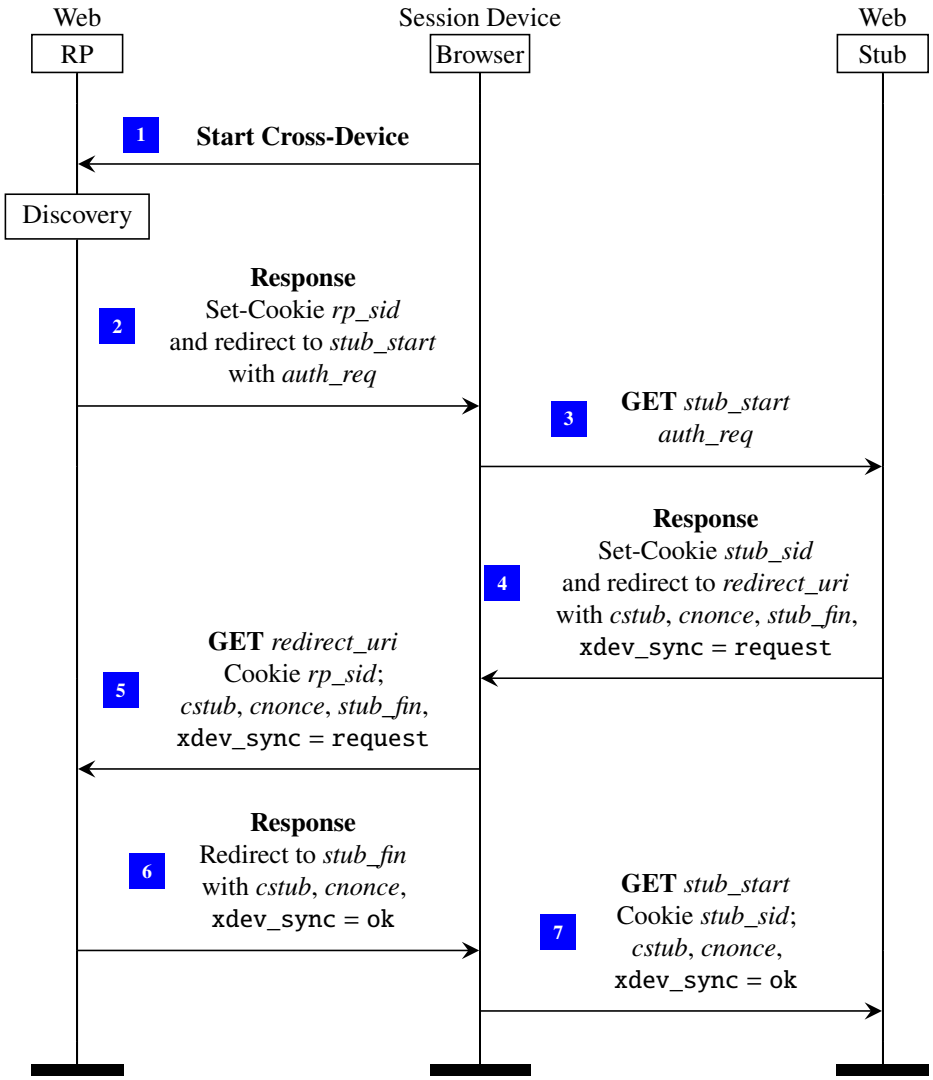

Figure 3.3: Cross-Device Flow Attack Mitigation – Sync Flow

RP retrieves the sync fin endpoint from the parameter `stub_fin` in the sync request and adds the parameters `cstub` and `cnonce` from the sync request, as well as a parameter `xdev_sync` set to `ok`. The RP then redirects to this modified sync fin endpoint (Messages 6 and 7).

On a request to the sync fin endpoint, the Cross-Device Stub retrieves the session from the cookie in the header. The Cross-Device Stub then verifies that the parameters `cstub` and `cnonce` are equal to the ones stored in this session. If these checks succeed the Cross-Device Stub is convinced that the authentication request is for a session of the browser with the RP that is associated with the redirection endpoint and ready to render the authentication request in a QR code.

Note that before the sync flow the RP needs to learn the sync start endpoint of the Cross-Device Stub. As we consider the Cross-Device Stub to be closely associated with the Self-Issued OP, we suggest that the metadata for the Cross-Device Stub is given together with the metadata of the Self-Issued OP. The Configuration provider can give the Cross-Device Stub sync start endpoint as another parameter of the Self-Issued OP discovery metadata when using dynamic discovery.

3.4 Cross-Device Stub and Self-Issued OP Pairing

For our mitigation of the attack on the cross-device flow, we need that the Self-Issued OP of the user on the authentication device only processes authentication requests that are associated with a session between the user browser on the session device and the Cross-Device Stub. To achieve this we need to establish a binding between the Cross-Device Stub session and the Self-Issued OP of the user.

We establish this binding by sharing a secret between the Self-Issued OP application of the user and the Cross-Device Stub (associated with the session in the user browser). For this, the Self-Issued OP generates a nonce or some comparable secret and displays it to the user. We call this secret a stub token. The Cross-Device Stub then prompts the user to enter the stub token before rendering the QR code and includes the stub token in the QR code. When processing an authentication request from a QR code, the Self-Issued OP must check if the stub token is valid.

The mitigation of the attack relies on the attacker being unable to obtain this stub token. For our mitigation, we assume that the user will only provide the stub token of the Self-Issued OP to an honest Cross-Device Stub in an honest browser, and also that the user does not scan the QR code with the stub token with a malicious application or leak it to another party by other means.

Note that the stub token might be single-use and generated freshly for each request, or be stored in the session or as a cookie in the browser by the Cross-Device Stub and reused. An attacker must not be able to guess this stub token, and thus the stub token must be a secret value chosen to fulfill this requirement, e.g., a random string of sufficient length.

Note that the Cross-Device Stub web service simply signing the QR code does not suffice to achieve the binding of Cross-Device Stub and the Self-Issued OP of the user we need. A signature of the Cross-Device Stub does not bind the QR code to a session in the user browser unless the used key pair is associated with this session. In the latter case, however, a key exchange between Self-Issued OP on the authentication device and Cross-Device Stub in the session context of the browser on the session device is necessary.

4 Local Response Leak Attack

While the version of the Self-Issued OP specification researched in this thesis urges caution when invoking the Self-Issued OP application, it gives little guidance on how the Self-Issued OP application should invoke the browser for the response. The attack described in this section shows that for this response implementers must proceed with caution as well.

This section describes a scenario where an attacker can obtain an authentication response, including an ID token and possible Verifiable Credentials of the user that are valid at an honest RP. As a result, the attacker could impersonate the user at the RP, inject an identity of the attacker in the user session, or associate Verifiable Credentials of the user with an attacker identity.

A core reason for this attack to be possible is the change in the communication structure as compared to the traditional OpenID Connect Implicit Flow using redirects. In contrast to the OpenID Connect Implicit Flow the browser invokes the Self-Issued OP application. This application might, in general, not be able to identify the calling application. For the authentication response to the browser the Self-Issued OP invokes operating system functionality to invoke a handler for the RP redirection URL. Depending on the operating system, however, a malicious application might offer to handle this redirection URL. As a result, an attacker might be able to bypass the use of a legitimate browser and confuse the user about the session they are authenticating.

4.1 Assumptions

For this attack, we assume that the attacker has an application installed on the user device where the Self-Issued OP resides that is registered to handle the RP redirection endpoint.[1] We also assume that the attacker application can invoke the Self-Issued OP application.

We assume that the user will give consent at the Self-Issued OP application to answer requests if they look legitimate.

4.2 Attack Scenario

The attack scenario is sketched in Figure 4.1. Note that we omit the discovery step and the communication between RP and VDR for the case of a DID.

The attacker starts a login flow at the RP to obtain an authentication request (Messages 1 and 2).

[1] An example of this would be that the attacker application registers the RP redirection endpoint as an Android Deep Link [2] and the Self-Issued OP application does not verify this link.

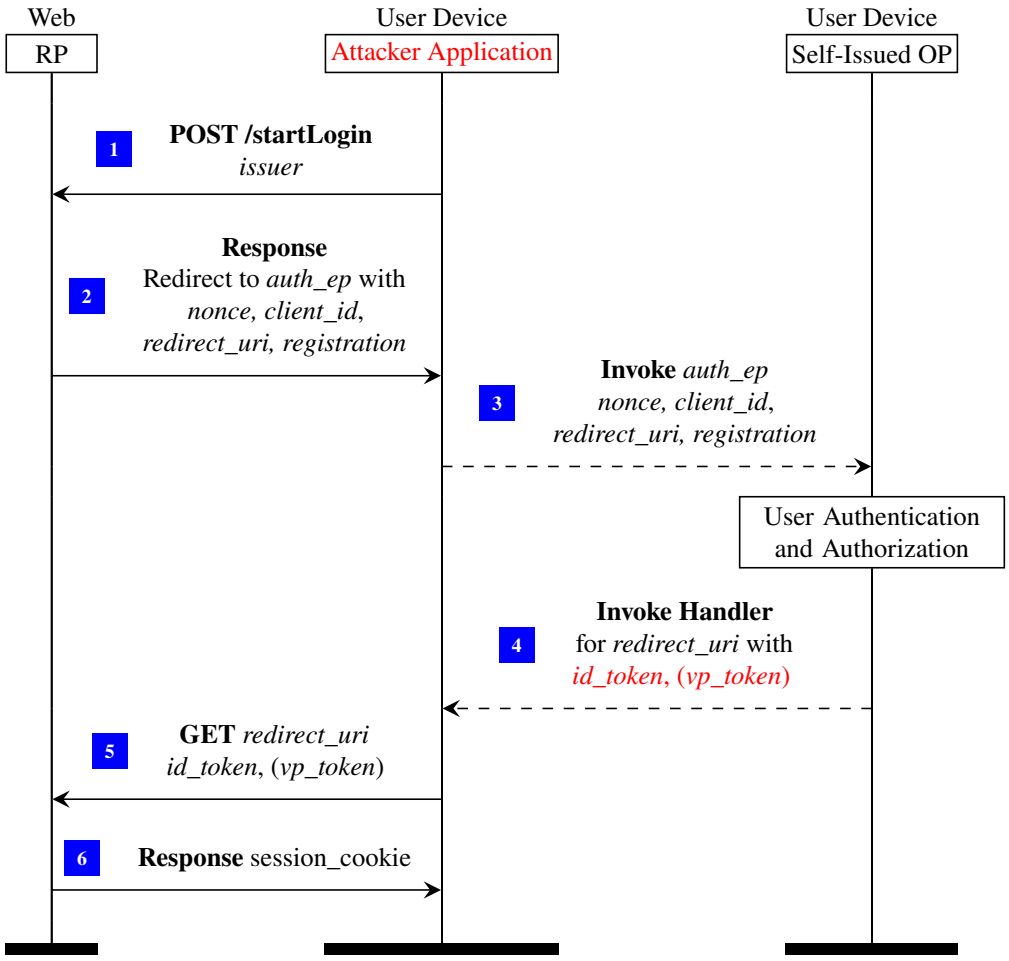

Figure 4.1: Local Response Leak Attack

Then, the attacker uses the malicious application on the user's device to invoke the Self-Issued OP application of the user with the authentication request created for the attacker (Message 3).

The Self-Issued OP then prompts the user to authorize an authentication request for the RP. To the user this request looks like a legitimate request to authenticate at the RP and thus we assume that the user gives consent to authenticate with one of the user identities. The Self-Issued OP application then creates an ID token and invokes the operating system with the redirection URL containing the ID token.

The operating system then either displays a list of possible handlers including the malicious application registered as a handler or chooses a handler non-deterministically. In both cases, it is possible that the attacker's application obtains the authentication response and with it the ID token (Message 4).

The attacker can then present this ID token to the RP and obtain a session cookie associated with the identity of the user (Messages 5 and 6) breaking authentication.

In an alternate version of the attack, the user starts a login flow at the RP with their legitimate browser and gives consent to authenticate in the Self-Issued OP. When choosing a handler for the redirection endpoint, however, the response is passed to the malicious application. This allows the attacker to retrieve the nonce parameter from the ID token and create an ID token for an identity of the attacker. The attacker can then invoke the user's browser with the attacker ID token and the RP logs the user in under the identity of the attacker breaking session integrity.

4.3 Proposed Mitigation

To prevent this attack, the Self-Issued OP needs to ensure that it passes the response only to legitimate browsers on the user device. The exact means for this depend on the operating system and the options it offers.

A possible mitigation approach that is widely available is for the Self-Issued OP to explicitly invoke the default browser of the system, assuming that this browser is honest. This approach however can cause problems if the user uses more than one browser.

In general, implementers of Self-Issued OP applications need to utilize the options available to them in the operating system environment and ensure that the authentication response is not leaked. For our model, we restrict the Self-Issued OP application to only answer to one browser we consider to be on the same user device.

We disclosed the problem to the OpenID Working Group and provided a Pull Request [14] to contribute a security consideration regarding this problem to the specification on the 29th of January 2022. The Pull Request was merged on the 3rd of February and will be part of future versions of the specification.

5 The Web Infrastructure Model

The Web Infrastructure Model is a Dolev-Yao style model of the web. It was originally proposed by Fett et al. in [7] under the name FKS model after the authors. Note that this thesis uses a more recent version of the WIM given in [10]. This section gives an informal summary of the WIM. The structure of the WIM summary given in this section follows the summary from [6].

5.1 General Concepts

The WIM models the web as a system of processes that act as browsers, web servers and attackers. These processes communicate by creating events that consist of a message, a sender and a recipient. For this each process is associated with a non-empty set of IP addresses that it listens to.

In each step of a run of the system a pending event is chosen non-deterministically, and a process that listens to the receiver address is invoked with the event. On the input of an event, the process can change its state and emit a set of events to the collection of pending events, from which then a new event is picked non-deterministically that again is processed.

Terms. Events and process states of the model are expressed as formal terms over a signature Σ. This signature consists of constants such as addresses and strings, function symbols such as $\mathsf{pub}(\cdot)$ for deriving a public key from a private key, sequences and projection symbols. On this signature Σ together with an infinite set of nonces $\mathcal{N}$, the WIM defines terms.

For example, an HTTPS GET request for the URL `https://example.com/p?key=v` that contains a cookie in the headers with name `sid` and the nonce v_2 as the value and an empty body is modeled as

$$\mathsf{enc_a}(\langle\langle \mathtt{HTTPReq}, v_1, \mathtt{GET}, \mathtt{example.com}, /\mathtt{p}, \langle\langle \mathtt{key}, \mathtt{v}\rangle\rangle, [\mathtt{Cookie} : \langle \mathtt{sid}, v_2\rangle], \langle\rangle\rangle, k'\rangle, \mathsf{pub}(k))$$

where k is the private TLS key associated with the domain `example.com`, v_1 is a nonce that serves as an identifier of the request and k' is a nonce that serves as a symmertic key to encrypt the response.

With the signature Σ we associate an equational theory that induces a congruence relation $\equiv$ that reflects the semantics of the function symbols included in Σ. For example, we get that $\mathsf{extractmsg}(\mathsf{sig}(x, k,)) \equiv x$ for arbitrary terms x and k — we model digital signatures to always allow extracting the message that was signed.

Processes. An atomic Dolev-Yao (DY) process consists of a set of addresses, a set of possible states of the process, a relation modeling the state transitions and emitted events when the process receives an event in a state and an initial state. For atomic DY processes, we require that the relation

is limited to transition where the output state and events can be derived from the previous state and the input event — this derivability is formally defined to match the semantic of the symbols in Σ. For example, we have for the set $\{\mathsf{enc_a}(m, \mathsf{pub}(k)), \mathsf{sig}(x, k')\}$ with m and x terms and k, k' nonces that x can be derived, but m cannot be derived (as the nonce k that serves as the private key is not in the set).

Attacker and Corruption. The attacker in the WIM is modeled as an atomic DY process that records all messages it receives in its state and may output any finite sequence of events that it can derive from its state and the input event. The attacker can also send a special corruption message that allows it to corrupt any process that receives this message. The process receiving such a corruption message then behaves like an attacker process. Processes that are not corrupted are called honest.

Web Systems. In the WIM, a system is defined to be a set of atomic DY processes. A configuration of a system consists of the states of all processes of the system, the pending events and an infinite sequence of unused nonces. A run of a system is a sequence of configurations where each configuration is obtained by invoking one process with an event of the pending events of the previous configuration that then performs one computation step. We call a transition from a configuration to the next a processing step.

A web system models the web infrastructure and web applications. It contains a system of (honest) processes such as web servers and browsers and attacker processes, scripts and an infinite sequence of trigger events (special events that model certain actions from the 'outside' or serve as dummy messages for state changes without a message such as the user pressing a button to reload a website, or entering a URL in the browser). Note that web systems in the WIM abstract certain points of the environment. The Public Key Infrastructure, for example, is simply modeled as a mapping of domains to public keys in the initial state.

5.2 Browser Model

The WIM provides a browser model that closely follows web standards. The state of the browser process includes among others a set of windows (as described below) that model browser tabs or top-level windows, cookies and web storage.

The JavaScript running in the browser is modeled as scripts that are defined similarly to DY processes. When the browser receives a trigger message, it provides the script with state information as input and applies its script relation to output a new state and a command. The browser then interprets the command. Examples for commands are the instruction to follow a link or the instruction to post a form. The attacker script is a special script that behaves similarly to the attacker process. It outputs everything it can derive from its inputs.

A window in the browser contains a set of documents, one of which is active. These document represent loaded web pages and include the history of the window. Each document contains among others a location URL, a referrer, a script and its state, and a set of subwindows that represent the iframes in the document. Each window therefore contains a tree of sub-windows.

The full formal definition of the browser model we use in the formal analysis in this thesis can be found in [10].

6 Overview of the Models of Self-Issued OpenID Providers

In this section, we describe the extension of WIM for the analysis of the Self-Issued OpenID Provider specification presented in this thesis. Note that we model the same-device Self-Issued OP specification and our variant of Cross-Device Self-Issued OP in separate models to limit the complexity of the analysis. This section first gives an overview of the extension of the WIM browser (Section 6.1), followed by the overview of the same-device Self-Issued OP model (Section 6.2) and the model of our cross-device attack mitigation (Section 6.3). We close this section with a brief overview of some limitations of our model in Section 6.4.

The formal model definitions for the same-device Self-Issued OP model can be found in Appendix B, the definition for the cross-device model can be found in Appendix C.

6.1 Browser Extensions

The WIM browser as given in [10] provides most of the browser functionality we need for our analysis. However, for the communication of the browser with a local (Self-Issued OP) application to model the concept of App Links or Universal Links required for Same-Device Self-Issued OP, we need an extension as described in Section 6.1.1. Similarly, we need browser support to model the rendering and scanning of a QR code for the cross-device Self-Issued OP model. We give an overview of this extension in Section 6.1.2.

6.1.1 Browser Extension for Same-Device Self-Issued OPs

For modeling Self-Issued OP applications local to a user device, we extend the WIM browser with functionality to support secure 'local' browser-to-application and application-to-browser communication as an abstraction of the concept of App Links or Universal Links.

For the extension to allow browser-to-application communication, we include an address and a public encryption key in the state of the browser for some URLs. With this, we model an application that is registered at the operating system to handle a URL. As we assume the registration of the URL for an application to be verifiable (like with App Links or Universal Links), these links are set to the correct processes in the initial state. If the browser then sends a request for a URL for which an application is registered, instead of resolving the URL via DNS, it directly sends the message to the process, encrypted with this key for local communication. Note that the encryption of this message models the assumption that the operating system is honest (at least if the browser is honest) and that a local call cannot be intercepted (e.g., by a malicious application registering the same URL).

We also need confidential communication from the Self-Issued OP application to the browser (see Chapter 4), where the Self-Issued OP application invokes the browser to open the RP website. To achieve this, we use the redirection functionality of the browser and allow the Self-Issued OP application to encrypt the URL in the location header towards the browser. With this, we model the case where the Self-Issued OP calls the default browser of the operating system, and the data transmitted with the call is not leaked, or the Self-Issued OP makes a secure call to a local browser by other means where an attacker cannot intercept the call.

6.1.2 Browser Extension for Cross-Device Self-Issued OPs

To model the rendering of QR codes which we need for the cross-device flow, we extend the WIM browser model to support an additional script command.

We define a script command of the shape $\langle \texttt{ShowQR}, \mathit{content} \rangle$ that models rendering a QR code with the content term *content*. When a script outputs this command, the browser picks a recipient address non-deterministically from a list of QR code recipients and sends the QR code message to this recipient. We set this list of recipients in the initial state. As we assume that QR codes remain secret, we limit the recipients to a single Self-Issued OP per browser (the Self-Issued OP of the same user) and also encrypt QR messages with a public key for the recipient.

6.2 Model of Same-Device Self-Issued OpenID Providers

This section gives informal descriptions of the processes that form the model of Self-Issued OpenID Providers for the same-device flow as web systems with a network attacker and an overview of the aspects of the Self-Issued OpenID Provider specification that are modeled. A Self-Issued OpenID Provider web system consists of Browser, RP, Self-Issued OP, Configuration Provider, and VDR processes and a network attacker process.

6.2.1 Processes

This section contains an informal description of the processes that are part of the web system analyzed in this thesis.

Attacker. In our model of the Self-Issued OP specification, we consider a network attacker as specified in the WIM specification. A network attacker can listen to any IP address and spoof outgoing addresses. We slightly extend this attacker model to allow the attacker to make calls to 'local' processes like the Self-Issued OP application.

As we intend to achieve security in the presence of a network attacker, we use cookies with the `__Host` prefix for sessions at the RP — otherwise a network attacker could inject cookie values and with them sessions via unencrypted communication.

Relying Party. The RP is modeled as a generic HTTPS server as given in [10]. It models an OAuth 2.0 client that supports user authentication using a Self-Issued OP similar to the OpenID Connect RP in [6].

The RP provides an index script, an endpoint for starting the login flow, endpoints for retrieving registration information and requests that were transmitted by reference, and an endpoint for the redirected authentication response. When the RP receives a request for the endpoint for the start of the login flow, the RP prepares the parameters of the authentication request and redirects to the Self-Issued OP with the request. When the RP receives a request to the response endpoint, the RP validates the authentication response in the request and on success responds with a cookie for an authenticated session. If the identifier in an ID token presented to the RP is a DID, the RP sends a request to the VDR to resolve the DID to its DID document that contains the associated public key(s) during validation.

Note that in our model the RP only supports Self-Issued OP and no other OpenID Connect or OAuth 2.0 flows. As a result, parameters to distinguish the Self-Issued OP protocol flow from other OAuth 2.0 flows are not included in the model for readability.

Configuration Provider. The Configuration Provider model is a generic HTTPS server that serves to make metadata about Self-Issued OP applications for an issuer identifier domain. For this, the Configuration Provider offers a single endpoint for HTTP GET requests for the path `/.well-known/openid-configuration`.

Self-Issued OpenID Provider. The Self-Issued OP application is modeled as a generic HTTPS server, with some modifications to represent the behavior as an application local to a user device. Each Self-Issued OP shares the domains of a Configuration Provider that is responsible for providing the Self-Issued OP metadata for the application. To model the local notion of Self-Issued OP applications each Self-Issued OP is associated with a browser modeling the local environment.

As the domains of a Self-Issued OP are not unique to an instance, each Self-Issued OP has a unique key pair for local calls instead of a TLS key pair. The public key of this key pair is known to the local browser and to the attacker who we assume can make calls to the Self-Issued OP application. The Self-Issued OP provides a single endpoint for processing an authentication request.

As it is vital that the response of the Self-Issued OP does not leak to a malicious party (see Chapter 4), the Self-Issued OP encrypts the redirection URI that contains the ID token towards the local browser to ensure confidentiality of communication. We model with this the case where the Self-Issued OP uses system-specific means to call the default browser or to verify the legitimacy of the browser it calls by other means. Note that we associate each Self-Issued OP with exactly one browser for simplicity. The reason for this is that if the Self-Issued OP may be convinced to pass its response to a single malicious browser the attack from Chapter 4 applies.

Verifiable Data Registry. We model each Verifiable Data Registry as a generic HTTPS server that allows to resolve DIDs of one DID method to their respective DID document and thus to their public keys. For this, the VDR model provides a single endpoint for HTTP GET requests. While in practical implementations the VDR is typically a distributed system, for the scope of this thesis, we simplify the VDR to a single server serving HTTPS requests.

6.2.2 Concepts

In this section, we give an overview of Self-Issued OP concepts included in the model and discuss modeling decisions associated with them.

RP Registration. The Self-Issued OP model supports providing RP registration metadata by value in the request and by reference. For this, the RP provides an endpoint for the Self-Issued OP to retrieve registration metadata using GET requests.

Note that the Self-Issued OP specification explicitly forbids the RP to specify `redirect_uris`. If the RP signs request the `client_id` must be resolved to a public key and the `jwks` value in the registration is not used. As other parameters for registration in the specification serve to distinguish other protocol flows from the Self-Issued OP protocol, specify supported formats and algorithms, and provide information to display to the user to support the consent decision, we limit the registration parameters to the supported self-issued identity types.

Self-Issued OP Discovery. In our model, the user provides an issuer identifier URL in the browser. The RP then retrieves the metadata for the Self-Issued OP at the Configuration Provider for the issuer identifier. Note that the RP can not retrieve the metadata directly from the Self-Issued OP as the RP a priori does not know how to address the Self-Issued OP. Our model does not support static Self-Issued OP metadata discovery, as dynamic discovery potentially creates a greater attack surface.

Signed Request Objects. Our model supports the two major kinds of requests, namely unsigned requests and requests in signed request objects. While for the unsigned case the *client_id* needs to be equal to the *redirect_uri*, signed requests do not have this restriction. Our model includes signed request objects by value in the `request` parameter of the request, and signed request objects passed by reference using the `request_uri` parameter.

Note that our model assumes that the underlying Public Key Infrastructure, that is also called trust framework, works as intended, and is static. While the Public Key Infrastructure is expected to be a whole system of its own in practical deployments, the mapping of domains to RP public keys for request signing is set in the initial state, analogously to the modeling of the Public Key Infrastructure for TLS encryption in the WIM.

Decentralized Identifiers. Our model includes a simplified version of DID Documents that can be retrieved from the VDR. We abstract from specific formats information and advanced concepts such as verification relationships. We also assume that the DID Documents that are governed by an honest VDR are static and there is at most one document per DID to ensure that the resolution of the DID Document is deterministic as the WIM does not model time.

Verifiable Credentials. In our model of Self-Issued OP, we include basic support for Verifiable Credentials — the RP might request a Verifiable Credential from the Self-Issued OP. We only send a single Verifiable Credential in a Verifiable Presentation per protocol flow, as for a list of Verifiable Credentials sent in the response, the RP would only repeat the same checks for each Verifiable

Credential and we expect no new behavior in this case. For our model, we assume that the set of Verifiable Credentials held by honest Self-Issued OPs is static. The issuance of new Verifiable Credentials we consider out of scope.

Note that we only model a highly abstracted view of the content of Verifiable Credentials, as their semantics could vary widely based on the specific RP needs and are not specified in [21]. To prevent the replay of Verifiable Credentials by unauthorized parties we use the Link Secrets Method as described in [3].

6.3 Model of Cross-Device Self-Issued OpenID Providers

In this section, we give an informal overview of the processes that form the model of the Cross-Device Self-Issued OpenID Provider web system with a network attacker and the aspects of the Self-Issued OpenID Provider specification that we include in this model. A Cross-Device Self-Issued OpenID Provider web system consists of browsers, RPs, Self-Issued OPs, Cross-Device Stubs, Configuration Providers and VDRs processes and a network attacker process.

6.3.1 Processes

In this section, we highlight the differences in the processes in the Cross-Device model from the Same-Device Self-Issued OP model. Note that the VDR process definition for the Cross-Device Self-Issued OpenID Provider web system is the same as in the same-device model and the Configuration Provider model is only adjusted to provide the Cross-Device Stub endpoint in an additional parameter in the metadata.

Attacker. The network attacker in the cross-device model is defined analogously to the attacker in the same-device model. Instead of allowing it to send messages to 'local' applications, we extend the attacker process to allow it to send QR code messages.

Relying Party. The RP process for the cross-device model is defined similarly to the same-device RP process. Core differences are that the Cross-Device RP provides an endpoint for the sync flow is added to the Cross-Device RP, that the Cross-Device RP redirects to the Cross-Device Stub rather than the Self-Issued OP when initiating a login flow and that the RP is adjusted to handle the cross-device response from an HTTP POST message directly from the Self-Issued OP. The Cross-Device RP is also simplified to not use signed requests.

Cross-Device Stub. The Cross-Device Stub is modeled as a generic HTTPS server. The Cross-Device Stub provides the sync start and sync fin endpoints for the sync flow. After the sync flow, it provides a script to retrieve the stub token, and after obtaining the stub token the Cross-Device Stub provides a script that models rendering the QR code containing the request in the browser.

Self-Issued OpenID Provider. The Self-Issued OP for the cross-device flow is adjusted to process QR codes instead of 'local' requests and check them for stub tokens, and also to support the cross-device response mode that instructs it to send the ID token directly to the RP in an HTTP POST message. Aside from this, its function is unchanged.

6.3.2 Concepts

The model of Cross-Device Self-Issued OP with the attack mitigation supports RP registration, DIDs and Verifiable Presentations as the same-device model. Self-Issued OP Discovery is extended with an additional parameter that allows discovering the Cross-Device Stub sync start endpoint for a Self-Issued OP.

Note that in the Cross-Device Self-Issued OP model with our mitigation we omit signed requests to focus on the core components for our mitigation. To utilize signed requests with our mitigation for the cross-device attack additional measures are necessary, in particular, to ensure the integrity of the redirection endpoint and the request nonce between Self-Issued OP and Cross-Device Stub. This could be achieved by the Cross-Device Stub resolving the redirection endpoint and the nonce for a signed request, or the Self-Issued OP verifying that the unsigned redirection endpoint and nonce the Cross-Device Stub uses are equal to the redirection endpoint and request nonce the Self-Issued OP extracted from the signed request. Due to the additional complexity that the support for signed requests add, the details of this, however, are out of the scope for this thesis.

6.4 Limitations

In this section, we briefly summarize some limitations of the Self-Issued OpenID Provider models in this thesis.

6.4.1 Isolated View

In this thesis, we model Same-Device Self-Issued OPs and our variant of Cross-Device Self-Issued OPs to mitigate the attack on the cross-device flow separately to keep the complexity of the protocols manageable. As a consequence, our analysis is not suitable to guarantee security in the case where Self-Issued OP or RP support both flows simultaneously. To allow both flows in parallel may require measures to isolate the flows, such as explicitly tracking the requested flow on the side of the RP and the Self-Issued OP (e.g., by distinguishing strictly by the requested response mode).

Similarly, the models of the Self-Issued OpenID Provider specification in this thesis only model the processes and functionality of Self-Issued OP and no other OpenID Connect or OAuth 2.0 flows to limit the scope of this thesis to a manageable size. As a result, the analysis in this thesis is not suitable to uncover vulnerabilities resulting from mix-ups between these flows or request forgery to applications and services outside the Self-Issued OP specification.

6.4.2 Invocation of Local Applications

Our model considers an abstract version of secure communication between the Self-Issued OP and the browser, such as App Links and Universal Links, as there is currently no known standard for this. As a consequence, it does not capture the details of particular implementations to achieve such secure communication.

6.4.3 Privacy Considerations

In this thesis, we do not analyze privacy considerations. Privacy, however, can be a major concern, when reusing identifiers at different RPs or if Verifiable Credentials contain sensitive data that an RP might not be authorized to request. Advanced concepts for privacy with Verifiable Credentials such as zero-knowledge proofs are not modeled in this thesis.

7 Security Properties

This section gives an informal overview of the security properties examined in this thesis. Precise, formal definitions of the properties are given in Appendix D. Note that we analyze the security properties, authentication, session integrity and holder binding, for our Same-Device Self-Issued OP model, but only consider the authentication property for the Cross-Device Self-Issued OP model in the scope of this thesis.

7.1 Same-Device Authentication

The authentication property states that an attacker must not be able to obtain a session the RP associates with an identity of an honest user.

As Self-Issued OPs are an extension of the OpenID Connect Implicit Flow, the authentication property is defined analogous to [6]. The notion of identity for Self-Issued OpenID Provider, however, differs and so do some details of the definition of the authentication property to match the use of Self-Issued Identities. For the support of DIDs, we need the VDR as an additional process participating in the flow.

Trivially, we need that all participating processes, the RP, the browser, the Self-Issued OP, and if the identity is an DID also the VDRs governing the identity to be honest. A malicious Self-Issued OP for example, could freely leak the private credentials for the identities of the user to the attacker and allow them to issue ID tokens for themselves. If the VDR governing a DID is malicious, it could allow the attacker to publish a public key for which they know the private key for an identity of an honest user. The RP then would accept ID tokens signed by the attacker's private key for the identity of the user.

7.2 Same-Device Session Integrity

For the session integrity property, we require that the user can only be logged in their browser if the user started a login process before. This property aims to prevent the case, where the attacker uses a CSRF attack to forcefully log the user in.

The property furthermore requires that the Self-Issued OP, the user chose when starting the login flow, participates to provide the identity the user is logged in as. The attacker cannot answer the authentication request by an RP instead of an honest Self-Issued OP if the user chose the Self-Issued OP in the browser. The session integrity property is similar to the session integrity for authentication in [6], however, in our model no user credentials are stored in the browser. The user interaction to authorize an authentication response for one of the user identities is moved from the browser posting a form to the OpenID Provider to the user directly interacting with the Self-Issued OP.

Here, we need again that the RP and browser are honest. Even if an Self-Issued OP is malicious it should not be able to forcefully log a user in. The login flow might be started giving a domain associated with any process, but if a log in via an honest Self-Issued OP is chosen in the browser, then we require that this Self-Issued OP answered the request. As resolving the authentication endpoint URL for the local Self-Issued OP requires an endpoint exposed to the internet, we need an additional party we call the Configuration Provider, that we require to be honest if the user chooses a domain associated with the Configuration Provider.

Note that we also use cookies with the `__Host` prefix, to achieve session integrity even for an active network attacker. This is necessary as general cookies do not provide integrity in the case of a network attacker — without the `__Host` prefix a network attacker could inject a session via unencrypted communication.

7.3 Same-Device Holder Binding

For holder binding, we require that the RP does only associate a Self-Issued Identity and a Verifiable Credential in a session if the same Self-Issued OP provides both and they thus belong to the same user. An attacker must not be able to inject a Verifiable Credential of themselves for an identity of the user, and an attacker must not be able to convince the RP to associate a Verifiable Credential of the user with an identity of the attacker.

For the holder binding property we require that the RP is honest. If the RP associates a Verifiable Credential with an identity we require that if the Self-Issued OP that is the holder of the identity or the Self-Issued OP that is the holder of the Verifiable Credential (and the browser of the user) is honest then that Self-Issued OP is the holder of both the identity and the Verifiable Credential. If the identity of a honest user is a DID we require that the VDR that governs this DID is honest.

7.4 Cross-Device Authentication

As for the same-device authentication property, the cross-device authentication property states that an attacker must not be able to obtain a session the RP associates with an identity of an honest user. As a consequence the definition of the cross-device authentication property is almost identical to the same-device case.

The differences are that we need additionally an honest Cross-Device Stub for this property. Also, we assume in the Cross-Device Self-Issued OP model that a user may enter a stub token at several browsers. As a consequence we require that all browsers that may obtain a stub token are honest.

8 Summary of Security Proofs

In this section, we give a brief summary of the security proofs for the previously sketched properties. The formal proofs of the security properties can be found in Appendix E.

8.1 Same-Device Authentication

For the attacker to obtain an authenticated session at the RP with an identity of the user, the attacker needs to provide an ID token for this identity. We first show that the RP only accepts ID tokens signed by the private key associated with an identity. This holds for DIDs if the governing VDR is honest, as then the RP will only associate each DID with the public keys in the correct DID document and use these public key for validating signatures.

As a result an attacker can only log in to the RP under the identity of a user if the attacker can provide an ID token for an identity of the user. If the Self-Issued OP of the user is honest, however, it will keep the private key for signing such ID tokens secret. Thus, only the Self-Issued OP of the user can create ID tokens for a user identity that the RP accepts.

We can show that an honest Self-Issued OP can not be confused about the redirection endpoint to send the response with the ID token to, and the Self-Issued OP ensures that it gives its response to the RP only via the local browser, which we assume to be honest. Thus, the attacker cannot obtain an ID token for a user identity that is usable at the RP. As a consequence, the attacker cannot obtain a session at the RP for a user identity.

8.2 Same-Device Session Integrity

For the proof of session integrity, we first show that for an honest RP to acknowledge a login for an honest browser under some identity, the attacker needs to present an ID token with the `nonce` parameter together with the associated session cookie in the browser. The RP only issues a session cookie and a `nonce` parameter combination for the browser, if the user started a login protocol flow in the browser. As the attacker cannot set a cookie issued for the attacker in an honest browser (note the `__Host` prefix), the user must have started a login flow at the RP if the RP logs the user in.

The attacker needs to inject a request with the correct `nonce` parameter into the browser, if the attacker wishes to log the user in with another identity than the user intended. We show that the attacker cannot learn this `nonce` parameter if RP, browser, Configuration Provider and Self-Issued OP are honest. This holds as the attacker can not confuse the RP about the authentication endpoint of the Self-Issued OP, can not intercept requests from browser to Self-Issued OP, and can not confuse

the Self-Issued OP about the redirection endpoint. The Self-Issued OP will only send responses to a redirection endpoint of the RP via the local browser. It follows that only the Self-Issued OP can provide an ID token that the RP accepts into the session.

8.3 Same-Device Holder Binding

For holder binding, we first show that, if the relevant parties are honest, an attacker can not obtain an ID token for an identity of a user or a Verifiable Presentation for a Verifiable Credential of the user respectively that the RP would accept.

We then show that the Verifiable Presentation and ID token associated in the RP session must have been created by the same honest Self-Issued OP. This holds as the ID token and the Verifiable Presentation need to be provided in the same message, and, as an attacker cannot know both the ID token and the Verifiable Presentation, the attacker cannot create this message. By reconstructing the path the message with ID token and the Verifiable Presentation have taken before being accepted into a session by the RP, we show that the attacker cannot abuse another process to combine an ID token or Verifiable Presentation of the attacker with an ID token or Verifiable Presentation of the user.

As an honest Self-Issued OP only creates ID tokens and Verifiable Presentations for which it holds the associated secret credentials, the attacker cannot confuse the RP to wrongly associate identities and Verifiable Credentials.

8.4 Cross-Device Authentication

To prove that the authentication property for the cross-device model holds, we show that a Self-Issued OP only sends a response with an ID token to a redirection endpoint for which there is a session with the RP associated with the redirection endpoint.

We show that the sync flow achieves that the request is associated by the RP with a session in the same browser as the Cross-Device Stub associates the request with. As the stub token can only be presented in a QR code in the user browser with content from the Cross-Device Stub, and the Cross-Device Stub only does this after the sync flow succeeded, we get that there is a session in the user browser with the RP and the request is associated with this session.

Similarly to the same-device case we also show that the Self-Issued OP cannot be confused about the redirection endpoint, and thus no attacker can obtain an ID token for an identity of the user that the RP accepts. Also, like in the same-device case, the RP cannot be confused about the key to use to verify the ID token signature. Thus, the RP only creates a service session for an identity of the user when the Self-Issued OP of the user provided the ID token and the RP associates this service session with a session in the browser of the user. It follows that only this browser can retrieve the service session cookie, and the attacker cannot obtain a service session for a user identity.

9 Conclusion and Outlook

In this thesis, we modeled the Self-Issued OpenID Provider same-device protocol flow together with the use of Verifiable Credentials in the WIM and analyzed its security. We also developed and analyzed a variant of the cross-device flow that hardens the flow against the known request replay attack on the cross-device flow.

For our model of the Self-Issued OpenID Provider same-device protocol flow, we defined a web system including Relying Party, Self-Issued OpenID Provider application, a web service for retrieving Self-Issued OpenID Provider metadata and an abstract version of a Verifiable Data Registry for retrieving the metadata for Decentralized Identifiers, as well as an extended browser to handle local application invocations. Similarly, we defined a web system for our variant of the cross-device flow with adjusted Relying Party and Self-Issued OpenID Provider, as well as an extended browser to model the rendering of QR codes and the additional Cross-Device Stub web service. On the modeled web systems we defined formal security properties and proved that they hold.

While proving that the security properties hold, we found a vulnerability in the specification of the Self-Issued OpenID Provider same-device flow that may result in the response for the authentication to leak to a malicious application on the user device. As a result, an attacker could impersonate the user. We contributed a security consideration to the specification to mitigate this problem.

We also contributed an extension to the WIM browser model that adds an abstract model of App-Links or Universal Links for the invocation of local applications such as the Self-Issued OpenID Provider application. A deeper analysis of the exact properties of different options to securely invoke applications from the web via URLs could be the subject of future work.

Verifiable Credentials and Presentations could hold sensitive information, and this thesis does not analyze privacy properties. Thus, formally analyzing the privacy implications of the use of Verifiable Credentials with Self-Issued OpenID Provider remains an open problem. The analysis in this thesis also does not encompass the issuance process of Verifiable Credentials, for which specifications such as *OpenID Connect for Verifiable Credential Issuance*, currently a draft by the OpenID Foundation, could be analyzed in future work.

Our formal analysis only captures Self-Issued OpenID Provider components of the Self-Issued OP same-device flow and cross-device flow in separate models. It also does not encompass other OAuth 2.0 protocol flows. As such future work could research the interaction between the Self-Issued OpenID Provider protocol flows and other OAuth 2.0 protocol flows to verify that the measures in the specification suffice to prevent protocol mix-up attacks.

Bibliography

[1] *Android App Links*. URL: https://developer.android.com/training/app-links (visited on 03/23/2022) (cit. on p. 22).

[2] *Android Deep Links*. URL: https://developer.android.com/training/app-links/deep-linking (visited on 03/23/2022) (cit. on p. 33).

[3] G. Cohen, O. Steele, W. Chang. *Presentation Exchange v1.0.0*. Ed. by D. Buchner, B. Zundel, M. Riedel. Decentralized Identity Foundation, 2021. URL: https://identity.foundation/presentation-exchange/spec/v1.0.0/ (visited on 03/23/2022). DIF Ratified Specification (cit. on pp. 21, 43, 56).

[4] Q. H. Do, P. Hosseyni, R. Kuesters, G. Schmitz, N. Wenzler, T. Wuertele. *A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification*. Cryptology ePrint Archive, Report 2021/1012. https://ia.cr/2021/1012. 2021 (cit. on p. 15).

[5] D. Fett, P. Hosseyni, R. Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 453–471. DOI: 10.1109/SP.2019.00067 (cit. on p. 15).

[6] D. Fett, R. Kuesters, G. Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: (Apr. 27, 2017). arXiv: 1704.08539 [cs.CR] (cit. on pp. 15, 37, 40, 47, 57, 64, 89, 95).

[7] D. Fett, R. Kusters, G. Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System". In: *2014 IEEE Symposium on Security and Privacy*. IEEE, May 2014. DOI: 10.1109/sp.2014.49 (cit. on pp. 15, 37).

[8] D. Fett, R. Küsters, G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *CoRR* abs/1601.01229 (2016). arXiv: 1601.01229. URL: http://arxiv.org/abs/1601.01229 (cit. on p. 15).

[9] D. Fett, R. Küsters, G. Schmitz. "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1358–1369. ISBN: 9781450338325. DOI: 10.1145/2810103.2813726. URL: https://doi.org/10.1145/2810103.2813726 (cit. on p. 15).

[10] D. Fett, R. Küsters, G. Schmitz. *The Web Infrastructure Model (WIM)*. Tech. rep. Version 1.0. SEC, University of Stuttgart, Germany, 2022. URL: https://www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf (cit. on pp. 15, 37–40, 57, 60, 61, 79, 81, 99, 117, 122, 125).

[11] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012. DOI: 10.17487/RFC6749. URL: https://rfc-editor.org/rfc/rfc6749.txt (visited on 03/23/2022) (cit. on p. 17).

[12] M. Jones, J. Bradley, N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: https://rfc-editor.org/rfc/rfc7515.txt (cit. on p. 17).

[13] M. Jones, J. Bradley, N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://rfc-editor.org/rfc/rfc7519.txt (cit. on p. 17).

[14] *PR 119: Security Consideration for confidentiality response*. Bitbucket Pull Request. Jan. 29, 2022. URL: https://bitbucket.org/openid/connect/pull-requests/119 (visited on 03/23/2022) (cit. on p. 35).

[15] N. Sakimura, J. Bradley, M. Jones. *OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1*. Nov. 8, 2014. URL: https://openid.net/specs/openid-connect-registration-1_0.html (visited on 03/23/2022) (cit. on p. 19).

[16] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, C. Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. Nov. 8, 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 03/23/2022) (cit. on pp. 15, 17).

[17] N. Sakimura, J.Bradley, M.Jones. *OpenID Connect Discovery 1.0 incorporating errata set 1*. OpenID Foundation, Nov. 8, 2014. URL: https://openid.net/specs/openid-connect-discovery-1_0.html (visited on 03/23/2022) (cit. on p. 19).

[18] M. Sporny, D. Longley, D. Chadwick. *Verifiable Credentials Data Model 1.0*. Ed. by M. Sporny, G. Noble, D. Longley, D. C. Burnett, B. Zundel. Nov. 19, 2019. URL: https://www.w3.org/TR/2021/REC-vc-data-model-20211109/ (visited on 03/23/2021) (cit. on pp. 15, 20, 56).

[19] M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, C. Allen. *Decentralized Identifiers (DIDs) v1.0*. Ed. by M. Sporny, A. Guy, M. Sabadello, D. Reed. Aug. 3, 2021. URL: https://www.w3.org/TR/2021/PR-did-core-20210803/ (visited on 03/23/2022) (cit. on pp. 20, 55).

[20] O. Steele, M. Sporny. *DID Specification Registries*. Ed. by O. Steele, M. Sporny, M. Prorock. 2021. URL: https://www.w3.org/TR/2021/NOTE-did-spec-registries-20211102/ (visited on 03/23/2022) (cit. on p. 20).

[21] O. Terbu, T. Lodderstedt, K. Yasuda, A. Lemmon, T. Looker. *OpenID Connect for Verifiable Presentations*. Jan. 28, 2022. URL: https://openid.net/specs/openid-connect-4-verifiable-presentations-1_0-08.html (visited on 02/15/2022) (cit. on pp. 15, 20, 43, 56).

[22] *Universal Links for Developers*. Apple Inc. URL: https://developer.apple.com/ios/universal-links/ (visited on 03/23/2022) (cit. on p. 22).

[23] K. Yasuda, M. B. Jones. "JWK Thumbprint URI". In: (Nov. 24, 2021). URL: https://www.ietf.org/archive/id/draft-jones-oauth-jwk-thumbprint-uri-00.html (visited on 02/15/2022) (cit. on p. 20).

[24] K. Yasuda, M. Jones. *Self-Issued OpenID Provider v2*. Standard. Jan. 28, 2022. URL: https://openid.net/specs/openid-connect-self-issued-v2-1_0-07.html (visited on 03/23/2022) (cit. on pp. 15, 17, 60).

All links were last followed on March 23, 2022.

A Data Structures

This section gives the formal model of data structures used in the models of Self-Issued OP in Appendix B and Appendix C.

A.1 Decentralized Identifiers

The data structures in this section describes the formal model of DIDs and DID Documents. These structures are specified in [19].

Definition A.1.1 (Decentralized Identifier)
Let $\mathsf{DIDMethod} \subset \mathbb{S}$ be a finite set, the set of all DID Methods. A *DID* is a term *did* of the form

$$did \equiv \langle \texttt{DID}, method, msid \rangle$$

with $method \in \mathsf{DIDMethod}$ and $msid \in \mathbb{S}$. The finite set of all DID terms is denoted DID. ◇

Definition A.1.2 (DID URL)
A *DID URL* is a term of the form $\langle did, key \rangle$ with $did \in \mathsf{DID}$ and $key \in \mathbb{S}$ (the relative identifier). The finite set of all DID URLs is denoted DIDURL. ◇

Definition A.1.3 (DID Document)
A *DID Document* is a term of the form

$$\langle \texttt{DIDDoc}, did, verification \rangle$$

with $did \in \mathsf{DID}$ and $verification \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ The finite set of all DID Documents is denoted DIDDoc. ◇

A.2 Self-Issued Identities

The Self-Issued OP specification defines two types of Self-Issued Identities, JWK Thumbprint URIs and DIDs. Formally, we describe Self-Issued Identities as follows.

Definition A.2.1 (Self-Issued Identity)
A *Self-Issued Identity* is a term *id* where either

- $id \in \mathsf{DID}$ (a Decentralized Identifier), or
- $id \equiv \langle \texttt{jkt}, \mathsf{hash}(\mathsf{pub}(k)) \rangle$ for a private signing key $k \in K^{\mathsf{ID}}_{\text{sign}}$ (a JWK Thumbprint URI).

With SIID we denote the finite set of Self-Issued Identities. ◇

ID tokens describe how identities are presented in OpenID Connect.

We model *Self-Issued ID tokens* as terms of the form

$$token \equiv \mathsf{sig}(\langle kid, token\_data\rangle, k)$$

with $kid \in \mathcal{T}_{\mathcal{N}}$, $k \in K^{\mathsf{ID}}_{\mathrm{sign}}$, $token\_data[\mathtt{nonce}] \not\equiv \langle\rangle$, $token\_data[\mathtt{iss}] \not\equiv \langle\rangle$ and $token\_data[\mathtt{aud}] \not\equiv \langle\rangle$ where exactly one of the following holds

- $token\_data[\mathtt{sub}] \in \mathsf{DID}$ and $token\_data[\mathtt{sub\_jwk}] \equiv \langle\rangle$ and $kid \in \mathsf{DIDURL}$, or
- $token\_data[\mathtt{sub\_jwk}] \not\equiv \langle\rangle$ and $token\_data[\mathtt{sub}] \equiv \langle \mathtt{jkt}, \mathsf{hash}(token\_data[\mathtt{sub\_jwk}])\rangle$.

A.3 Verifiable Credentials

In this section we describe our model of Verifiable Credentials and Verifiable Presentations, structures specified in [18], together with considerations for Self-Issued OP as in [21].

Definition A.3.1 (Verifiable Credential)
A *Verifiable Credential* is a term *vc* of the form

$$vc \equiv \mathsf{sig}(data, k)$$

with $data \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ and $k \in K^{\mathsf{VC}}_{\mathrm{sign}}$ such that $data[\mathtt{context}] \equiv \mathtt{VC}$, $data[\mathtt{issuer}] \equiv \mathsf{pub}(k)$, $data[\mathtt{type}] \in \mathbb{S}$ and $data[\mathtt{subject}] \in \mathcal{T}_{\mathcal{N}}$. The finite set of Verifiable Credentials is denoted VC.$\diamond$

We model *Verifiable Presentations* as terms *vp* of the form $vp \equiv \mathsf{sig}(vp\_data, k)$ with $vp\_data \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ and $k \in K^{\mathsf{VP}}_{\mathrm{sign}}$ such that $vp\_data[\mathtt{vc}] \in \mathsf{VC}$, $\mathsf{extractmsg}(vp\_data[\mathtt{vc}])[\mathtt{subject}] \equiv \mathsf{pub}(k)$, $vp\_data[\mathtt{challenge}] \not\equiv \langle\rangle$ and $vp\_data[\mathtt{audience}] \in \mathsf{URLs}$.

Note that our model of Verifiable Presentations binds claims to their holder via the 'Link Secrets' method described in [3]. Using this method the Verifiable Credentials is issued to a ephemeral public verification key the holder provides to the issuer where the holder (exclusively) knows the corresponding private signing key. A valid Verifiable Presentation for a Verifiable Credential then can only be created by the holder.

A.4 QR codes

In this section we describe our model of QR codes that we need for the cross-device Self-Issued OP model.

Definition A.4.1 (QR code)
A *QR code* is a term *qr* of the form $qr \equiv \langle \mathtt{QR}, content\rangle$ with $content \in \mathcal{T}_{\mathcal{N}}$. $\diamond$

Our browser model in supports a new command $\langle \mathtt{ShowQR}, content\rangle$ on which the browser sends such a QR code message to another process.

B Formal Model of Same-Device Self-Issued OpenID Providers

In this section we define the formal model of Self-Issued OPs in the WIM that forms the foundation of our analysis of the Self-Issued OP same-device flow. In our model we use the browser and the generic HTTPS server model as defined in [10]. Note that due to the similar semantic of the processes our model of the Self-Issued OP and the RP hold significant similarity to the OP and RP defined in [6].

B.1 Web System

We model Same-Device Self-Issued OPs as a web system: A Same-Device Self-Issued OpenID Provider web system with a network attacker is a web system $\mathcal{SIOID}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ of the form as follows.

The system $\mathcal{W}$ consists of a network attacker in Net, a finite set of browsers B, a finite set CP of Configuration Provider web servers, a finite set RP of Same-Device RP web servers, a finite set SIOP of Same-Device Self-Issued OpenID Providers and a finite set VDR of Verifiable Data Registry web servers. It is $\mathcal{W} = \mathsf{Net} \cup \mathsf{Hon}$ with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{CP} \cup \mathsf{SIOP} \cup \mathsf{VDR}$.

The set of scripts $\mathcal{S}$ and the mapping script we define as follows:

- $R^{\text{att}} \in \mathcal{S}$ with $\mathsf{script}(R^{\text{att}})$ = `att_script` (the attacker script),
- *script_rp_index* $\in \mathcal{S}$ mapped by script to `script_rp_index` (a script simulating the index page of the RP), and
- *script_rp_get_fragment* $\in \mathcal{S}$ mapped by script to `script_rp_get_fragment` (a script for obtaining the fragment in the authentication response to the RP).

As usual for the WIM, we initiate E^0 with an infinite (mathematical) sequence of trigger events $\langle a, a, \texttt{TRIGGER}\rangle$ for each $a \in \mathsf{IPs}$. The definitions of the processes are given in Appendices B.2 to B.6.

B.1.1 Addresses and Domains

The set IPs contains a finite set of addresses for the network attacker in Net, for every web browser in B, for every Same-Device RP in RP, for every Configuration Provider CP and for every Verifiable Data Registry in VDR and a single address for every Same-Device Self-Issued OP in SIOP. The

assignment from atomic processes to their subset of IPs is denoted by addr. By Doms we denote a finite set of domains, containing domains for the network attacker in Net, for every Same-Device RP in RP, for every Configuration Provider in CP and for every Verifiable Data Registry in VDR.

Let configProvider : SIOP $\to$ CP be a mapping assigning each Same-Device Self-Issued OP a Configuration Provider. The domains associated of $i \in$ SIOP are the domains of the Configuration Provider configProvider(i). Note that some distinct Self-Issued OP processes might have the same domain. The assignment from atomic DY processes to their subset of Doms is denoted by dom.

B.1.2 Nonces and Keys

We partition the set of nonces $\mathcal{N}$ into sets

$$\mathcal{N} := N \dot\cup K_{\text{TLS}} \dot\cup K_{\text{local}} \dot\cup K_{\text{sign}}$$

with N an infinite set and finite sets $K_{\text{TLS}}, K_{\text{local}}$ and K_{sign}.

Nonces in N are available for each atomic DY process in $\mathcal{W}$ and are used for dynamically generated secrets such as session identifiers and message nonces.

The set K_{TLS} contains secret keys for TLS encryption. Let tlskey : Doms $\to K_{\text{TLS}}$ be an injective mapping that assigns a private key to each domain. We define for each atomic DY process y the set $\text{tlskeys}^y = \{\langle d, \text{tlskey}(d)\rangle \mid d \in \text{dom}(y)\}$. We also define keyMapping $= \{\langle d, \text{pub}(\text{tlskey}(d))\rangle \mid y \in \mathcal{W}, d \in \text{dom}(y)\}$.

The set K_{local} contains secret encryption keys to formalize the confidentiality of local Self-Issued OP calls. Let localkey : SIOP $\cup$ B $\to K_{\text{local}}$ be an injective mapping that assigns a private key to each Same-Device Self-Issued OP and browser.

The nonces in K_{sign} are secret signing keys and again partitioned into four sets: $K_{\text{sign}} := K^{\text{ID}}_{\text{sign}} \dot\cup K^{\text{RP}}_{\text{sign}} \dot\cup K^{\text{VC}}_{\text{sign}} \dot\cup K^{\text{VP}}_{\text{sign}}$. The set $K^{\text{RP}}_{\text{sign}}$ contains signing keys for the requests of Relying Parties, while $K^{\text{ID}}_{\text{sign}}$ contains signing keys for ID tokens belonging to user identities held by a Same-Device Self-Issued OP. Let rp_signkey : RP $\to K^{\text{RP}}_{\text{sign}}$ denote an injective mapping that assigns a signing key to each Relying Party. The set $K^{\text{VC}}_{\text{sign}}$ contains signing keys for signing Verifiable Credentials, the set $K^{\text{VP}}_{\text{sign}}$ contains signing keys expected to be held by Self-Issued OpenID Providers and to be used for signing Verifiable Presentations.

B.1.3 DID governor

Let method_gov : DIDMethod $\to \mathcal{W}$ be an injective mapping. We define the mapping

$$\text{govenor} : \text{DID} \to \mathcal{W}, \langle \texttt{DID}, m, i\rangle \mapsto \text{method\_gov}(m).$$

For an atomic DY process y we denote with DIDDoc^y the set $\{d \mid d \in \text{DIDDoc}, d.\texttt{did} \in \text{govenor}^{-1}(y)\}$, which we restrict with that there for each $did \in$ DID is at most one $doc \in \text{DIDDoc}^y$ with $doc.\texttt{did} \equiv did$ if the process y is an honest VDR. We need this restriction as there is no notion of time in the WIM and to ensure that only one DID Document is valid at a time. Each Verifiable Data Registry in VDR governs a DID method and holds all DID Documents for this method.

The attacker also might govern a DID Method and hold DID Documents for this method. Let $\mathsf{resolveDoc} : \mathsf{DID} \to \mathsf{DIDDoc} \cup \{\bot\}$ be an injective mapping which maps *did* to a DID Document *doc* with $doc.\texttt{did} \equiv did$ and $doc \in \mathsf{DIDDoc}^{\mathsf{govenor}(did)}$ if possible and $\bot$ otherwise.

B.1.4 Identity Holder

Let $\mathsf{id\_holder} : \mathsf{SIID} \to \mathcal{W}$ be a mapping that maps each Self-Issued Identity to an atomic DY process we call the Identity Holder. By SIID^y for $y \in \mathcal{W}$ we denote the set $\mathsf{id\_holder}^{-1}(y)$. While typically Self-Issued OPs are the Identity Holders of Self-Issued Identities, the attacker might also be the Identity Holder for some Self-Issued Identities. Let vkid be a mapping that maps Self-Issued Identities $id \in \mathsf{SIID}$ to a set of DID URLs $\langle id, v\rangle$ with $v \in \mathsf{resolveDoc}(id).\texttt{verification}$ if $id \in \mathsf{DID}$ and $v \equiv \bot$ otherwise. We define the mapping proof_key to map SIID to a set of $K^{\mathsf{ID}}_{\text{sign}}$ such that

$$\begin{aligned} k \in \mathsf{proof\_key}(id) \iff\ & (id \equiv \langle \texttt{jkt}, \mathsf{hash}(\mathsf{pub}(k))\rangle) \\ & \vee (id \in \mathsf{DID} \wedge kid \in \mathsf{vkid}(id) \wedge doc \equiv \mathsf{resolveDoc}(id) \\ & \quad \wedge \mathsf{pub}(k) \equiv doc.\texttt{verification}[kid.\texttt{key}]) \end{aligned}$$

Let kid be a mapping from Self-Issued identities $id \in \mathsf{SIID}$ such that $\mathsf{kid}(id) \in \mathsf{vkid}(id)$ is a fixed element if $id \in \mathsf{DID}$ and $\mathsf{kid}(id) \equiv \bot$ otherwise. Let kid be a mapping from Self-Issued identities $id \in \mathsf{SIID}$ to nonces in $K^{\mathsf{ID}}_{\text{sign}}$ such that $\mathsf{keyForID}(id) = k$ with $k \in \mathsf{proof\_key}(id) \subset K^{\mathsf{ID}}_{\text{sign}}$ be the uniquely defined element, such that $id \equiv \langle \texttt{jkt}, \mathsf{hash}(\mathsf{pub}(k))\rangle$, or $id \in \mathsf{DID}$, $doc \equiv \mathsf{resolveDoc}(id)$ and $\mathsf{pub}(k) \equiv doc.\texttt{verification}[\mathsf{kid}(id).\texttt{key}]$. We define for each atomic DY process y the set of identities with their associated key material for which the process is the Identity Holder as

$$\mathsf{ID\_Records}^y := \{[\texttt{id} : id, \texttt{kid} : \mathsf{kid}(id), \texttt{key} : \mathsf{keyForID}(id)] \mid id \in \mathsf{SIID}^y\}.$$

B.1.5 Credential Holder

Let $\mathsf{key\_holder} : K^{\mathsf{VP}}_{\text{sign}} \to \mathcal{W}$ be a mapping that assigns each signing key to a process. We define the mapping $\mathsf{vc\_holder} : \mathsf{VC} \to \mathcal{W}$ with $vc \mapsto y$ if and only if there is a key $k \in K^{\mathsf{VP}}_{\text{sign}}$ with $\mathsf{key\_holder}(k) = y$ and $\mathsf{extractmsg}(vc)[\texttt{subject}] \equiv \mathsf{pub}(k)$. We denote for each atomic DY process $y \in \mathcal{W}$ the set of Verifiable Credential records held by y as

$$\mathsf{VCW}^y := \{[\texttt{vc} : vc, \texttt{key} : k] \mid \mathsf{key\_holder}(k) = \mathsf{vc\_holder}(vc) = y\}.$$

We expect such a process y to be a Self-Issued OP, but it might also be the attacker.

B.1.6 Attacker

We consider a network attacker as defined for the WIM. The network attacker *att* in Net can listen to all IP addresses, thus we define $I^{att} = \mathsf{IPs}$. The initial state of the attacker is

$$s_0^{att} = \langle\langle \mathsf{tlskeys}^{att}\rangle, \langle \mathsf{keyMapping}\rangle, \langle \mathsf{ID\_Records}^{att}\rangle, \langle \mathsf{VCW}^{att}\rangle, pubSigKeys, pubLocalKeys\rangle$$

where $pubSigKeys \equiv \langle\{\mathsf{pub}(k) \mid k \in K_{\text{sign}}\}\rangle$ and $pubLocalKeys \equiv \langle\{\mathsf{pub}(k) \mid k \in K_{\text{local}}\}\rangle$.

The attacker knows all public keys, in particular they know the public keys for the local calls, meaning that they can make 'local' calls themselves. This models the case where the attacker controls an application that resides on the same device as a Self-Issued OP or a browser.

A network attacker can intercept and inject messages. As a consequence the network attacker controls DNS which is typically not protected. Let $dnsAddress^{\mathsf{Net}} \in \mathsf{IPs}$ be an address of the network attacker. We use this address in the state definitions of the other processes as the DNS server address.

B.1.7 Corruption

The Self-Issued OPs, RPs, Configuration Providers and the Verifiable Data Registries in the model are extensions of the generic HTTPS server model from [10] and can become corrupted as described in it: When they receive the message `CORRUPT`, the corruption flag ($s.\mathtt{corrupt}$) in their state is set to some other value than $\bot$. They then store all incoming messages on their state and send some message derivable from their state to some atomic DY process.

We call a Self-Issued OP, an RP or a Verifiable Data Registry honest, if $s.\mathtt{corrupt} \equiv \bot$, otherwise we say they are corrupted. Analogously, we call a browser honest, if in its state $s.\mathtt{isCorrupted} \equiv \bot$, otherwise we call it corrupted.

B.2 Browser Extension

In this section we define the extension of the original WIM browser used in this thesis. Each process $b \in \mathsf{B}$ is atomic DY process with addresses $I^b := \mathsf{addr}(b)$.

For defining the browser for our model we first need to define a notion of 'local' applications. Let $\mathsf{browser} : \mathsf{SIOP} \to \mathsf{B}$ be a mapping from Same-Device Self-Issued OPs to the browser running on the same device with $i_1 \neq i_2 \wedge \mathsf{dom}(i_1) \cap \mathsf{dom}(i_2) \neq \emptyset \Rightarrow \mathsf{browser}(i_1) \neq \mathsf{browser}(i_2)$[1]. We define for each browser b the set

$$\begin{aligned}\mathsf{LocalLinks}^b := \{\langle url, \langle address, pubKey\rangle\rangle \mid\ & i \in \mathsf{browser}^{-1}(b),\\ & pubKey = \mathsf{pub}(\mathsf{localkey}(i)), address \in \mathsf{addr}(i), d \in \mathsf{dom}(i),\\ & url \equiv \langle \mathtt{URL}, \mathsf{S}, d, /\mathtt{auth}, \langle\rangle, \bot\rangle\}\end{aligned}$$

which links URLs of Same-Device Self-Issued OPs to their respective address.

Definition B.2.1
A *state* $s \in Z^b$ *of a browser* b is a term of the form $\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping, sts, DNSaddress, pendingDNS, pendingRequests, isCorrupted, localLinks, localCallKey\rangle$ with the terms $windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping, sts, DNSaddress, pendingDNS, pendingRequests$ and $isCorrupted$ as in the original WIM browser definition, and $localLinks \in \left[\mathsf{URLs} \times \mathcal{T}_{\mathcal{N}}\right]$ and $localCallKey \in K_{\text{local}}$.

[1] We assume no two Same-Device Self-Issued OPs with the same domain can be registered at the same browser, see the known security issue with respect to custom URL schemes in the security considerations of [24].

Algorithm B.1 Web Browser Model Extension: Prepare headers, do DNS resolution, save message.

```
1: function HTTP_SEND(reference, message, url, origin, referrer, referrerPolicy, a, s′)
2:     if message.host ∈⟨⟩ s′.sts then
3:         let url.protocol := S
4:     end if
5:     let cookies := ⟨{⟨c.name, c.content.value⟩|c ∈⟨⟩ s′.cookies [message.host]
       ↪ ∧ (c.content.secure ⟹ (url.protocol = S))}⟩
6:     let message.headers[Cookie] := cookies
7:     if origin ≢ ⊥ then
8:         let message.headers[Origin] := origin
9:     end if
10:    if referrerPolicy ≡ noreferrer then
11:        let referrer := ⊥
12:    end if
13:    if referrer ≢ ⊥ then
14:        if referrerPolicy ≡ origin then
15:            let referrer := ⟨URL, referrer.protocol, referrer.host, /, ⟨⟩, ⊥⟩
16:        end if
17:        let referrer.fragment := ⊥
18:        let message.headers[Referer] := referrer
19:    end if
20:    let simpleUrl := ⟨URL, url.protocol, url.host, url.path, ⟨⟩, ⊥⟩
21:    if simpleUrl ∈ s′.localLinks then
           ↪ → Skip DNS for registered links to local apps.
22:        let addr := s′.localLinks[simpleUrl].address
23:        let pubKey := s′.localLinks[simpleUrl].pubKey
24:        let s′.pendingRequests := s′.pendingRequests
           ↪ +⟨⟩ ⟨reference, message, url, v_local_https, addr⟩
25:        let message := enc_a(⟨message, v_local_https⟩, pubKey)
               ↪ → Requests to 'local' apps are always encrypted.
26:        stop ⟨⟨s′.localLinks[simpleUrl], a, message⟩⟩, s′
27:    end if
28:    let s′.pendingDNS[v_8] := ⟨reference, message, url⟩
29:    stop ⟨⟨s′.DNSaddress, a, ⟨DNSResolve, message.host, v_8⟩⟩⟩, s′
30: end function
```

An *initial state* s_0^b *of* b is a state of b with $s_0^b.\texttt{windows} \equiv \langle\rangle$, $s_0^b.\texttt{ids} \equiv \langle\rangle$, $s_0^b.\texttt{secrets} \equiv \langle\rangle$, $s_0^b.\texttt{cookies} \equiv \langle\rangle$, $s_0^b.\texttt{localStorage} \equiv \langle\rangle$, $s_0^b.\texttt{sessionStorage} \equiv \langle\rangle$, $s_0^b.\texttt{keyMapping} \equiv \langle\text{keyMapping}\rangle$, $s_0^b.\texttt{sts} \equiv \langle\rangle$, $s_0^b.\texttt{DNSaddress} \equiv dnsAddress^{\text{Net}}$, $s_0^b.\texttt{pendingDNS} \equiv \langle\rangle$, $s_0^b.\texttt{pendingRequests} \equiv \langle\rangle$, $s_0^b.\texttt{isCorrupted} \equiv \bot$, $s_0^b.\texttt{localLinks} \equiv \langle\text{LocalLinks}^b\rangle$, $s_0^b.\texttt{localCallKey} \equiv \text{localkey}(b)$. ◇

We use the WIM browser as defined in [10] with adjustments to the browser relation for our extension that we give in Algorithm B.1 and Algorithm B.2 where our extensions marked in blue.

Algorithm B.2 Web Browser Model Extension: Process an HTTP response.

```
1: function PROCESSRESPONSE(response, reference, request, requestUrl, key, f, s′)
2:     if Set-Cookie ∈ response.headers then
3:         for each c ∈^⟨⟩ response.headers [Set-Cookie], c ∈ Cookies do
4:             let s′.cookies [request.host]
                   ↪ := AddCookie(s′.cookies [request.host], c, requestUrl.protocol)
5:         end for
6:     end if
7:     if Strict-Transport-Security ∈ response.headers ∧ requestUrl.protocol ≡ S then
8:         let s′.sts := s′.sts +^⟨⟩ request.host
9:     end if
10:    if Referer ∈ request.headers then
11:        let referrer := request.headers[Referer]
12:    else
13:        let referrer := ⊥
14:    end if
15:    if Location ∈ response.headers ∧ response.status ∈ {303, 307} then
16:        let url := response.headers [Location]
17:        if LocalResponse ∈ response.headers then
18:            let url := dec_a(url, s′.localCallKey)
19:        end if
20:        if url.fragment ≡ ⊥ then
21:            let url.fragment := requestUrl.fragment
22:        end if
23:        let method′ := request.method
24:        let body′ := request.body
25:        if Origin ∈ request.headers
             ↪ ∧ (⟨url.host, url.protocol⟩ ≡ ⟨request.host, requestUrl.protocol⟩
             ↪ ∨ ⟨request.host, requestUrl.protocol⟩ ≡ request.headers[Origin]) then
26:            let origin := request.headers[Origin]
27:        else
28:            let origin := ⊥
29:        end if
30:        if response.status ≡ 303 ∧ request.method ∉ {GET, HEAD} then
31:            let method′ := GET
32:            let body′ := ⟨⟩
33:        end if
34:        if ∃w̄ ∈ Subwindows(s′) such that s′.w̄.nonce ≡ π2(reference) then
35:            let req := ⟨HTTPReq, v6, method′, url.host, url.path, url.parameters, ⟨⟩, body′⟩
36:            let referrerPolicy := response.headers[ReferrerPolicy]
37:            call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, s′)
38:        end if
39:    end if
```

```
40:     switch π1(reference) do
41:         case REQ
42:             let w̄ ← Subwindows(s′) such that s′.w̄.nonce ≡ π2(reference) if possible;
                ↪ otherwise stop
43:             if response.body ≁ ⟨∗, ∗⟩ then
44:                 stop ⟨⟩, s′
45:             end if
46:             let script := π1(response.body)
47:             let scriptstate := π2(response.body)
48:             let d := ⟨v7, requestUrl, response.headers, referrer, script, scriptstate, ⟨⟩, ⟨⟩, ⊤⟩
49:             if s′.w̄.documents ≡ ⟨⟩ then
50:                 let s′.w̄.documents := ⟨d⟩
51:             else
52:                 let ī ← ℕ such that s′.w̄.documents.ī.active ≡ ⊤
53:                 let s′.w̄.documents.ī.active := ⊥
54:                 remove s′.w̄.documents.(ī + 1) and all following documents
                    ↪ from s′.w̄.documents
55:                 let s′.w̄.documents := s′.w̄.documents +⟨⟩ d
56:             end if
57:             stop ⟨⟩, s′
58:         case XHR
59:             let w̄ ← Subwindows(s′), d̄ such that s′.d̄.nonce ≡ π2(reference)
                ↪ ∧ s′.d̄ = s′.w̄.activedocument if possible; otherwise stop
60:             let headers := response.headers − Set-Cookie
61:             let s′.d̄.scriptinputs := s′.d̄.scriptinputs +⟨⟩
                ⟨XMLHTTPREQUEST, headers, response.body, π3(reference)⟩
62:             stop ⟨⟩, s′
63: end function
```

B.3 Same-Device Relying Party

A Same-Device 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). The addresses are defined as $I^r := \mathsf{addr}(r)$.

A Same-Device RP might trust or support only a subset of DID Methods. Let $\mathsf{supportedMethods}^r$ be a subset of $\mathsf{DIDMethod}$ for each $r \in \mathsf{RP}$. We define $\mathsf{didResolvers}^r := \{\langle m, d\rangle \mid \mathsf{method\_gov}(m) = v, m \in \mathsf{supportedMethods}^r, d \in \mathsf{dom}(v)\}$ the set of supported DID methods together with the governing Verifiable Data Registry domains for each Relying Party $r \in \mathsf{RP}$. Let $\mathsf{trustedVCIss}$ be a mapping that maps each Relying Party $r \in \mathsf{RP}$ to a subset of $\{\mathsf{pub}(k) \mid k \in K^{\mathsf{VC}}_{\mathrm{sign}}\}$, the public keys of Verifiable Credentials issuers. By this we formalize a notion of trust into a subset of Verifiable Credential issuers.

Definition B.3.1
A *state* $s \in Z^r$ *of an Same-Device RP* r is a term of the form $\langle$*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*, *sessions*, *oidcConfigCache*, *requestSigKey*, *didCache*, *didResolvers*, *trustedVCIss*$\rangle$ 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* $\in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (components as in the generic HTTPS web server model), *sessions* $\in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, *oidcConfigCache* $\in [\mathsf{Doms} \times \mathsf{URLs}]$, *requestSigKey* $\in K^{\mathsf{RP}}_{\mathrm{sign}}$, *didCache* $\in [\mathsf{DID} \times \mathsf{DIDDoc}]$, *didResolvers* $\in [\mathbb{S} \times \mathsf{Doms}]$ and *trustedVCIss* $\subset^{\langle\rangle} K^{\mathsf{VC}}_{\mathrm{sign}}$.

An *initial state* s_0^r *of* r is a state of r with $s_0^r.\texttt{DNSAddress} \equiv dnsAddress^{\mathsf{Net}}$, $s_0^r.\texttt{pendingDNS} \equiv \langle\rangle$, $s_0^r.\texttt{pendingRequests} \equiv \langle\rangle$, $s_0^r.\texttt{corrupt} \equiv \bot$, $s_0^r.\texttt{keyMapping} \equiv \langle \mathsf{keyMapping} \rangle$, $s_0^r.\texttt{tlskeys} \equiv \langle \mathsf{tlskeys}^r \rangle$ (components of the generic HTTPS server model), and $s_0^r.\texttt{sessions} \equiv \langle\rangle$, $s_0^r.\texttt{oidcConfigCache} \equiv \langle\rangle$, $s_0^r.\texttt{requestSigKey} \equiv \mathsf{rp\_signkey}(r)$, $s_0^r.\texttt{didCache} \equiv \langle\rangle$, $s_0^r.\texttt{didResolvers} \equiv \langle \mathsf{didResolvers}^r \rangle$, and $s_0^r.\texttt{trustedVCIss} \equiv \langle \mathsf{trustedVCIss}(r) \rangle$. ◇

In the following, we specify the relation R^r of the Same-Device Relying Party. Our model extends the generic HTTPS server model, thus we only specify the additional algorithms. These algorithms are given in Algorithms B.6 to B.13. In these algorithms, we use placeholders to generate nonces for which we give a list here:

- $\nu_{\texttt{rp\_login}}$: Session ID for new login flows, key in *sessions*, in Algorithm B.5.
- $\nu_{\texttt{did\_req}}$: Request nonce for DID resolution in Algorithm B.12.
- $\nu_{\texttt{discover\_req}}$: Request nonce for Self-Issued OP discovery request in Algorithm B.13.
- $\nu_{\texttt{rp\_req\_nonce}}$: Nonce for the `nonce` parameter in the authentication request in Algorithm B.13.
- $\nu_{\texttt{reg\_ref}}$: Nonce for relating registration requests from the Self-Issued OP to the associated session in Algorithm B.13.
- $\nu_{\texttt{req\_ref}}$: Nonce for relating requests for request objects to the associated session in Algorithm B.13.
- $\nu_{\texttt{service\_session}}$: Service Session ID (for logged in users) in Algorithm B.9.

The Same-Device RP model uses two scripts, *script_rp_get_fragment* and *script_rp_index*, similar to the scripts in [6] but with some adjustments given in Algorithm B.3 (changes marked in blue) and Algorithm B.4 (issuer parameter is not included in our version as we do not need it). Both scripts use $\mathsf{GETURL}(tree, docnonce)$ defined as in [6]. This function GETURL retrieves the document d identified by *docnonce* from the document tree *tree* and returns the URL $d.\texttt{location}$. Note that we use cookies with the `__Host` prefix, as we wish to achieve security against a network attacker.

Some RPs might be part of a trust framework that allows to resolve trusted metadata about the RP, in particular a public verification key. We define for each $i \in \mathsf{SIOP}$ the term $\mathsf{RRP}^i \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ where domains in $\mathsf{dom}(r)$ are mapped to $\mathsf{rp\_signkey}(r)$ for $r \in \mathsf{RP}$.

Algorithm B.3 Relation of *script_rp_index*.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$

1: **let** *switch* $\leftarrow \{\texttt{auth}, \texttt{link}\}$ → Non-deterministically decide whether to start a login flow or to follow some link.
2: **if** *switch* $\equiv$ `auth` **then** → **Start login flow.**
3: **let** *url* := GETURL(*tree*, *docnonce*) → Retrieve own URL.
4: **let** *url′* := $\langle$`URL`, `S`, *url*.`host`, `/startLogin`, $\langle\rangle$, $\bot\rangle$ → Assemble the login request URL.
5: **let** *host* $\leftarrow$ Doms → Non-det. select issuer URL host.
6: **let** *iss_id* := $\langle$`URL`, `S`, *host*, $\langle\rangle$, $\langle\rangle\rangle$ → Assemble issuer identifier URL.
7: **let** *command* := $\langle$`FORM`, *url′*, `POST`, *iss_id*, $\bot\rangle$ → Post a form including the desired issuer URL to the RP.
8: **stop** $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$
9: **else** → **Follow some link.**
10: **let** *protocol* $\leftarrow$ {`P`, `S`} → Non-deterministically select protocol (HTTP or HTTPS).
11: **let** *host* $\leftarrow$ Doms → Non-det. select host.
12: **let** *path* $\leftarrow \mathbb{S}$ → Non-det. select path.
13: **let** *fragment* $\leftarrow \mathbb{S}$ → Non-det. select fragment part.
14: **let** *parameters* $\leftarrow [\mathbb{S} \times \mathbb{S}]$ → Non-det. select parameters.
15: **let** *url* := $\langle$`URL`, *protocol*, *host*, *path*, *parameters*, *fragment*$\rangle$ → Assemble URL.
16: **let** *command* := $\langle$`HREF`, *url*, $\bot$, $\bot\rangle$ → Follow link to the selected URL.
17: **stop** $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$
18: **end if**

Algorithm B.4 Relation of *script_rp_get_fragment*.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$

1: **let** *url* := GETURL(*tree*, *docnonce*)
2: **let** *url′* := $\langle$`URL`, `S`, *url*.`host`, `/redirect_ep`, $\langle\rangle$, $\bot\rangle$
3: **let** *command* := $\langle$`FORM`, *url′*, `POST`, *url*.`fragment`, $\bot\rangle$
4: **stop** $\langle s, cookies, localStorage, sessionStorage, command\rangle$

Algorithm B.5 Same-Device Relying Party R^r: Process an HTTPS request.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s′)   → Process an incoming HTTPS request.
2:     if m.path ≡ / then   → Serve index page.
3:         let headers := [ReferrerPolicy : origin]
4:         let m′ := enc_s(⟨HTTPResp, m.nonce, 200, headers, ⟨script_rp_index, ⟨⟩⟩⟩, k)
5:         stop ⟨⟨f, a, m′⟩⟩, s′   → Send script_rp_index in HTTP response.
6:     else if m.path ≡ /startLogin ∧ m.method ≡ POST then   → Serve request to start login.
7:         if m.headers[Origin] ≢ ⟨m.host, S⟩ then   → CSRF protection
8:             stop
9:         end if
10:        let issuer_id := m.body   → Retrieve the requested issuer identifier.
11:        if issuer_id ∉ URLs ∨ issuer_id.protocol ≢ S ∨ issuer_id.parameters ≢ ⟨⟩
               ↪ ∨ issuer_id.fragment ≢ ⊥ then
12:            stop
13:        end if
14:        let sessionId := ν_rp_login   → Choose fresh nonce for session id.
15:        let s′.sessions[sessionId] := [startRequest : [message:m, key : k,
               ↪ receiver : a, sender : f], issuer : issuer_id]
16:        call START_LOGIN_FLOW(sessionId, s′)
17:    else if m.path ≡ /registration then
18:        call SEND_REGISTRATION_METADATA(m, k, a, f, s′)
19:    else if m.path ≡ /request_ep then
20:        call SEND_REQUEST_OBJECT(m, k, a, f, s′)
21:    else if m.path ≡ /redirect_ep then
22:        let sessionId := m.headers[Cookie][⟨__Host, sessionId⟩]
23:        if sessionId ∉ s′.sessions then
24:            stop
25:        end if
26:        let session := s′.sessions[sessionId]   → Retrieve session data.
27:        if m.method ≡ GET then
28:            let headers := ⟨⟨ReferrerPolicy, origin⟩⟩
29:            let m′ := enc_s(⟨HTTPResp, m.nonce, 200, headers, ⟨script_rp_get_fragment, ⟨⟩⟩⟩, k)
30:            stop ⟨⟨f, a, m′⟩⟩, s′   → Send script_rp_get_fragment in response.
31:        else if m.method ≡ POST then
32:            let data := m.body
33:        else
34:            stop
35:        end if
36:        let s′.sessions[sessionId][redirect_ep_req] := [message : m, key : k,
               ↪ receiver : a, sender : f]
37:        let s′.sessions[sessionId][id_token] := data[id_token]
38:        if vp_token ∈ data then
39:            let s′.sessions[sessionId][vp_token] := data[vp_token]
40:        end if
41:        call CHECK_ID_TOKEN(sessionId, a, s′)
42:    end if
43: end function
```

Algorithm B.6 (Same-Device) Relying Party R^r: Process an HTTPS response.

```
1: function PROCESS_HTTPS_RESPONSE(m, reference, request, a, f, s′)   → Process an HTTPS response.
2:     let session := s′.sessions[reference[session]]
3:     if reference[responseTo] ≡ DID_RESOLVE then
4:         let doc := m.body
5:         if doc.did ≢ reference[requestedDID] then
               ↪   → DID document for requested DID.
6:             stop
7:         end if
8:         let s′.didCache[doc.did] := doc
9:         call CHECK_ID_TOKEN(reference[session], s′)
10:    else if reference[responseTo] ≡ CONFIG then
11:        let config := m.body
12:        if session[issuer] ≢ config[issuer] then
13:            stop
14:        end if
15:        let s′.oidcConfigCache[session[issuer].host] := config
               ↪   → Store configuration.
16:        call START_LOGIN_FLOW(reference[session], s′)
17:    end if
18:    stop
19: end function
```

Algorithm B.7 (Same-Device) Relying Party R^r: Validate an ID token.

```
1: function CHECK_ID_TOKEN(sessionId, a, s′)    → Check the contents of an ID token.
2:     let session := s′.sessions[sessionId]    → Retrieve session data.
3:     let id_token := session[id_token]
4:     let data := extractmsg(id_token).token_data    → Retrieve ID token data.
5:     let issuer := session[issuer]
6:     if data[iss] ≢ issuer then    → Check issuer.
7:         stop
8:     end if
9:     if data[aud] ≢ ⟨⟩ ∨ session[client_id] ∉^⟨⟩ data[aud] then
           ↪    → Check intended audience.
10:        stop
11:    end if
12:    let registration := session[registration]
13:    let sub := data[sub]
14:    if ∃t : sub ≡ ⟨jkt, t⟩ ∧ jkt ∈^⟨⟩ registration[subject_syntax_types_supported] then
15:        let v_key := data[sub_jwk]    → Retrieve public verification key of the token subject.
16:        if sub.2 ≢ hash(v_key) then    → Check sub value is thumbprint of given public key.
17:            stop
18:        end if
19:    else if sub ∈ DID
               ↪ ∧ did ∈^⟨⟩ registration[subject_syntax_types_supported] then
20:        if s′.didCache[sub] ≢ ⟨⟩ then    → Check if DID Document is in cache.
21:            let kid := extractmsg(id_token).kid
22:            let didDoc := s′.didCache[sub]
23:            if kid ∉ DIDURL ∨ kid.did ≢ sub ∨ kid.key ∉ didDoc.verification
                   ↪ then    → Check that kid references a verification method for sub as defined by the
                                DID Document.
24:                stop
25:            end if
26:            let v_key := didDoc.verification[kid.key]
                   ↪    → Retrieve key identified by kid.
27:        else    → Resolve DID Document.
28:            call REQUEST_RESOLVE_DID(sessionId, sub, a, s′)
29:        end if
30:    else
31:        stop
32:    end if
33:    if checksig(id_token, v_key) ≢ ⊤ then    → Check ID token signature.
34:        stop
35:    end if
36:    if data[nonce] ≡ ⟨⟩ ∨ session[nonce] ≢ data[nonce] then    → Check nonce.
37:        stop
38:    end if
39:    if claims ∈ session then    → RP has requested a verifiable presentation.
40:        call CHECK_VERIFIABLE_PRESENTATIONS(sub, sessionId, s′)
41:    end if
42:    call START_SERVICE_SESSION(sub, ⊥, sessionId, s′)
43: end function
```

Algorithm B.8 (Same-Device) Relying Party R^r: Validate a Verifiable Presentation.

function CHECK_VERIFIABLE_PRESENTATIONS(*sub*, *sessionId*, s') → **Verify the Verifiable Presentation against the request.**
  let *session* := s'.sessions[*sessionId*]
  if vp_token $\notin$ *session*[claims] $\vee$ vp_token $\notin$ *session* **then**
    stop
  end if
  let *vp* := *session*[vp_token]
  let *claimType* := *session*[claims][vp_token][type]
  if extractmsg(*vp*)[challenge] $\not\equiv$ *session*[nonce] **then**
    stop
  end if
  if extractmsg(*vp*)[domain] $\equiv \langle\rangle \vee$ extractmsg(*vp*)[domain] $\not\equiv$ *session*[client_id] **then**
    stop
  end if
  let *vc* := extractmsg(*vp*)[vc] → Extract the Verifiable Credential.
  let *vc_data* := extractmsg(*vc*) → Extract the Verifiable Credential content.
  let *v_key* := *vc_data*[subject]
  if *vc_data*[context] $\not\equiv$ VC $\vee$ *vc_data*[issuer] $\notin^{\langle\rangle}$ s'.trustedVCIss
    ↪ $\vee$ *vc_data*[type] $\not\equiv$ *claimType* **then**
    stop
  end if
  if checksig(*vp*, *v_key*) $\not\equiv \top$ **then** → Check the presentation signature.
    stop
  end if
  if checksig(*vc*, *vc_data*[issuer]) $\not\equiv \top$ **then** → Check the credential signature.
    stop
  end if
  call START_SERVICE_SESSION(*sub*, *vc*, *sessionId*, s')
end function

Algorithm B.9 Same-Device Relying Party R^r: Start a service session.

function START_SERVICE_SESSION(*id*, *vc*, *sessionId*, s') → **Start a service session.**
  let s'.sessions[*sessionId*][loggedInAs] := *id*
  let s'.sessions[*sessionId*][serviceSessionId] := $v_{\texttt{service\_session}}$
  if *vc* $\not\equiv \bot$ **then**
    let s'.sessions[*sessionId*][credential] := *vc*
  else
    let s'.sessions[*sessionId*][credential] := $\bot$
  end if
  let *sessionCookie* := $\langle\langle$__Host, serviceSessionId$\rangle, \langle v_{\texttt{service_session}}, \top, \top, \top\rangle\rangle$
  let *headers* := [ReferrerPolicy : origin, Set-Cookie : *sessionCookie*]
  let *request* := s'.sessions[*sessionId*][redirect_ep_req]
  let m' := $\text{enc}_\text{s}(\langle$HTTPResp, *request*[message].nonce, 200, *headers*, ok$\rangle$, *request*[key])
    ↪ → Confirm successful login
  stop $\langle\langle$*request*[sender], *request*[receiver], $m'\rangle\rangle$, s'
end function

Algorithm B.10 (Same-Device) Relying Party R^r: Answer a request for registration metadata.

function SEND_REGISTRATION_METADATA(m, k, a, f, s') → **Process a request for registration metadata.**
  if m.method ≠ GET **then**
    stop
  end if
  let *session* **such that** $sid \in s'$.sessions ∧ s'.sessions[sid] ≡ *session*
    ↪ ∧ *session*[registration_uri].parameters[ref] ≡ m.parameters[ref]
    ↪ **if possible; otherwise stop**
  let *headers* := [ReferrerPolicy : origin]
  let *regObj* := *session*[registration]
  let m' := $\mathsf{enc}_s(\langle \text{HTTPResp}, m.\text{nonce}, 200, headers, regObj \rangle, k)$
  stop $\langle\langle f, a, m' \rangle\rangle, s'$
end function

Algorithm B.11 Same-Device Relying Party R^r: Answer a request for a request object.

function SEND_REQUEST_OBJECT(m, k, a, f, s') → **Process a request for a request object.**
  if m.method ≠ GET **then**
    stop
  end if
  let *session* **such that** $sid \in s'$.sessions ∧ s'.sessions[sid] ≡ *session*
    ↪ ∧ *session*[request_uri] ≠ ⟨⟩
    ↪ ∧ *session*[request_uri].parameters[ref] ≡ m.parameters[ref]
    ↪ **if possible; otherwise stop**
  let *headers* := [ReferrerPolicy : origin]
  let *requestObj* := *session*[requestObj]
  let m' := $\mathsf{enc}_s(\langle \text{HTTPResp}, m.\text{nonce}, 200, headers, requestObj \rangle, k)$
  stop $\langle\langle f, a, m' \rangle\rangle, s'$
end function

Algorithm B.12 (Same-Device) Relying Party R^r: Request DID resolution.

function REQUEST_RESOLVE_DID(*sessionId*, *did*, a, s') → **Request a DID Document.**
  let *resolver* := s'.didResolvers[*did*.method]
  let *params* := [did : *did*]
  let *request* := ⟨HTTPReq, $v_{\text{did\_req}}$, GET, *resolver*, /resolve, *params*, ⟨⟩, ⟨⟩⟩
  let *reference* := [session : *sessionId*, responseTo : DID_RESOLVE, requestedDID : *did*]
  call HTTPS_SIMPLE_SEND(*reference*, *request*, s', a)
end function

Algorithm B.13 Same-Device Relying Party R^r: Prepare the authentication request.

```
1:  function START_LOGIN_FLOW(sessionId, a, s′)    → Start a login flow.
2:      let session := s′.sessions[sessionId]    → Retrieve session.
3:      if session[issuer].host ∉ s′.oidcConfigCache then
            ↪    → Start dynamic discovery.
4:          let host := session[issuer].host
5:          let path := /.well − known/openid − configuration
6:          let request := ⟨HTTPReq, v_discover_req, GET, host, path, [], ⟨⟩, ⟨⟩⟩
7:          let reference := [session : sessionId, responseTo : CONFIG]
8:          call HTTPS_SIMPLE_SEND(reference, request, s′, a)
9:      end if
10:     let oidcConfig := s′.oidcConfigCache[session[issuer].host]
11:     let redirectUri ← {⟨URL, S, d, /redirect_ep, ⟨⟩, ⊥⟩ | d ∈ dom(r)}
          ↪    → Choose redirect URI for some domain.
12:     let data := [response_type : ⟨id_token⟩, redirect_uri : redirectUri,
                ↪  nonce : v_rp_req_nonce]
13:     let syntax_types ← {⟨did⟩, ⟨jkt⟩, ⟨did, jkt⟩}
14:     let registration := [subject_syntax_types_supported : syntax_types]
15:     let registrationByRef ← {⊤, ⊥}    → Choose whether to send registration by reference.
16:     if registrationByRef ≡ ⊤ then
17:         let registrationUris := {⟨URL, S, d, /registration, ⟨⟩, ⊥⟩| d ∈ dom(r)}
18:         let registrationUri ← registrationUris
19:         let registrationUri.parameters := [ref : v_reg_ref]
20:         let data[registration_uri] := registrationUri
21:         let s′.sessions[sessionId][registration] := registration
22:     else
23:         let data[registration] := registration
24:     end if
25:     let requestVP ← {⊤, ⊥}    → Choose whether to request a Verifiable Presentation.
26:     if requestVP then
27:         let vpType ← 𝕊
28:         let data[claims][vp_token] := [input_descriptor : vpType]
29:     end if
30:     let authEndpoint := oidcConfig[authorization_endpoint]
31:     let useRequestObject ← {⊤, ⊥}    → Choose whether to send signed request object.
32:     if useRequestObject then
33:         let client_id ← {⟨URL, S, d, ⟨⟩, ⟨⟩, ⊥⟩| d ∈ dom(r)}
34:     else
35:         let client_id := redirectUri
36:     end if
37:     let data[client_id] := client_id
38:     let s′.sessions[sessionId] := s′.sessions[sessionId] ∪ data
```

39: **if** *useRequestObject* **then**
40: **let** *requestObj* := [aud : *session*[issuer], iss : *client_id*] ∪ *data*
41: **let** *signedRequest* := sig(*requestObj*, *s′*.requestSigKey)
42: **let** *s′*.sessions[*sessionId*][requestObj] := *signedRequest*
43: **let** *requestByRef* ← {⊤, ⊥} → Choose whether to send the request object by reference.
44: **if** *requestByRef* ≡ ⊤ **then**
45: **let** *requestUri* ← {⟨URL, S, *d*, /request_ep, ⟨⟩, ⊥⟩ | *d* ∈ dom(*r*)}
46: **let** *requestUri*.parameters := [ref : $v_{\texttt{req\_ref}}$]
47: **let** *s′*.sessions[*sessionId*][request_uri] := *requestUri*
48: **let** *authEndpoint*.parameters := *authEndpoint*.parameters
↪ ∪ [request_uri : *requestUri*]
↪ → Add reference to signed request.
49: **else**
50: **let** *authEndpoint*.parameters := *authEndpoint*.parameters
↪ ∪ [request : *signedRequest*]
↪ → Add signed request.
51: **end if**
52: **let** *authEndpoint*.parameters := *authEndpoint*.parameters
↪ ∪ [response_type : ⟨id_token⟩, client_id : *data*[client_id],
↪ nonce : *data*[nonce]]
↪ → Add required request parameters directly aswell.
53: **else**
54: **let** *authEndpoint*.parameters := *authEndpoint*.parameters ∪ *data*
↪ → Add request parameters.
55: **end if**
56: **let** *headers* := [Location : *authEndpoint*, ReferrerPolicy : origin]
57: **let** *headers*[Set-Cookie] := [⟨__Host, sessionId⟩ : ⟨*sessionId*, ⊤, ⊤, ⊤⟩]
58: **let** *request* := *s′*.sessions[*sessionId*][startRequest]
59: **let** *m′* := enc_{s}(⟨HTTPResp, *request*[message].nonce, 303, *headers*, ⊥⟩, *request*[key])
60: **stop** ⟨⟨*request*[sender], *request*[receiver], *m′*⟩⟩, *s′*
61: **end function**

B.4 Configuration Provider

A Configuration Provider $c \in \mathsf{CP}$ is a web server modeled as an atomic process (I^c, Z^c, R^c, s_0^c) with addresses $I^c := \mathsf{addr}(c)$.

Definition B.4.1
A *state* $s \in Z^c$ *of an Configuration Provider* c is a term of the form $\langle$*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*$\rangle$

with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ and $tlskeys \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (components as in the generic HTTPS server model).

An *initial state* s_0^c *of* c is a state of c with $s_0^c.\mathtt{DNSAddress} \equiv dnsAddress^{\mathsf{Net}}$, $s_0^c.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^c.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^c.\mathtt{corrupt} \equiv \bot$, $s_0^c.\mathtt{keyMapping} \equiv \langle\mathsf{keyMapping}\rangle$, $s_0^c.\mathtt{tlskeys} \equiv \langle\mathsf{tlskeys}^c\rangle$. ◇

In the following, we specify the relation R^c of the Configuration Provider. The model is an extension of the generic HTTPS server model and thus we only specify the algorithm that differs from this model. This algorithm is given in Algorithm B.14.

B.5 Same-Device Self-Issued OpenID Provider

A Same-Device Self-Issued OP $i \in \mathsf{SIOP}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with addresses $I^i := \mathsf{addr}(i)$.

In slight abuse of the generic HTTPS server model we define $\mathsf{privLocalKeys}^i := \{\langle d, k\rangle \mid d \in \mathsf{Doms}(i), k = \mathsf{localkey}(i)\}$. We use this instead of the *tlskeys*, the generic HTTPS server model expects, to accept local calls instead of TLS encrypted ones, in slight abuse of the model.

Algorithm B.14 Configuration Provider R^c: Process an HTTPS request.

```
1:  function PROCESS_HTTPS_REQUEST(m, k, a, f, s')    → Process an HTTPS request.
2:      if m.path ≡ /.well − known/openid − configuration then
3:          if m.method ≠ GET then
4:              stop
5:          end if
6:          let issuer := ⟨URL, S, m.host, ⟨⟩, ⟨⟩, ⊥⟩
7:          let auth_ep ← {⟨URL, S, d, /auth, ⟨⟩, ⊥⟩ | d ∈ dom(c)}
8:          let metaData := [issuer : issuer, authorization_endpoint : auth_ep]
9:          let syntax_types ← {⟨did⟩, ⟨jkt⟩, ⟨jkt, did⟩}
10:         let metaData[subject_syntax_types_supported] := ⟨syntax_types⟩
11:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, metaData⟩, k)
12:         stop ⟨⟨f, a, m'⟩⟩, s'
13:     end if
14:     stop
15: end function
```

Algorithm B.15 Same-Device Self-Issued IdP R^i: Process a HTTPS request.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s′)    → Process an incoming HTTPS request.
2:     if m.path ≡ /auth then    → Authorization Endpoint.
3:         if m.method ≡ GET then
4:             let data := m.parameters
5:         else if m.method ≡ POST then
6:             let data := m.body
7:         end if
8:         if nonce ∉ data then
9:             stop
10:        end if
11:        let sessionId := v_id_request
12:        let s′.sessions[sessionId] := [startRequest : [message : m, key : k,
               ↪ receiver : a, sender : f], issuer : ⟨URL, S, m.host, ⟨⟩, ⟨⟩, ⊥⟩]
13:        if request_uri ∈ data ∧ request ∉ data then
               ↪ → Signed request sent by reference.
14:            let request_uri := data[request_uri]
15:            let request := ⟨HTTPReq, v_req_obj_req, GET, request_uri.host, request_uri.path,
                   ↪ request_uri.parameters, ⟨⟩, ⟨⟩⟩
16:            let reference := [session : sessionId, responseTo : RP_REQUEST_OBJ]
17:            call HTTPS_SIMPLE_SEND(reference, request, s′, a)
18:        else if request_uri ∉ data ∧ request ∈ data then
               ↪ → Request is sent as a signed object.
19:            let reqData := extractmsg(data[request])
20:            if reqData[aud] ≢ s′.sessions[sessionId][issuer] then
21:                stop
22:            end if
23:            let clientId := reqData[client_id]
24:            if clientId ∉ URLs ∨ clientId.protocol ≢ S ∨ redirect_uri ∉ reqData then
25:                stop
26:            end if
27:            let c_k such that c_k ≡ s′.clientKeys[clientId.host] if possible; otherwise stop
28:            if checksig(data[request], c_k) ≢ ⊤ then    → Check request object signature.
29:                stop
30:            end if
31:            let unsigned such that unsigned[param] ≡ data[param]
                   ↪ ⟺ (param ∈ data ∧ param ∉ reqData) if possible; otherwise stop
32:            let data := reqData ∪ unsigned    → Signed request data supersedes request parameters.
33:        else if request_uri ∉ data ∧ request ∉ data then    → Unsigned request.
34:            if redirect_uri ∉ data ∨ data[redirect_uri] ≢ data[client_id] then
35:                stop
36:            end if
37:        else
38:            stop
39:        end if
40:        let s′.sessions[sessionId][params] := data    → Store request data.
41:        call SEND_TOKEN(sessionId, a, s′)
42:    end if
43: end function
```

Algorithm B.16 Same-Device Self-Issued OP R^i: Process an HTTPS response.

```
1: function PROCESS_HTTPS_RESPONSE(m, reference, request, a, f, s′)   → Process an HTTPS response.
2:     let sessionId := reference[session]
3:     if reference[responseTo] ≡ RP_REGISTRATION then   → Process registration.
4:         let registration := m.body   → Retrieve registration.
5:         let s′.sessions[sessionId][registration] := registration
6:         call SEND_TOKEN(sessionId, a, s′)   → Continue processing of request.
7:     else if reference[responseTo] ≡ RP_REQUEST_OBJ then   → Process request object.
8:         let startMessage := s′.sessions[sessionId][startRequest][message]
9:         if startMessage.method ≡ GET then
10:            let unsignedData := startMessage.parameters
11:        else if startMessage.method ≡ POST then
12:            let unsignedData := startMessage.body
13:        end if
14:        let requestObj := m.body   → Retrieve signed request.
15:        let reqData := extractmsg(requestObj)
16:        if reqData[aud] ≠ s′.sessions[sessionId][issuer] then
17:            stop
18:        end if
19:        let clientId := reqData[client_id]
20:        if clientId ∉ URLs ∨ clientId.protocol ≠ S ∨ redirect_uri ∉ reqData then
21:            stop
22:        end if
23:        let cKey such that cKey ≡ s′.clientKeys[clientId.host] if possible; otherwise stop
24:        if checksig(requestObj, cKey) ≠ ⊤ then   → Check request signature.
25:            stop
26:        end if
27:        let unsigned such that unsigned[param] ≡ unsignedData[param]
               ↪ ⟺ (param ∈ unsignedData ∧ param ∉ data) if possible; otherwise stop
28:        let data := reqData ∪ unsigned   → Signed request object data supersedes request parameters.
29:        let s′.sessions[sessionId][params] := data
30:        call SEND_TOKEN(sessionId, a, s′)   → Continue processing of request.
31:    end if
32:    stop
33: end function
```

Definition B.5.1

A *state* $s \in Z^i$ *of a Same-Device Self-Issued OP* i is a term of the form $\langle$*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*, *sessions*, *identities*, *credentials*, *clientKeys*, *browserEncKey*$\rangle$ with *DNSaddress* $\in$ IPs, *pendingDNS* $\in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *pendingRequests* $\in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *corrupt* $\in \mathcal{T}_\mathcal{N}$, *keyMapping* $\in [\mathsf{Doms} \times \mathcal{T}_\mathcal{N}]$, *tlskeys* $\in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (components as in the generic HTTPS server model), *sessions* $\in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *identities* a sequence of $[\mathbb{S} \times \mathcal{T}_\mathcal{N}]$, *credentials* $\in \mathcal{T}_\mathcal{N}$, *clientKeys* $\in [\mathsf{Doms} \times \mathcal{T}_\mathcal{N}]$ and *browserEncKey* $\in K_{\mathrm{local}}$.

An *initial state* s_0^i *of* i is a state of i with s_0^i.DNSAddress $\equiv$ *dnsAddress*$^{\mathsf{Net}}$, s_0^i.pendingDNS $\equiv \langle\rangle$, s_0^i.pendingRequests $\equiv \langle\rangle$, s_0^i.corrupt $\equiv \bot$, s_0^i.keyMapping $\equiv \langle$keyMapping$\rangle$, s_0^i.tlskeys $\equiv \langle$privLocalKeys$^i\rangle$, s_0^i.sessions $\equiv \langle\rangle$, s_0^i.identities $\equiv \langle$ID_Records$^i\rangle$, s_0^i.clientKeys $\equiv$ RRP^i, s_0^i.credentials $\equiv \langle$VCW$^i\rangle$, s_0^i.browserEncKey $\equiv$ pub(localkey(browser(i))). ◇

Algorithm B.17 Same-Device Self-Issued OP R^i: Send ID Token and Verifiable Presentation.

1: **function** SEND_TOKEN(*sessionId*, *a*, *s′*) → **Send out the requested ID token and if requested a Verifiable Presentation.**
2: **let** *req_data* := *s′*.sessions[*sessionId*][params] → Retrieve request parameters.
3: **if** *req_data*[response_type] ≢ ⟨id_token⟩ **then**
4: **stop**
5: **end if**
6: **if** registration ∈ *s′*.sessions[*sessionId*] **then**
7: **let** *registration* := *s′*.sessions[*sessionId*][registration]
8: **else if** registration ∈ *req_data* **then**
9: **let** *registration* := *req_data*[registration]
10: **else if** registration_uri ∈ *req_data* **then**
11: **let** *registration_uri* := *req_data*[registration_uri]
12: **let** *request* := ⟨HTTPReq, $v_{\texttt{reg\_req}}$, GET, *registration_uri*.host, *registration_uri*.path, ↪ *registration_uri*.parameters, ⟨⟩, ⟨⟩⟩
13: **let** *reference* := [session : *sessionId*, responseTo : RP_REGISTRATION]
14: **call** HTTPS_SIMPLE_SEND(*reference*, *request*, *s′*, *a*)
15: **else**
16: **stop**
17: **end if**
18: **if** *registration*[subject_syntax_types_supported] ≡ ⟨did, jkt⟩ **then**
19: **let** *sub_type* ← {did, jkt}
20: **else**
21: **let** *sub_type* := *registration*[subject_syntax_types_supported]
22: **end if**
23: **if** *sub_type* ≡ did **then**
24: **let** *did_id_records* := {*r* | *r* ∈⟨⟩ *s′*.identities ∧ *r*[id] ∈ DID}
25: **let** *id_record* ← *did_id_records*
26: **else if** *sub_type* ≡ jkt **then**
27: **let** *jkt_id_records* := {*r* | ∃*mid* : *r* ∈⟨⟩ *s′*.identities ∧ *r*[id] ≡ ⟨jkt, *mid*⟩}
28: **let** *id_record* ← *jkt_id_records*
29: **else**
30: **stop**
31: **end if**
32: **let** *token_data*[sub] := *id_record*[id] → Retrieve identity.
33: **let** *token_data*[aud] := ⟨*req_data*[client_id]⟩
34: **let** *token_data*[nonce] := *req_data*[nonce]
35: **let** *token_data*[iss] := *s′*.sessions[*sessionId*][issuer]
36: **let** *sig_key* := *id_record*[key] → Retrieve the signing key for the identity.
37: **if** *sub_type* ≡ jkt **then**
38: **let** *kid* := ⟨⟩
39: **let** *token_data*[sub_jwk] := pub(*sig_key*) → Include the public key.
40: **else**
41: **let** *kid* := *id_record*[kid] → Include reference to verification key in DID Document.
42: **end if**

43: **if** vp_token ∈ *req_data*[claims] **then** → Parameters request Verifiable Presentation.
44: **let** *claimType* := *req_data*[claims][vp_token][input_descriptor]
45: **let** *vc*, *vc_k* **such that** $t \in^{\langle\rangle} s'$.credentials ∧ *t*[vc] ≡ *vc* ∧ *t*[key] ≡ *vc_k*
↪ ∧ *d* ≡ extractmsg(*vc*) ∧ *d*[type] ≡ *claimType* **if possible; otherwise stop**
↪ → Retrieve a suitable Verifiable Credential.
46: **let** *vp_data* := [vc : *vc*, challenge : *req_data*[nonce], domain : *token_data*[aud]]
↪ → Prepare the Verifiable Presentation.
47: **let** *vp* := sig(*vp_data*, *vc_k*)
48: **let** *responseData*[vp_token] := *vp*
49: **end if**
50: **let** *id_token* := sig(⟨*kid*, *token_data*⟩, *sig_key*)
51: **let** *responseData*[id_token] := *id_token*
52: **let** *request* := *s*′.sessions[*sessionId*][startRequest]
53: **let** *redirect_uri* := *req_data*[redirect_uri]
54: **let** *redirect_uri*.fragment := *redirect_uri*.fragment ∪ *responseData*
55: **let** *redirect_uri* := enc_a(*redirect_uri*, *s*′.browserEncKey)
56: **let** *m*′ := enc_s(⟨HTTPResp, *request*[message].nonce, 303,
↪ ⟨⟨Location, *redirect_uri*⟩ , ⟨LocalResponse, ⊤⟩ ⟩, ⟨⟩⟩, *request*[key])
57: **stop** ⟨⟨*request*[sender], *request*[receiver], *m*′⟩⟩, *s*′
58: **end function**

As for RPs, the Self-Issued OP relation R^i is based on the model of generic HTTPS servers, and we only specify algorithms that differ or do not exist in the generic model. These algorithms are given in Algorithms B.15 to B.17. The following list shows the placeholders we use in the Self-Issued OP relation:

- $\nu_{\texttt{id\_request}}$: Session ID for new requests, key in *sessions* (Algorithm B.15)
- $\nu_{\texttt{req\_obj\_req}}$: Request nonce for resolving a request by reference (Algorithm B.15)
- $\nu_{\texttt{reg\_req}}$: Request nonce for resolving a registration request (Algorithm B.17)

B.6 Verifiable Data Registry

A Verifiable Data Registry $d \in \mathsf{VDR}$ is a web server modeled as an atomic process (I^d, Z^d, R^d, s_0^d) with addresses $I^d := \mathsf{addr}(d)$.

Definition B.6.1
A *state* $s \in Z^d$ *of a Verifiable Data Registry* d is a term of the form ⟨*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*, *method*, *docs*⟩ with *DNSaddress* ∈ IPs, *pendingDNS* ∈ $[\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *pendingRequests* ∈ $[\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *corrupt* ∈ $\mathcal{T}_\mathcal{N}$, *keyMapping* ∈ $[\mathsf{Doms} \times \mathcal{T}_\mathcal{N}]$, *tlskeys* ∈ $[\mathsf{Doms} \times K_{\text{TLS}}]$ (components as in the generic HTTPS server model), *method* ∈ $\mathbb{S}$ and *docs* a sequence of DID Documents.

An *initial state* s_0^d *of* d is a state of d with s_0^d.DNSAddress ≡ $dnsAddress^{\mathsf{Net}}$, s_0^d.pendingDNS ≡ ⟨⟩, s_0^d.pendingRequests ≡ ⟨⟩, s_0^d.corrupt ≡ ⊥, s_0^d.keyMapping ≡ ⟨keyMapping⟩, s_0^d.tlskeys ≡ ⟨$\mathsf{tlskeys}^d$⟩, s_0^d.method ≡ $\mathsf{method\_gov}^{-1}(d)$ and s_0^d.docs ≡ ⟨DIDDoc^d⟩. ◇

Algorithm B.18 Verifiable Data Registry R^d: Process an HTTPS request.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s′)   → Process an HTTPS request.
2:     if m.path ≡ /resolve ∧ m.method ≢ GET then   → DID resolution.
3:         let did := m.parameters[did]
4:         if did ∉ DID then   → Only DIDs can be resolved.
5:             stop
6:         end if
7:         if did.method ≢ s′.method then   → Check requested method.
8:             stop
9:         end if
10:        let doc such that doc.did ≡ did ∧ doc ∈^⟨⟩ s′.docs if possible; otherwise stop
11:        let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, doc⟩, k)
12:        stop ⟨f, a, m′⟩, s′
13:    else
14:        stop
15:    end if
16: end function
```

In the following, we specify the relation R^d of the VDR. Our model extends the generic HTTPS server model, thus we only specify the algorithm that differs from this model. We give this algorithm as Algorithm B.18.

C Formal Model of Cross-Device Self-Issued OpenID Providers

In this section we define the formal model of our attack mitigation for Cross-Device Self-Issued OpenID Providers. The model in this section uses the browser and the generic HTTPS server model as defined in [10]. Note that several parts of the model code in this section are the same as in algorithms with the same name in the model of Same-Device Self-Issued OP in Appendix B. For readability, we highlight these equal parts in the code in gray.

C.1 Web System

We model the extension of OpenID Connect for Cross-Device Self-Issued OPs as a web system: A Cross-Device Self-Issued OpenID Provider web system with a network attacker is a web system $\chi\mathcal{SIOID}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ of the form as follows.

The system $\mathcal{W}$ consists of an network attacker in Net, a finite set of browsers xB, a finite set xCP of Configuration Provider web servers, a finite set xRP of Cross-Device Relying Party web servers, a finite set xStub of Cross-Device Stubs, a finite set xSIOP of Cross-Device Self-Issued OPs and a finite set VDR of Verifiable Data Registry web servers. It is $\mathcal{W} = \mathsf{Net} \cup \mathsf{Hon}$ with $\mathsf{Hon} := \mathsf{xB} \cup \mathsf{xRP} \cup \mathsf{xStub} \cup \mathsf{xCP} \cup \mathsf{xSIOP} \cup \mathsf{VDR}$.

The set of scripts $\mathcal{S}$ and the mapping script we define as follows:

- $R^{\text{att}} \in \mathcal{S}$ with $\mathsf{script}(R^{\text{att}})$ = `att_script` (Attacker Script),
- *script_rp_index* $\in \mathcal{S}$ mapped by script to `script_rp_index` (Script simulating the index page of the RP),
- *script_stub_token* $\in \mathcal{S}$ mapped by script to `script_stub_token` (Script simulating the user entering the secret stub token at the Cross Device Stub) and
- *script_render_qr* $\in \mathcal{S}$ mapped by script to `script_render_qr` (Script that models displaying a QR code).

As usual for the WIM, we initiate E^0 with an infinite (mathematical) sequence of trigger events $\langle a, a, \texttt{TRIGGER}\rangle$ for each $a \in \mathsf{IPs}$. Note that we reuse for this model use the mappings that are defined in Appendices B.1.3 to B.1.5. The definition for VDR processes is the same as for the same-device model (Appendix B.6) and we define the remaining processes in Appendices C.2 to C.6.

We define the addresses IPs and domains Doms for the Cross-Device Self-Issued OpenID Provider web system analogously to the Same-Device model with additionally a finite set of addresses and domains for Stubs in xStub. Again configProvider : xSIOP → xCP assigns each Self-Issued OP a Configuration provider, the domains of each $i \in$ xSIOP are those of configProvider(i) and dom assigns each atomic DY process its subset of domains and addr each process its addresses.

We again partition the set $\mathcal{N}$ into sets

$$\mathcal{N} := N \dot\cup K_{\text{TLS}} \dot\cup K_{\text{sign}} \dot\cup K_{\text{QR}} \dot\cup \mathsf{StubToken}$$

with N an infinite set and finite sets $K_{\text{TLS}}, K_{\text{QR}}$, StubToken and K_{sign}. The sets N and K_{TLS} are analogously used as for the Same-Device model, as well as tlskey, tlskeys^y for each atomic DY process and keyMapping. The nonces in K_{sign} are secret signing keys and again partitioned as $K_{\text{sign}} := K^{\mathsf{ID}}_{\text{sign}} \dot\cup K^{\mathsf{VC}}_{\text{sign}} \dot\cup K^{\mathsf{VP}}_{\text{sign}}$. The sets $K^{\mathsf{ID}}_{\text{sign}}$, $K^{\mathsf{VC}}_{\text{sign}}$ and $K^{\mathsf{VP}}_{\text{sign}}$ are as in the Same-Device Model.

To achieve confidentiality of QR code communication we assign to each Self-Issued OP $i \in$ xSIOP a key pair with private key in K_{QR}. For this let qrkey : xSIOP → K_{QR} be an injective mapping that assigns a unique private key for receiving QR codes to each Self-Issued OP.

The set StubToken is used as given in Appendix C.1.1.

Analogously to the Same-Device Self-Issued OP model we define the network attacker *att* in Net with $I^{att} =$ IPs. The initial state of the attacker for the Cross-Device model we define as

$$s_0^{att} = \langle\langle \mathsf{tlskeys}^{att}\rangle, \langle \mathsf{keyMapping}\rangle, \langle \mathsf{ID\_Records}^{att}\rangle, \langle \mathsf{VCW}^{att}\rangle, \mathit{pubSigKeys}, \mathit{pubQRKeys}\rangle$$

where $\mathit{pubSigKeys} \equiv \langle\{\mathsf{pub}(k) \mid k \in K_{\text{sign}}\}\rangle$ and $\mathit{pubQRKeys} \equiv \langle\{\mathsf{pub}(k) \mid k \in K_{\text{QR}}\}\rangle$ (instead of allowing the attacker to make local calls we allow injecting QR codes). Again with $\mathit{dnsAddress}^{\mathsf{Net}}$ we denote an address of the network attacker to be used for DNS requests.

We define corruption analogously as in same-device model in Appendix B.1.

C.1.1 Stub Tokens

We partition the set of stub tokens StubToken into disjoint subsets StubToken^i for each Cross-Device Self-Issued OP $i \in$ xSIOP.

Let stub : xCP → xStub be a mapping that associates each Configuration Provider with a Cross-Device Stub. We also associate each Self-Issued OP i with a trusted Cross-Device Stub with a mapping defined as trustedStub(i) := stub(configProvider(i)).

We associate each browser with a Self-Issued OP i (that represents the user of the browser that might scan a QR code using the Self-Issued OP application i). For this, let user : xB → xSIOP be a mapping. Now we define the mapping userBrowser$(i) := \{b \in \mathsf{xB} \mid \mathsf{user}(b) = i\}$ that maps $i \in$ xSIOP to a subset of browsers from xB. For each browser in userBrowser(i) the Self-Issued OP generates a Stub Token. Let browserStubToken^i : userBrowser(i) → StubToken^i be an injective mapping for each $i \in$ xSIOP .

We define

$$\mathsf{Secrets}^b := \{\langle\langle d, \mathsf{S}\rangle, \mathit{token}\rangle \mid i = \mathsf{user}(b), u = \mathsf{trustedStub}(i), \\ d \in \mathsf{dom}(u), \mathit{token} \equiv \mathsf{browserStubToken}^i(b)\}$$

for each browser $b \in \mathsf{xB}$, the set of stub tokens that the user may enter at the browser for the respective Cross-Device Stub domain. Note that this models our assumption that the user only enters the stub token at the Cross-Device Stub associated with a Cross-Device Self-Issued OP, at least as long as the browser is honest.

C.2 Browser Extension

In this section we define the extension of the WIM browser we use for Cross-Device Self-Issued OP. Each browser $b \in \mathsf{xB}$ is modeled as an atomic DY process (I^b, Z^b, R^b, s_0^b) with $I^b := \mathsf{addr}(b)$.

For each browser $b \in \mathsf{xB}$ we define

$$\mathsf{QRChannel}^b := \{\langle a, \mathsf{pub}(\mathsf{qrkey}(i))\rangle \mid i = \mathsf{user}(b), a \in \mathsf{addr}(i)\}.$$

Note that we assume that only one Self-Issued OP of the user can retrieve QR codes from the (honest) browser.

Definition C.2.1
A *state* $s \in Z^b$ *of a browser* b is a term of the form $\langle$*windows*, *ids*, *secrets*, *cookies*, *localStorage*, *sessionStorage*, *keyMapping*, *sts*, *DNSaddress*, *pendingDNS*, *pendingRequests*, *isCorrupted*, *qrchannels*$\rangle$ with the terms *windows*, *ids*, *secrets*, *cookies*, *localStorage*, *sessionStorage*, *keyMapping*, *sts*, *DNSaddress*, *pendingDNS*, *pendingRequests* and *isCorrupted* as in the original WIM browser definition, and $qrchannels \in \mathcal{T}_{\mathcal{N}}$.

An *initial state* s_0^b *of* b is a state of b with $s_0^b.\mathtt{windows} \equiv \langle\rangle$, $s_0^b.\mathtt{ids} \equiv \langle\rangle$, $s_0^b.\mathtt{secrets} \equiv \langle\mathsf{Secrets}^b\rangle$, $s_0^b.\mathtt{cookies} \equiv \langle\rangle$, $s_0^b.\mathtt{localStorage} \equiv \langle\rangle$, $s_0^b.\mathtt{sessionStorage} \equiv \langle\rangle$, $s_0^b.\mathtt{keyMapping} \equiv \langle\mathsf{keyMapping}\rangle$, $s_0^b.\mathtt{sts} \equiv \langle\rangle$, $s_0^b.\mathtt{DNSaddress} \equiv dnsAddress^{\mathsf{Net}}$, $s_0^b.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^b.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^b.\mathtt{isCorrupted} \equiv \bot$, $s_0^b.\mathtt{qrchannels} \equiv \langle\mathsf{QRChannel}^b\rangle$. ◇

The relation R^b is defined as in the WIM browser model from [10] with an added command for QR codes. The adjusted part is marked in blue.

Algorithm C.1 Web Browser Model Extension: Execute a script.

1: **function** RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
2: **let** *tree* := Clean(s', $s'.\overline{d}$)
3: **let** *cookies* := $\langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
↪ $\wedge\, c.\texttt{content.httpOnly} = \bot$
↪ $\wedge\left(c.\texttt{content.secure} \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S}\right)\right)\}\rangle$
4: **let** *tlw* ← $s'.\texttt{windows}$ **such that** *tlw* is the top-level window containing $\overline{d}$
5: **let** *sessionStorage* := $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, \mathit{tlw}.\texttt{nonce}\rangle\right]$
6: **let** *localStorage* := $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
7: **let** *secrets* := $s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
8: **let** R **such that** $R = \text{script}^{-1}(s'.\overline{d}.\texttt{script})$ **if possible; otherwise stop**
9: **let** *in* := $\langle \mathit{tree}, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}, s'.\overline{d}.\texttt{scriptinputs}, \mathit{cookies},$
↪ $\mathit{localStorage}, \mathit{sessionStorage}, s'.\texttt{ids}, \mathit{secrets}\rangle$
10: **let** $\mathit{state}' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $\mathit{cookies}' \leftarrow \text{Cookies}^{\mathcal{V}}$, $\mathit{localStorage}' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
↪ $\mathit{sessionStorage}' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $\mathit{command} \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
↪ $\mathit{out} := \langle \mathit{state}', \mathit{cookies}', \mathit{localStorage}', \mathit{sessionStorage}', \mathit{command}\rangle$
↪ **such that** $\mathit{out} = \mathit{out}^{\lambda}[v_{10}/\lambda_1, v_{11}/\lambda_2, \dots]$ **with** $(\mathit{in}, \mathit{out}^{\lambda}) \in R$
11: **let** $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right] :=$
↪ $\langle \text{CookieMerge}(s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right], \mathit{cookies}', s'.\overline{d}.\texttt{origin.protocol})\rangle$
12: **let** $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right] := \mathit{localStorage}'$
13: **let** $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, \mathit{tlw}.\texttt{nonce}\rangle\right] := \mathit{sessionStorage}'$
14: **let** $s'.\overline{d}.\texttt{scriptstate} := \mathit{state}'$
15: **let** *referrer* := $s'.\overline{d}.\texttt{location}$
16: **let** *referrerPolicy* := $s'.\overline{d}.\texttt{headers[ReferrerPolicy]}$
17: **let** *docorigin* := $s'.\overline{d}.\texttt{origin}$
18: **switch** *command* **do**
19: **case** $\langle \texttt{HREF}, \mathit{url}, \mathit{hrefwindow}, \mathit{noreferrer}\rangle$
20: **let** $\overline{w}'$:= GETNAVIGABLEWINDOW($\overline{w}$, *hrefwindow*, *noreferrer*, s')
21: **let** *req* := $\langle \texttt{HTTPReq}, v_4, \texttt{GET}, \mathit{url}.\texttt{host}, \mathit{url}.\texttt{path}, \langle\rangle, \mathit{url}.\texttt{parameters}, \langle\rangle\rangle$
22: **if** *noreferrer* $\equiv \top$ **then**
23: **let** *referrerPolicy* := noreferrer
24: **end if**
25: **let** s' := CANCELNAV($s'.\overline{w}'.\texttt{nonce}, s'$)
26: **call** HTTP_SEND($s'.\overline{w}'.\texttt{nonce}$, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, s')
27: **case** $\langle \texttt{IFRAME}, \mathit{url}, \mathit{window}\rangle$
28: **if** *window* $\equiv$ _SELF **then**
29: **let** $\overline{w}' := \overline{w}$
30: **else**
31: **let** $\overline{w}'$:= GETWINDOW($\overline{w}$, *window*, s')
32: **end if**
33: **let** *req* := $\langle \texttt{HTTPReq}, v_4, \texttt{GET}, \mathit{url}.\texttt{host}, \mathit{url}.\texttt{path}, \langle\rangle, \mathit{url}.\texttt{parameters}, \langle\rangle\rangle$
34: **let** $w' := \langle v_5, \langle\rangle, \bot\rangle$
35: **let** $s'.\overline{w}'.\texttt{activedocument.subwindows}$
↪ $:= s'.\overline{w}'.\texttt{activedocument.subwindows} +^{\langle\rangle} w'$
36: **call** HTTP_SEND(v_5, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, s')

```
37:         case ⟨FORM, url, method, data, hrefwindow⟩
38:             if method ∉ {GET, POST} then
39:                 stop
40:             end if
41:             let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s′)
42:             if method = GET then
43:                 let body := ⟨⟩
44:                 let parameters := data
45:                 let origin := ⊥
46:             else
47:                 let body := data
48:                 let parameters := url.parameters
49:                 let origin := docorigin
50:             end if
51:             let req := ⟨HTTPReq, v4, method, url.host, url.path, ⟨⟩, parameters, body⟩
52:             let s′ := CANCELNAV(s′.w̄′.nonce, s′)
53:             call HTTP_SEND(s′.w̄′.nonce, req, url, origin, referrer, referrerPolicy, s′)
54:         case ⟨SETSCRIPT, window, script⟩
55:             let w̄′ := GETWINDOW(w̄, window, s′)
56:             let s′.w̄′.activedocument.script := script
57:             stop ⟨⟩, s′
58:         case ⟨SETSCRIPTSTATE, window, scriptstate⟩
59:             let w̄′ := GETWINDOW(w̄, window, s′)
60:             let s′.w̄′.activedocument.scriptstate := scriptstate
61:             stop ⟨⟩, s′
62:         case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
63:             if method ∈ {CONNECT, TRACE, TRACK} ∧ xhrreference ∉ {N, ⊥} then
64:                 stop
65:             end if
66:             if url.host ≢ docorigin.host ∨ url ≢ docorigin.protocol then
67:                 stop
68:             end if
69:             if method ∈ {GET, HEAD} then
70:                 let data := ⟨⟩
71:                 let origin := ⊥
72:             else
73:                 let origin := docorigin
74:             end if
75:             let req := ⟨HTTPReq, v4, method, url.host, url.path, url.parameters, ⟨⟩, data⟩
76:             call HTTP_SEND(⟨s′.d.nonce, xhrreference⟩, req, url, origin, referrer, referrerPolicy, s′)
77:         case ⟨BACK, window⟩
78:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
79:             NAVBACK(w̄, s′)
80:             stop ⟨⟩, s′
81:         case ⟨FORWARD, window⟩
82:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
83:             NAVFORWARD(w̄, s′)
84:             stop ⟨⟩, s′
```

```
85:       case ⟨CLOSE, window⟩
86:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
87:         remove s′.w̄′ from the sequence containing it
88:         stop ⟨⟩, s′
89:       case ⟨POSTMESSAGE, window, message, origin⟩
90:         let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
91:         if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
              ↪ ∧(origin ≢ ⊥ ⟹ s′.w̄′.documents.j̄.origin ≡ origin) then
92:           let s′.w̄′.documents.j̄.scriptinputs
                ↪ := s′.w̄′.documents.j̄.scriptinputs
                ↪ +^⟨⟩ ⟨POSTMESSAGE, s′.w̄.nonce, docorign, message⟩
93:         end if
94:         stop ⟨⟩, s′
95:       case ⟨ShowQR, content⟩      → Command for rendering a QR code.
96:         let qr := ⟨QR, content⟩
97:         let ⟨addr, key⟩ such that ⟨addr, key⟩ ∈ s′.qrchannels if possible; otherwise stop
98:         let m′ := enc_a(qr, key)
99:         stop ⟨addr, addr, m′⟩, s′
100:      case else
101:        stop
102: end function
```

C.3 Cross-Device Relying Party

A Cross-Device Relying Party $r \in \mathsf{xRP}$ is a web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r). The addresses are defined as $I^r := \mathsf{addr}(r)$.

The state of a Cross-Device RP is defined the same as for a Same-Device RP (Appendix B.3). Like the Same-Device RP the Cross-Device RP relation is an extension of the generic HTTPS server model with algorithms given in Algorithms C.2 to C.5 as well as Algorithm B.6, Algorithm B.8, Algorithm B.10, Algorithm B.12 and Algorithm B.13 as in the (Same-Device) RP model. The script *script_rp_index* that the Cross-Device RP uses is the same as in the Same-Device RP model (Algorithm B.3).

Aside from to the nonces from Algorithm B.6, Algorithm B.8, Algorithm B.10, Algorithm B.12 and Algorithm B.13, that are the same as for the Same-Device RP, the following nonces are used.

- $\nu_{\mathsf{rp\_login}}$: Session ID for new login flows, key in *sessions*, in Algorithm C.4.
- $\nu_{\mathtt{rp\_req\_nonce}}$: Nonce for the `nonce` parameter in the authentication request in Algorithm C.5.
- $\nu_{\mathtt{reg\_ref}}$: Nonce for relating registration requests from the Self-Issued OP to the associated session in Algorithm C.5.
- $\nu_{\mathtt{service\_session}}$: Service Session ID (for logged in users) in Algorithm C.3.

Algorithm C.2 Cross-Device Relying Party R^r: Process a sync request.

```
1:  function PROCESS_SYNC(m, k, a, f, s′)    → Answer a sync request.
2:      let sessionId := m.headers[Cookie][⟨__Host, sessionId⟩]
3:      if sessionId ∉ s′.sessions then    → Check if session is known.
4:          stop
5:      end if
6:      let cnonce := m.parameters[cnonce]
7:      if cnonce ≢ s′.sessions[sessionId][nonce] then    → Check if nonce is known in the session.
8:          stop
9:      end if
10:     let stub_fin := m.parameters[stub_fin]    → Retrieve stub fin endpoint.
11:     let stub_fin.parameters[xdev_sync] := ok
12:     let cstub := m.parameters[cstub]
13:     let stub_fin.parameters[cstub] := cstub
14:     let stub_fin.parameters[cnonce] := cnonce
15:     let headers := [Location : stub_fin, ReferrerPolicy : origin]
16:     let m′ := enc_s(⟨HTTPResp, m.nonce, 303, headers, ⟨⟩⟩, k)
17:     stop ⟨f, a, m′⟩, s′
18: end function
```

Algorithm C.3 Cross-Device Relying Party R^r: Start a service session.

```
1:  function START_SERVICE_SESSION(id, vc, sessionId, s′)    → Start a service session.
2:      let s′.sessions[sessionId][loggedInAs] := id
3:      let s′.sessions[sessionId][serviceSessionId] := v_service_session
4:      if vc ≢ ⊥ then
5:          let s′.sessions[sessionId][credential] := vc
6:      else
7:          let s′.sessions[sessionId][credential] := ⊥
8:      end if
9:      stop ⟨⟩, s′
10: end function
```

Algorithm C.4 Cross-Device Relying Party R^r: Process an HTTPS request.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s′)   → Process an HTTPS request.
2:     if m.path ≡ / then   → Serve index page.
3:         let headers := [ReferrerPolicy : origin]
           ↪   → Set Referrer Policy for the index page of the RP
4:         let sessionId := m.headers[Cookie][⟨__Host, sessionId⟩]
5:         if sessionId ∈ s′.sessions ∧ ⟨__Host, serviceSessionId⟩ ∉ m.headers[Cookie]
           ↪   ∧ serviceSessionId ∈ s′.sessions[sessionId] then
6:             let ssid := s′.sessions[sessionId][serviceSessionId]
7:             let cookie := ⟨⟨__Host, serviceSessionId⟩, ⟨ssid, ⊤, ⊤, ⊤⟩⟩
8:             let headers[Set-Cookie] := cookie
               ↪   → Confirm successful login with service session cookie.
9:         end if
10:        let m′ := enc_s(⟨HTTPResp, m.nonce, 200, headers, ⟨script_rp_index, ⟨⟩⟩⟩, k)
11:        stop ⟨⟨f, a, m′⟩⟩, s′   → Send script_rp_index in response.
12:    else if m.path ≡ /startLogin ∧ m.method ≡ POST then   → Serve request to start login.
13:        if m.headers[Origin] ≢ ⟨m.host, S⟩ then   → CSRF protection.
14:            stop
15:        end if
16:        let issuer_id := m.body   → Retrieve the requested issuer identifier.
17:        if issuer_id ∉ URLs ∨ issuer_id.protocol ≢ S ∨ issuer_id.parameters ≢ ⟨⟩
               ↪   ∨ issuer_id.fragment ≢ ⊥ then
18:            stop
19:        end if
20:        let sessionId := v_rp_login   → Choose fresh nonce for session id.
21:        let s′.sessions[sessionId] := [startRequest : [message:m, key : k,
           ↪   receiver : a, sender : f], issuer : issuer_id]
22:        call START_LOGIN_FLOW(sessionId, s′)
23:    else if m.path ≡ /registration then
24:        call SEND_REGISTRATION_METADATA(m, k, a, f, s′)
25:    else if m.path ≡ /redirect_ep then
26:        if m.parameters[xdev_sync] ≡ request then   → Sync request.
27:            call PROCESS_SYNC(m, k, a, f, s′)
28:        else   → Cross-Device Response.
29:            if m.method ≢ POST then
30:                stop
31:            end if
32:            let data := m.body
33:            let sessionId such that idt ≡ data[id_token] ∧ td ≡ extractmsg(idt).token_data
                   ↪   ∧ td[nonce] ≡ s′.sessions[sessionId][nonce]  if possible; otherwise stop
                   ↪   → Retrieve session for nonce in ID token.
34:        end if
35:        let s′.sessions[sessionId][id_token] := data[id_token]
36:        if vp_token ∈ data then
37:            let s′.sessions[sessionId][vp_token] := data[vp_token]
38:        end if
39:        call CHECK_ID_TOKEN(sessionId, a, s′)
40:    end if
41: end function
```

Algorithm C.5 Cross-Device Relying Party R^r: Prepare the authentication request.

```
1: function START_LOGIN_FLOW(sessionId, a, s′)    → Start a login flow.
2:     let session := s′.sessions[sessionId]
3:     if session[issuer].host ∉ s′.oidcConfigCache then
           ↪    → Dynamic Self-Issued OP discovery.
4:         let host := session[issuer].host
5:         let path := /.well − known/openid − configuration
6:         let request := ⟨HTTPReq, v_discover_req, GET, host, path, [], ⟨⟩, ⟨⟩⟩
7:         let reference := [session : sessionId, responseTo : CONFIG]
8:         call HTTPS_SIMPLE_SEND(reference, request, s′, a)
9:     end if
10:    let oidcConfig := s′.oidcConfigCache[session[issuer].host]
11:    let redirectUri ← {⟨URL, S, d, /redirect_ep, ⟨⟩, ⊥⟩ | d ∈ dom(r)}
12:    let data := [response_type : ⟨id_token⟩, redirect_uri : redirectUri,
                ↪  nonce : v_rp_req_nonce]
13:    let syntax_types ← {⟨did⟩, ⟨jkt⟩, ⟨did, jkt⟩}
14:    let registration := [subject_syntax_types_supported : syntax_types]
15:    let registrationByRef ← {⊤, ⊥}    → Choose whether to send registration by value or by reference.
16:    if registrationByRef ≡ ⊤ then
17:        let registrationUris := {⟨URL, S, d, /registration, ⟨⟩, ⊥⟩| d ∈ dom(r)}
18:        let registrationUri ← registrationUris
19:        let registrationUri.parameters := [ref : v_reg_ref]
20:        let data[registration_uri] := registrationUri
21:        let s′.sessions[sessionId][registration] := registration
22:    else
23:        let data[registration] := registration
24:    end if
25:    let requestVP ← {⊤, ⊥}    → Choose whether to request a Verifiable Presentation.
26:    if requestVP then
27:        let vpType ← 𝕊
28:        let data[claims][vp_token] := [input_descriptor : vpType]
29:    end if
30:    let authEndpoint := oidcConfig[authorization_endpoint]
31:    let client_id := redirectUri
32:    let data[client_id] := client_id
33:    let s′.sessions[sessionId] := s′.sessions[sessionId] ∪ data
34:    let authEndpoint.parameters := authEndpoint.parameters ∪ data
35:    let authEndpoint.parameters[response_mode] := post
36:    let stub_ep := oidcConfig[stub_start]
37:    let stub_ep.parameters[qr_request] := authEndpoint
38:    let headers := [Location : stub_ep, ReferrerPolicy : origin]
39:    let headers[Set-Cookie] := ⟨⟨__Host, sessionId⟩, ⟨sessionId, ⊤, ⊤, ⊤⟩⟩⟩
40:    let request := s′.sessions[sessionId][startRequest]
41:    let m′ := enc_s(⟨HTTPResp, request[message].nonce, 303, headers, ⊥⟩, request[key])
42:    stop ⟨⟨request[sender], request[receiver], m′⟩⟩, s′
43: end function
```

Algorithm C.6 Cross-Device Configuration Provider R^c: Process an HTTPS request.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')   → Process an HTTPS request.
2:     if m.path ≡ /.well − known/openid − configuration then
3:         if m.method ≠ GET then
4:             stop
5:         end if
6:         let issuer := ⟨URL, S, m.host, ⟨⟩, ⟨⟩, ⊥⟩
7:         let auth_ep := {⟨URL, S, d, /auth, ⟨⟩, ⊥⟩ | d ∈ dom(c)}
8:         let metaData := [issuer : issuer, authorization_endpoint : auth_ep]
9:         let metaData[stub_start] := ⟨URL, S, s'.stub, /stub/start, ⟨⟩, ⊥⟩
10:        let syntax_types ← {⟨did⟩, ⟨jkt⟩, ⟨jkt, did⟩}
11:        let metaData[subject_syntax_types_supported] := ⟨syntax_types⟩
12:        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, metaData⟩, k)
13:        stop ⟨⟨f, a, m'⟩⟩, s'
14:    end if
15:    stop
16: end function
```

C.4 Configuration Provider

A Configuration Provider $c \in \mathsf{xCP}$ is a web server modeled as an atomic process (I^c, Z^c, R^c, s_0^c) with addresses $I^c := \mathsf{addr}(c)$.

Let $stub\_dom^c$ be a domain in $\mathsf{dom}(\mathsf{stub}(c))$.

Definition C.4.1
A *state* $s \in Z^c$ *of an Configuration Provider* c is a term of the form $\langle DNSAddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys\ stub\rangle$ with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $pendingRequests \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $tlskeys \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (components as in the generic HTTPS server model) and $stub \in \mathsf{Doms}$.

An *initial state* s_0^c *of* c is a state of c with $s_0^c.\mathtt{DNSAddress} \equiv dnsAddress^{\mathsf{Net}}$, $s_0^c.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^c.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^c.\mathtt{corrupt} \equiv \bot$, $s_0^c.\mathtt{keyMapping} \equiv \langle \mathsf{keyMapping}\rangle$, $s_0^c.\mathtt{tlskeys} \equiv \langle \mathsf{tlskeys}^c\rangle$ and $s_0^c.\mathtt{stub} \equiv stub\_dom^c$. ◇

In the following, we specify the relation R^c of the Configuration Provider. The model is an extension of the generic HTTPS server model and thus we only specify the algorithm that differs from this model. This algorithm is given in Algorithm C.6.

C.5 Cross-Device Stub

A Cross-Device Stub $u \in \mathsf{xStub}$ is a web server modeled as an atomic DY process (I^u, Z^u, R^u, s_0^u). The addresses are defined as $I^u := \mathsf{addr}(u)$.

Definition C.5.1
A *state* $s \in Z^u$ *of a Cross-Device Stub* u is a term of the form $\langle$*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*, *sessions*$\rangle$ with *DNSaddress* $\in \mathsf{IPs}$, *pendingDNS* $\in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *pendingRequests* $\in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *corrupt* $\in \mathcal{T}_\mathcal{N}$, *keyMapping* $\in [\mathsf{Doms} \times \mathcal{T}_\mathcal{N}]$, *tlskeys* $\in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (components as in the generic HTTPS server model) and *sessions* $\in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$.

An *initial state* s_0^u *of* u is a state of u with $s_0^u.\texttt{DNSAddress} \equiv dnsAddress^{\mathsf{Net}}$, $s_0^u.\texttt{pendingDNS} \equiv \langle\rangle$, $s_0^u.\texttt{pendingRequests} \equiv \langle\rangle$, $s_0^u.\texttt{corrupt} \equiv \bot$, $s_0^u.\texttt{keyMapping} \equiv \langle\mathsf{keyMapping}\rangle$, $s_0^u.\texttt{tlskeys} \equiv \langle\mathsf{tlskeys}^u\rangle$ and $s_0^u.\texttt{sessions} \equiv \langle\rangle$. ◇

The relation for the Cross-Device Stub is defined as the generic HTTPS server model with the differing algorithm defined in Algorithm C.9 where we use the following nonces:

- $\nu_{\texttt{stub\_state}}$: Nonce for CSRF protection and
- $\nu_{\texttt{stub\_session}}$: Nonce for the session identifier.

The Cross-Device Stub also uses the two scripts *script_stub_token_form* (similar to the script *script_idp_form* from [6]) and *script_render_qr* (that may follow a link like *script_rp_index* from [6]). Note that the function GETURL is again defined as in [6].

Algorithm C.7 Relation of *script_render_qr*.

Input: $\langle$*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*$\rangle$
if *scriptstate* $\not\equiv \langle\rangle$ **then** → **Display a QR code.**
  let *content* := *scriptstate*
  let *command* := $\langle$ShowQR, *content*$\rangle$
  let *scriptstate*′ := $\langle\rangle$
  stop $\langle$*scriptstate*′, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$
else → **Follow some link.**
  let *protocol* ← {P, S} → Non-deterministically select protocol (HTTP or HTTPS).
  let *host* ← Doms → Non-det. select host.
  let *path* ← $\mathbb{S}$ → Non-det. select path.
  let *fragment* ← $\mathbb{S}$ → Non-det. select fragment part.
  let *parameters* ← $[\mathbb{S} \times \mathbb{S}]$ → Non-det. select parameters.
  let *url* := $\langle$URL, *protocol*, *host*, *path*, *parameters*, *fragment*$\rangle$ → Assemble URL.
  let *command* := $\langle$HREF, *url*, $\bot$, $\bot\rangle$ → Follow link to the selected URL.
  stop $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$
end if

Algorithm C.8 Relation of *script_stub_token_form*.

Input: $\langle$*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*$\rangle$
let *url* := GETURL(*tree*, *docnonce*)
let *url*′ := $\langle$URL, S, *url*.host, /token, $\langle\rangle$, $\langle\rangle\rangle$
let *formData* := *scriptstate*
let *secret* ← *secrets*
let *formData*[stub_token] := *secret*
let *command* := $\langle$FORM, *url*′, POST, *formData*, $\bot\rangle$
stop $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$

Algorithm C.9 Cross-Device Stub R^u : Process an HTTPS request.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s′)   → Process an HTTPS request.
2:     if m.path ≡ /stub/start then   → Start sync flow.
3:         let req := m.parameters[qr_req]   → Retrieve authentication request.
4:         if req ∉ URLs ∨ req.parameters[redirect_uri] ∉ URLs then
5:             stop
6:         end if
7:         let s′.sessions[ν_stub_session] := [request : req, state : ν_stub_state]
8:         let nonce := req.parameters[nonce]
9:         let redirect_uri := req.parameters[redirect_uri]
10:        let redirect_uri.parameters := redirect_uri.parameters
             ↪ ∪ [cnonce : nonce, cstub : ν_stub_state]
11:        let redirect_uri.parameters[xdev_sync] := request
12:        let stub_fin_ep ← {⟨URL, S, d, /stub/fin, ⟨⟩, ⊥⟩ | d ∈ dom(u)}
13:        let redirect_uri.parameters[stub_fin] := stub_fin_ep
14:        let setcookie := ⟨Set-Cookie, ⟨⟨__Host, sessionId⟩, ⟨ν_stub_session, ⊤, ⊤, ⊤⟩⟩⟩
15:        let m′ := enc_s(⟨HTTPResp, m.nonce, 303, ⟨⟨Location, redirect_uri⟩, setcookie,
             ↪ ⟨ReferrerPolicy, origin⟩⟩, ⟨⟩⟩, k)
16:        stop ⟨f, a, m′⟩, s′
17:    else if m.path ≡ /stub/fin ∧ m.parameters[xdev_sync] ≡ ok then   → End sync flow.
18:        let sid := m.headers[Cookie][⟨__Host, sessionId⟩]
19:        if sid ∉ s′.sessions then
20:            stop
21:        end if
22:        let record := s′.sessions[sid]
23:        let state := m.parameters[cstub]
24:        let cnonce := m.parameters[cnonce]
25:        if state ≢ record[state] ∨ cnonce ≢ record[request].parameters[nonce] then
             ↪ → Check for request correspondence.
26:            stop
27:        end if
28:        let s′.sessions[sid][sync] := ok
29:        let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟨ReferrerPolicy, origin⟩⟩,
             ↪ ⟨script_stub_token_form, ⟨⟩⟩⟩, k)
30:        stop ⟨f, a, m′⟩, s′
31:    else if m.path ≡ /token ∧ m.headers[Origin] ≡ ⟨m.host, S⟩ then
         ↪ → Stub token endpoint.
32:        let sid := m.headers[Cookie][⟨__Host, sessionId⟩]
33:        let record := s′.sessions[sid]
34:        if record[sync] ≢ ok then
35:            stop
36:        end if
37:        let stub_req := record[request]   → Retrieve authentication request.
38:        let token := m.body[stub_token]   → Retrieve stub token.
39:        let stub_req.parameters := stub_req.parameters ∪ [stub_token : token]
40:        let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟨ReferrerPolicy, origin⟩⟩,
             ↪ ⟨script_render_qr, stub_req⟩⟩, k)
41:        stop ⟨f, a, m′⟩, s′
42:    end if
43: end function
```

C.6 Cross-Device Self-Issued OP

A Cross-Device Self-Issued OP $i \in \mathsf{xSIOP}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with addresses $I^i := \mathsf{addr}(i)$.

Definition C.6.1
A *state* $s \in Z^i$ *of a Cross-Device Self-Issued OP* i is a term of the form $\langle$*DNSAddress*, *pendingDNS*, *pendingRequests*, *corrupt*, *keyMapping*, *tlskeys*, *sessions*, *identities*, *credentials*, *qrkey*, *stub_tokens*$\rangle$ with *DNSaddress* $\in \mathsf{IPs}$, *pendingDNS* $\in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, *pendingRequests* $\in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, *corrupt* $\in \mathcal{T}_{\mathcal{N}}$, *keyMapping* $\in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, *tlskeys* $\in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (components as in the generic HTTPS server model), *sessions* $\in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, *identities* a sequence of $[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, *credentials* $\in \mathcal{T}_{\mathcal{N}}$, *qrkey* $\in K_{\mathrm{QR}}$ and *stub_tokens* a sequence of StubToken.

An *initial state* s_0^i *of* i is a state of i with $s_0^i.\texttt{DNSAddress} \equiv dnsAddress^{\mathsf{Net}}$, $s_0^i.\texttt{pendingDNS} \equiv \langle\rangle$, $s_0^i.\texttt{pendingRequests} \equiv \langle\rangle$, $s_0^i.\texttt{corrupt} \equiv \bot$, $s_0^i.\texttt{keyMapping} \equiv \langle\mathsf{keyMapping}\rangle$, $s_0^i.\texttt{tlskeys} \equiv \langle\rangle$, $s_0^i.\texttt{sessions} \equiv \langle\rangle$, $s_0^i.\texttt{identities} \equiv \langle\mathsf{ID\_Records}^i\rangle$, $s_0^i.\texttt{credentials} \equiv \langle\mathsf{VCW}^i\rangle$, $s_0^i.\texttt{qrkey} \equiv \mathsf{qrkey}(i)$ and $s_0^i.\texttt{stub\_tokens} \equiv \langle\mathsf{StubToken}^i\rangle$. ◇

The Cross-Device Self-Issued OP relation R^i is again based on the WIM generic HTTPS server model and we specify the algorithms that differ in Algorithms C.10 to C.13. In these algorithms the Cross-Device Self-Issued OP uses the following placeholders for nonces.

- $\nu_{\texttt{qr\_sess}}$: Session ID nonce, used in Algorithm C.12.
- $\nu_{\texttt{post}}$: Nonce for the request that contains the authentication response, used in Algorithm C.13.

Algorithm C.10 Cross-Device Self-Issued OP R^i: Process an HTTPS response.

```
1: function PROCESS_HTTPS_RESPONSE(m, reference, request, a, f, s')   → Process an HTTPS response.
2:     let sessionId := reference[session]
3:     if reference[responseTo] ≡ RP_REGISTRATION then   → Process RP registration.
4:         let registration := m.body
5:         let s'.sessions[sessionId][registration] := registration
6:         call SEND_TOKEN(sessionId, a, s')   → Continue processing of request.
7:     end if
8:     stop
9: end function
```

Algorithm C.11 Cross-Device Self-Issued OP R^i : Process a non-generic message.

```
1: function PROCESS_OTHER(m, a, f, s')
2:     if dec_a(m, s'.qrkey) ≡ ⟨QR, content⟩ then   → QR code message.
3:         let qr := dec_a(m, s'.qrkey)
4:         call PROCESS_QR(qr.content, a, s')
5:     end if
6:     stop
7: end function
```

Algorithm C.12 Cross-Device Self-Issued OP R^i : Process a QR code.

```
1:  function PROCESS_QR(content, a, s′)    → Process a QR code.
2:      if content ∉ URLs ∨ content.host ∉ dom(i) then
3:          stop
4:      end if
5:      let data := content.parameters
6:      if data[response_mode] ≠ post then    → Check cross-device response mode.
7:          stop
8:      end if
9:      if data[stub_token] ∉^⟨⟩ s′.stub_tokens then    → Check if stub token is valid.
10:         stop
11:     end if
12:     let sessionId := v_qr_sess
13:     let s′.sessions[sessionId] := [issuer : ⟨URL, S, content.host, ⟨⟩, ⟨⟩, ⊥⟩ ]
14:     if data[redirect_uri] ∉ URLs ∨ data[redirect_uri].protocol ≠ S then
15:         stop
16:     end if
17:     if data[redirect_uri] ≠ data[client_id] then
18:         stop
19:     end if
20:     let s′.sessions[sessionId][params] := data
21:     call SEND_TOKEN(sessionId, a, s′)
22: end function
```

Algorithm C.13 Cross-Device Self-Issued OP R^i: Send ID token and Verifiable Presentation.

function SEND_TOKEN(*sessionId*, *a*, *s′*) → **Send ID token and if requested a Verifiable Presentation.**
  let *req_data* := *s′*.sessions[*sessionId*][params]
  if *req_data*[response_type] ≢ ⟨id_token⟩ **then**
    stop
  end if
  if registration ∈ *s′*.sessions[*sessionId*] **then**
    let *registration* := *s′*.sessions[*sessionId*][registration]
  else if registration ∈ *req_data* **then**
    let *registration* := *req_data*[registration]
  else if registration_uri ∈ *req_data* **then**
    let *registration_uri* := *req_data*[registration_uri]
    let *request* := ⟨HTTPReq, $v_{\texttt{reg\_req}}$, GET, *registration_uri*.host, *registration_uri*.path, ↪ *registration_uri*.parameters, ⟨⟩, ⟨⟩⟩
    let *reference* := [session : *sessionId*, responseTo : RP_REGISTRATION]
    call HTTPS_SIMPLE_SEND(*reference*, *request*, *s′*, *a*)
  else
    stop
  end if
  if *registration*[subject_syntax_types_supported] ≡ ⟨did, jkt⟩ **then**
    let *sub_type* ← {did, jkt}
  else
    let *sub_type* := *registration*[subject_syntax_types_supported]
  end if
  if *sub_type* ≡ did **then**
    let *did_id_records* := {*r* | *r* ∈⟨⟩ *s′*.identities ∧ *r*[id] ∈ DID}
    let *id_record* ← *did_id_records*
  else if *sub_type* ≡ jkt **then**
    let *jkt_id_records* := {*r* | ∃*mid* : *r* ∈⟨⟩ *s′*.identities ∧ *r*[id] ≡ ⟨jkt, *mid*⟩}
    let *id_record* ← *jkt_id_records*
  else
    stop
  end if
  let *token_data*[sub] := *id_record*[id]
  let *token_data*[aud] := ⟨*req_data*[client_id]⟩
  let *token_data*[nonce] := *req_data*[nonce]
  let *token_data*[iss] := *s′*.sessions[*sessionId*][issuer]
  let *sig_key* := *id_record*[key]
  if *sub_type* ≡ jkt **then**
    let *kid* := ⟨⟩
    let *token_data*[sub_jwk] := pub(*sig_key*) → Include public key.
  else
    let *kid* := *id_record*[kid] → Include reference to verification key in DID Document.
  end if

```
43:     if vp_token ∈ req_data[claims] then     → Parameters request additional claims.
44:         let claimType := req_data[claims][vp_token][input_descriptor]
45:         let vc, vc_k such that t ∈⟨⟩ s′.credentials ∧ t[vc] ≡ vc ∧ t[key] ≡ vc_k
                ↪ ∧ d ≡ extractmsg(vc) ∧ d[type] ≡ claimType  if possible; otherwise stop
                ↪     → Retrieve a suitable Verifiable Credential.
46:         let vp_data := [vc : vc, challenge : req_data[nonce], domain : token_data[aud]]
                ↪     → Prepare the Verifiable Presentation.
47:         let vp := sig(vp_data, vc_k)
48:         let responseData[vp_token] := vp
49:     end if
50:     let id_token := sig(⟨kid, token_data⟩, sig_key)
51:     let responseData[id_token] := id_token
52:     let redirect_uri := req_data[redirect_uri]
53:     let req := ⟨HTTPReq, v_post, POST, redirect_uri.host, redirect_uri.path,
                ↪ redirect_uri.parameters, ⟨⟩, responseData⟩
54:     call HTTPS_SIMPLE_SEND(⟨⟩, req, s′, a)
55: end function
```

D Formal Security Properties

This section gives the formal definitions of the security properties that the proofs in Appendix E analyze. Note that the definition of the authentication and session integrity property are very similar to [6] due to corresponding semantics of the properties but differ subtly due to the differences of general OpenID Connect and the Self-Issued OpenID Provider specification.

D.1 Same-Device Authentication

Authentication is the core desired property for Self-Issued OPs. Intuitively, authentication means that an attacker is not able to log in at an honest RP under the identity of a user if the browser, the Self-Issued OP holding the user's identity, and if the identity utilizes a DID also the Verifiable Data Registry governing the DID, are honest. First we formalize the notion of a login at a RP as obtaining a service session nonce associated with the identity.

Definition D.1.1 (Service Session)
We say that there is a *service session identified by a nonce service for a Self-Issued Identity id at some RP r* in a configuration (S, E, N) of a run ρ of a Same-Device Self-Issued OpenID Provider web system if and only if there exists some nonce *sid* (a session identifier) such that $S(r).\texttt{sessions}[sid][\texttt{loggedInAs}] \equiv id$ and $S(r).\texttt{sessions}[sid][\texttt{serviceSessionId}] \equiv service$. ◇

Definition D.1.2 (Same-Device Authentication)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. We call $\mathcal{SIOID}^n$ *secure w.r.t. authentication* if and only if for every run ρ of $\mathcal{SIOID}^n$, every configuration in (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $id \in \mathsf{SIID}$ where $\mathsf{id\_holder}(id) = i$ is an honest Self-Issued OP in S and the browser $\mathsf{browser}(i) \in \mathsf{B}$ is honest in S, every service session identified by some nonce *service* for the Self-Issued Identity *id* at RP *r* and, if $id \in \mathsf{DID}$ then the Verifiable Data Registry $\mathsf{govenor}(id)$ is an honest Verifiable Data Registry in S, we have that $service \notin d_\emptyset(S(att))$ where *att* is the network attacker (i.e., the attacker cannot derive the service session nonce *service* from its knowledge). ◇

D.2 Same-Device Session Integrity

The attacker should not be able to forcefully log the user in at an RP. We require that if the user (in their browser) is logged in at an RP, they first expressed the wish to be logged in at the RP via a local Self-Issued OP and chose an identity at that Self-Issued OP.

First we formalize the notion that an user has an authenticated session with the RP for some identity in their browser. Note that to satisfy this definition the identity the user is authenticated as is not necessarily one held by the Self-Issued OP of the user, it might be one of the attacker.

Definition D.2.1 (User is logged in)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For a run ρ of $\mathcal{SIOID}^n$ we say a browser b was authenticated to an RP r under a Self-Issued Identity id in a login session identified by the nonce $lsid$ in a processing step Q in ρ with

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

for some configurations (S, E, N) and (S', E', 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 $m.\texttt{headers}$ contains a header of the form $\langle \texttt{Set-Cookie}, \langle\langle \texttt{\_\_Host}, \texttt{serviceSessionId}\rangle, \langle ssid, \top, \top, \top\rangle\rangle\rangle$ for some nonce $ssid$ and we have that there is a term $session$ such that $S(r).\texttt{sessions}[lsid] \equiv session$, $session[\texttt{serviceSessionId}] \equiv ssid$, and $session[\texttt{loggedInAs}] \equiv id$.

We then write $\mathsf{loggedIn}^{Q}_{\rho}(b, r, id, lsid)$. ◇

Next, we define the notion that the user has started a flow to login at an RP via a local Self-Issued OP.

Definition D.2.2 (User started login flow)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For a run ρ of $\mathcal{SIOID}^n$ we say the browser b started a login session identified by a nonce sid at the RP r via an atomic DY process i in a processing step Q in ρ if

1. in that processing step the browser b received a trigger message, selected a document loaded from an origin of r, executed the script *script_rp_index* (Algorithm B.3) and in that script (1) b chose a domain $d \in \mathsf{dom}(i)$ in Line 5 and (2) executed the Line 8, and
2. r sends an HTTPS response corresponding to the HTTPS request sent by b in Q (in a later processing step in ρ) and in that response, there is a header of the form $\langle \texttt{Set-Cookie}, [\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle sid, \top, \top, \top\rangle]\rangle$.

We then write $\mathsf{started}^{Q}_{\rho}(b, r, i, sid)$. ◇

Definition D.2.3 (User chose identity at Self-Issued OP)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For a run ρ of $\mathcal{SIOID}^n$ we say a user chose an identity id at a Self-Issued OP i for a login session identified by a nonce sid at the RP r if $\mathsf{id\_holder}(id) = i$ and there is a processing step Q in ρ with

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

for some configurations (S, E, N) and (S', E', 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 to i, and we have that $m.\texttt{headers}$ contains a header of the form $\langle \texttt{Location}, \mathsf{enc_a}(redirect\_uri, \mathsf{pub}(\mathsf{localkey}(\mathsf{browser}(i))))\rangle$ with $data \equiv \mathsf{extractmsg}(redirect\_uri.\texttt{fragment}[\texttt{id\_token}]).\texttt{token\_data}$ where

1. $data[\texttt{sub}] \equiv id$,
2. $aud \in^{\langle\rangle} data[\texttt{aud}]$ for a term aud with $aud.\texttt{host} \in \mathsf{dom}(r)$ and $aud.\texttt{protocol} \equiv \texttt{S}$,

3. $data[\texttt{nonce}] \equiv nonce$ for a term $nonce$,

and it is $S(r).\texttt{session}[sid][\texttt{nonce}] \equiv nonce$.

We then write $\mathsf{choseIdentity}^{Q}_{\rho}(r, id, i, sid)$. ◇

Definition D.2.4 (Same-Device Session Integrity)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. We call $\mathcal{SIOID}^n$ *secure w.r.t. session integrity* if and only if for every run ρ of $\mathcal{SIOID}^n$, every processing step Q in ρ with

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

for some configurations (S, E, N) and (S', E', N'), every browser $b \in \mathsf{B}$ that is honest in S, every Relying Party $r \in \mathsf{RP}$ that is honest in S, every $id \in \mathsf{SIID}$, every nonce $lsid$, where $\mathsf{loggedIn}^{Q}_{\rho}(b, r, id, lsid)$ holds, we have that there exists an atomic DY process i such that

1. there exists a processing step Q' in ρ before Q such that $\mathsf{started}^{Q'}_{\rho}(b, r, i, lsid)$, and
2. if i is an honest Self-Issued OP in S with $\mathsf{browser}(i) = b$ and $\mathsf{configProvider}(i)$ is an honest Configuration Provider in S then there exists a processing step Q'' in ρ before Q such that $\mathsf{choseIdentity}^{Q''}_{\rho}(r, id, i, lsid)$ holds. ◇

D.3 Same-Device Holder Binding

Intuitively, the attacker should not be able to convince the RP to associate a Verifiable Credential of an honest user with the attacker or a Verifiable Credential of the attacker with an honest user. We formalize this by demanding, that if a Self-Issued Identity and a Verifiable Credential are accepted into the same service session at an (honest) RP, they must be from the same (honest) holder (Self-Issued OP) acting on behalf of the user, if certain parties are honest.

Definition D.3.1 (Service Session with a Verifiable Credential)
We say that there is a *service session identified by a nonce service with a Verifiable Credential vc at some RP r* in a configuration (S, E, N) of a run ρ of a Same-Device Self-Issued OpenID Provider web system if and only if there exists some nonce sid such that $S(r).\texttt{sessions}[sid][\texttt{credential}] \equiv vc$ and $S(r).\texttt{sessions}[sid][\texttt{serviceSessionId}] \equiv service$. ◇

Definition D.3.2 (Same-Device Holder Binding)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. We call $\mathcal{SIOID}^n$ *secure w.r.t. holder binding* iff for every run ρ of $\mathcal{SIOID}^n$, every configuration in (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $id \in \mathsf{SIID}$, every service session identified by some nonce $service$ for the Self-Issued Identity id at RP r, and if $id \in \mathsf{DID}$ the Verifiable Data Registry $\mathsf{govenor}(id)$ is an honest Verifiable Data Registry in S, every $vc \in \mathsf{VC}$ and every service session identified by the nonce $service$ with the Verifiable Credential vc at RP r if $\mathsf{id\_holder}(id) = i$ is an honest Self-Issued OP in S and the browser $\mathsf{browser}(i)$ is honest in S, or $\mathsf{vc\_holder}(vc) = y$ is an honest Self-Issued OP in S and the browser $\mathsf{browser}(y)$ is honest in S, it holds that $\mathsf{vc\_holder}(vc) = \mathsf{id\_holder}(id)$. ◇

D.4 Cross-Device Authentication

Analogously, to the same-device case we define a service session at an RP and the authentication property for the Cross-Device Self-Issued OpenID Provider web system.

Definition D.4.1 (Cross-Device Service Session)
We say that there is a *service session identified by a nonce service for a Self-Issued Identity id at some Cross-Device RP r* in a configuration (S, E, N) of a run ρ of a Cross-Device Self-Issued OpenID Provider web system if and only if there exists some nonce *sid* (a session id) such that $S(r).\texttt{sessions}[sid][\texttt{loggedInAs}] \equiv id$ and $S(r).\texttt{sessions}[sid][\texttt{serviceSessionId}] \equiv service$. ◇

Definition D.4.2 (Cross-Device Authentication)
Let $\chi SIOI\mathcal{D}^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. We call $\chi SIOI\mathcal{D}^n$ *secure w.r.t. authentication* iff for every run ρ of $\chi SIOI\mathcal{D}^n$, every configuration in (S, E, N) in ρ, every $r \in \mathsf{xRP}$ that is honest in S, every $id \in \mathsf{SIID}$ where $\mathsf{id\_holder}(id) = i$ is an honest Cross-Device Self-Issued OP in S where all $b \in \mathsf{userBrowser}(i)$ are honest browsers in S and $\mathsf{trustedStub}(i)$ is an honest Cross-Device Stub in S, every service session identified by some nonce *service* for the Self-Issued Identity *id* at r and, if $id \in \mathsf{DID}$ then the Verifiable Data Registry $\mathsf{govenor}(id)$ is an honest Verifiable Data Registry in S, we have that $service \notin d_\emptyset(S(att))$ where *att* is the network attacker. ◇

E Proof of Security Properties

This section gives the proofs for the Self-Issued OP security properties defined in Appendix D. Note that the proofs argue about algorithms of the browser model or the generic HTTPS server model from [10] which we refer to as the browser model or the generic HTTPS server model. We also use Lemma 1 of [10] in several places which we refer to as the Browser HTTPS Lemma.

We first prove some general properties for the same-device model (Appendix E.1) that we use to prove authentication (Appendix E.2), session integrity (Appendix E.3) and holder binding (Appendix E.4) for the Self-Issued OP same-device flow. In Appendix E.5, we similarly prove some general properties for our model of the Self-Issued OP cross-device flow variant that we use to prove the authentication property for the cross-device Self-Issued OP model in Appendix E.6. Note that we omit the prefix Same-Device or Cross-Device in the proofs these section when the context clarifies which process definition is meant.

E.1 General Properties of Same-Device Self-Issued OPs

Lemma E.1.1 (Redirection Endpoint Integrity)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Self-Issued Identity $id \in \mathsf{SIID}$ with $\mathsf{id\_holder}(id) = i$ an honest Same-Device Self-Issued OP in S and $\mathsf{browser}(i)$ an honest browser in S, every Same-Device RP r that is honest in S, and every ID Token t with $t \equiv \mathsf{sig}(z, sk)$ for a term z and $sk \in \mathsf{proof\_key}(id)$ and $data \equiv z.\texttt{token\_data}$ with $data[\texttt{aud}].\texttt{host} \in \mathsf{dom}(r)$, $data[\texttt{aud}].\texttt{protocol} \equiv \mathsf{S}$ and $data[\texttt{sub}] \equiv id$, we have that if i sends a message $m' \equiv \mathsf{enc_s}(m, k)$ for a term k an HTTP response message m with $m.\texttt{headers}[\texttt{Location}] \equiv \mathsf{enc_a}(u, pk)$ for a redirection URL u with $u.\texttt{fragment}[\texttt{id\_token}] \equiv t$ and $pk \equiv \mathsf{pub}(key)$ for a nonce $key \in K_{\text{local}}$, it holds that $u.\texttt{protocol} \equiv \mathsf{S}$ and $u.\texttt{host} \in \mathsf{dom}(r)$.

Proof. Note that the only point where a Self-Issued OP i that is honest in S sends a response message m' with URL u in location header as in the Lemma text, is in Line 57 of Algorithm B.17.

We see that the URL u must have been set to

$$redirect\_uri \equiv S(i).\texttt{sessions}[isid][\texttt{params}][\texttt{redirect\_uri}]$$

(Lines 2 and 53, before the response parameters are added in the Line after) for a term $isid$ and thus $u.\texttt{host} \equiv redirect\_uri.\texttt{host}$ and $u.\texttt{protocol} \equiv redirect\_uri.\texttt{protocol}$. The only two points where this value could have been set are Line 40 of Algorithm B.15 and Line 29 of Algorithm B.16.

Let $data \equiv \mathsf{extractmsg}(t).\texttt{token\_data}$ for $t \equiv u.\texttt{fragment}[\texttt{id\_token}]$.

We now consider three cases:

1. If Line 40 of Algorithm B.15 was executed we get that Line 34 was executed and thus

$$\begin{aligned} \mathit{redirect\_uri} &\equiv S(i).\mathtt{sessions}[\mathit{isid}][\mathtt{params}][\mathtt{redirect\_uri}] \\ &\equiv S(i).\mathtt{sessions}[\mathit{isid}][\mathtt{params}][\mathtt{client\_id}], \end{aligned}$$

2. or Line 40 of Algorithm B.15 was executed and Lines 18 to 32 have been executed. In this case i must have processed a message *req* with $\mathtt{request} \in \mathit{req}.\mathtt{parameters}$ and $\mathsf{sig}(\mathit{reqObj}, k') \equiv \mathit{req}.\mathtt{parameters}[\mathtt{request}]$ for some terms *reqObj* and k'.
3. If Line 29 of Algorithm B.16 is executed, i must have received a response message *resp* with a signed request object in $\mathit{resp}.\mathsf{body} \equiv \mathsf{sig}(\mathit{reqObj}', k'')$ for terms reqObj' and k'' (Lines 14 and 24).

Case 1: client_id = redirect_uri. Note that in this case we have that

$$\mathit{redirect\_uri} \equiv S(i).\mathtt{sessions}[\mathit{isid}][\mathtt{params}][\mathtt{client\_id}].$$

We get from Lines 2 and 33 of Algorithm B.17 that must have been executed that $\mathit{data}[\mathsf{aud}] \equiv S(i).\mathtt{sessions}[\mathit{isid}][\mathtt{params}][\mathtt{client\_id}]$. Thus, if we have $\mathit{data}[\mathsf{aud}].\mathtt{host} \in \mathsf{dom}(r)$ and $\mathit{data}[\mathsf{aud}].\mathtt{protocol} \equiv \mathsf{S}$ we get

$$u.\mathtt{host} \equiv \mathit{redirect\_uri}.\mathtt{host} \equiv \mathit{data}[\mathsf{aud}].\mathtt{host} \in \mathsf{dom}(r)$$

and

$$u.\mathtt{protocol} \equiv \mathit{redirect\_uri}.\mathtt{protocol} \equiv \mathit{data}[\mathsf{aud}].\mathtt{protocol} \equiv \mathsf{S}.$$

The lemma holds for this case.

Case 2: Signed request sent by value. We now consider the case where the request was sent as a signed dictionary $\mathit{signedReq} \equiv \mathsf{sig}(\mathit{reqObj}, k')$ in a message *req* as $\mathit{req}.\mathtt{parameters}[\mathtt{request}] \equiv \mathit{signedReq}$.

In this case we have by Lines 19, 23, 24, 32 and 40 of Algorithm B.15 that

$$S(i).\mathtt{sessions}[\mathit{isid}][\mathtt{params}][\mathtt{client\_id}] \equiv \mathit{clientId}$$

with $\mathit{clientId} \in \mathsf{URLs}$, $\mathit{clientId}.\mathtt{protocol} \equiv \mathsf{S}$ and $\mathit{clientId} \equiv \mathit{reqObj}[\mathtt{client\_id}]$. We also get from Lines 27 and 28 that

$$\mathsf{checksig}(\mathit{signedReq}, S(i).\mathtt{clientKeys}[\mathit{clientId}.\mathtt{host}]) \equiv \top$$

and, from Lines 2 and 33 of Algorithm B.17 that

$$\mathit{data}[\mathsf{aud}] \equiv S(i).\mathtt{sessions}[\mathit{isid}][\mathtt{params}][\mathtt{client\_id}] \equiv \mathit{clientId}.$$

For $\mathit{redirect\_uri} \equiv S(i).\mathtt{sessions}[\mathit{isid}][\mathtt{params}][\mathtt{redirect\_uri}]$ (Lines 2 and 53) we get from Lines 24, 32 and 40 of Algorithm B.15 that $\mathit{redirect\_uri} \equiv \mathit{reqObj}[\mathtt{redirect\_uri}]$.

Note that i retrieves the key for verifying the signature $cKey$ in Line 27 of Algorithm B.15 such that $cKey \equiv S(i).\texttt{clientKeys}[clientId.\texttt{host}]$. As i is honest in S we have

$$cKey \equiv s_0^i.\texttt{clientKeys}[clientId.\texttt{host}] \equiv \mathsf{RRP}^i[clientId.\texttt{host}].$$

By the definition of RRP^i and $clientId.\texttt{host} \equiv data[\texttt{aud}].\texttt{host} \in \mathsf{dom}(r)$ we have that

$$cKey \equiv \mathsf{RRP}^i[clientId.\texttt{host}] \equiv \mathsf{rp\_signkey}(r).$$

In the initial state only the RP r knows $cKey$ and as it is honest up to S, r does not leak this value (r only uses $cKey \equiv S(r).\texttt{requestSignKey}$ in Line 41 of Algorithm B.13 to sign, it never includes it in a message such that it can be derived). Thus, only r can have created the term $req.\texttt{parameters}[\texttt{request}] \equiv signedReq \equiv \mathsf{sig}(reqObj, cKey)$.

The only point where r might sign using this key $cKey \equiv S(r).\texttt{requestSignKey}$ is Line 41 of Algorithm B.13. Thus, we get $reqObj[\texttt{client\_id}] \equiv \langle \texttt{URL}, \texttt{S}, dom_1, \langle\rangle, \langle\rangle, \bot\rangle$ for $dom_1 \in \mathsf{dom}(r)$ (Lines 33, 37 and 40) and $reqObj[\texttt{redirect\_uri}] \equiv \langle \texttt{URL}, \texttt{S}, dom_2, /\texttt{redirect\_ep}, \langle\rangle, \bot\rangle$ for $dom_2 \in \mathsf{dom}(r)$ (Lines 11, 12 and 40). Using this we get $u.\texttt{host} \equiv redirect\_uri.\texttt{host} \equiv reqtObj[\texttt{redirect\_uri}].\texttt{host} \equiv dom_2 \in \mathsf{dom}(r)$ and $u.\texttt{protocol} \equiv redirect\_uri.\texttt{protocol} \equiv reqObj[\texttt{redirect\_uri}].\texttt{protocol} \equiv \texttt{S}$. The lemma holds for this case.

Case 3: Signed request sent by reference. Together with the stored (unsigned) data of the original request, we get a case analogous to the case for the signed request sent by value. The main difference is that $signedReq' \equiv resp.\texttt{body}$ (Line 14) for an HTTP response message that i must have processed in Algorithm B.16 instead $signedReq \equiv req.\texttt{parameters}[\texttt{request}]$ for a request message req that i must have processed in the previous case. We have that Lines 15 to 28 of Algorithm B.16 correspond to Lines 19 to 32 of Algorithm B.15 with the processing in Algorithm B.17 being the same, and also r can create the signed request only in Algorithm B.13, such that the arguments from the above case apply, and the Lemma holds for this case as well. □

Lemma E.1.2 (ID Tokens do not leak)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Self-Issued Identity $id \in \mathsf{SIID}$ where $\mathsf{id\_holder}(id) = i$ is an honest Same-Device Self-Issued OP in S and $\mathsf{browser}(i)$ an honest browser in S, every Same-Device RP r that is honest in S, and every ID Token t with $t \equiv \mathsf{sig}(z, k)$ for a dictionary term z and $k \in \mathsf{proof\_key}(id)$ and $data \equiv z.\texttt{token\_data}$ with $data[\texttt{aud}].\texttt{host} \in \mathsf{dom}(r)$, $data[\texttt{aud}].\texttt{protocol} \equiv \texttt{S}$, $data[\texttt{sub}] \equiv id$ and every atomic DY process $p \in \mathcal{W} \setminus \{i, r, \mathsf{browser}(i)\}$ we have that $t \notin d_\emptyset(S(p))$.

Proof. The signing key $k \in \mathsf{proof\_key}(id)$ in the initial state is only known to $\mathsf{id\_holder}(id) = i$. As i is honest up in the state S, i only retrieves such a signing key k in Lines 23 to 28 and 36 of Algorithm B.17 and only uses this key k to derive the public key $\mathsf{pub}(k)$ (Line 39) and sign (Line 50). Thus, i does not leak this key to another process (neither $\mathsf{pub}(k)$ nor $\mathsf{sig}(z, k)$ for some term z allow to derive k). Therefore, every term $t \equiv \mathsf{sig}(z, k)$ for some term z must have been created by i.

Such a token term t is emitted by i only in a message $m' \equiv \mathsf{enc_s}(m, lkey)$ for a term $lkey \equiv S(i).\texttt{browserEncKey}$ and an HTTP response message m with $m.\texttt{headers}[\texttt{Location}] \equiv \mathsf{enc_a}(u, pk)$ for a redirection URL u with $u.\texttt{fragment}[\texttt{id\_token}] \equiv t$ (Lines 50 to 57 of Algorithm B.17). By the definition of the initial state of i and that i never changes this componet we

get $lkey \equiv s_0^i.\texttt{browserEncKey} \equiv \mathsf{pub}(\mathsf{localkey}(\mathsf{browser}(i)))$. As $\mathsf{browser}(i) = b$ is honest up to S, b is the only process that knows $\mathsf{localkey}(b)$ in the initial state, and b does not leak the nonce $s_0^b.\texttt{localCallKey} \equiv \mathsf{localkey}(b)$, as it only uses it to decrypt when receiving a request, only b may decrypt $m.\texttt{headers}[\texttt{Location}]$ and obtain u. The only point where b decrypts using this key is Line 18 in Algorithm B.2. The browser then sends an HTTP(S) request to the URL u given in the location header.

By Lemma E.1.1, we have that for the redirection URL u it holds that $u.\texttt{host} \in \mathsf{dom}(r)$ and $u.\texttt{protocol} \equiv \texttt{S}$.

The browser b now sends a message containing u encrypted towards r like

$$m'_{\texttt{br}} \equiv \mathsf{enc_a}(\langle m_{\texttt{br}}, k_{\texttt{br}}\rangle, s_0^b.\texttt{keyMapping}[u.\texttt{host}])$$

for an HTTP request $m_{\texttt{br}}$ and a nonce $k_{\texttt{br}}$. Note that since b is honest in S, we have $s_0^b.\texttt{keyMapping} \equiv S(b).\texttt{keyMapping}$. It follows that $m'_{\texttt{br}} \equiv \mathsf{enc_a}(\langle m_{\texttt{br}}, k_{\texttt{br}}\rangle, \mathsf{pub}(\mathsf{tlskey}(u.\texttt{host})))$. Note that $m_{\texttt{br}}$ does not contain $u.\texttt{fragment}$, but the complete URL u including $u.\texttt{fragment}$ is stored in the pending requests of b. From there b might pass it on for a redirect response, or store it in a document when processing a response that is not a redirect. We thus track which responses b might process. Using the Browser HTTPS Lemma, we get that only r can decrypt the message $m'_{\texttt{br}}$ and create a response for the message, as $u.\texttt{host} \in \mathsf{dom}(r)$, in the initial state only r knows the private TLS key $\mathsf{tlskey}(dom)$ for $dom \in \mathsf{dom}(r)$ and r is honest up to S and thus does not leak $\mathsf{tlskey}(dom)$ in a state prior to S.

As i returned the HTTP status code 303 with the response to the browser and the request i processed before must have been a GET or POST request, the HTTP request $m_{\texttt{br}}$ to r must be a GET request. Thus, r might process the request only starting in Line 2, Line 17, Line 19 or Line 21 of Algorithm B.5.

If r processes the request starting at Line 17, r executes Algorithm B.10 and retrieves $registration \equiv S(r).\texttt{sessions}[sid][\texttt{registration}]$ (Lines 5 and 7) for a term sid and places it in the body of the response. Note that r must have set this value in Algorithm B.13, Line 21, where r only includes the key `subject_syntax_types_supported` in the dictionary. If b tries to retrieve the script for this string, b stops, thus this case can not lead to t being leaked. If r processes the request starting at Line 19, r invoke Algorithm B.11 where r might retrieve $S(r).\texttt{sessions}[sid][\texttt{requestObject}] \equiv reqObj$ (Lines 5 and 7) for a term sid and places it in the body of the response. Note that r must have set this value in Algorithm B.13 where we get from Line 41 that the response body has the shape $reqObj \equiv \mathsf{sig}(z_1, z_2)$ for some terms z_1 and z_2. When processing such a body b stops, at is not of the expected shape of a script and an initial script state.

Starting processing the request at Line 2, r sends a response to b with the script string for *script_rp_index* given in Algorithm B.3.

In the case where r starts the processing of the GET request in Line 21 of Algorithm B.5. In this case r responds to b with the script string for *script_rp_get_fragment* in the body.

For the possible scripts *script_rp_index* and *script_rp_get_fragment* in the response, b then assembles a document that contains the request URL u (including t) as the location and stores it in its state. The browser b now may only retrieve this document when reloading (then b sends a message like $m'_{\texttt{br}}$ to r again), or when processing the scripts *script_rp_get_fragment* or *script_rp_index*

from the response. Note that t is only included in $u.\texttt{fragment}$ or the body of a POST request to r. As a result we have that b will not leak t via the `Referer` header because an honest browser never includes the fragment in this header.

The script *script_rp_index* (Algorithm B.3) either outputs a `FORM` command, that does not retrieve a fragment from the document tree and thus cannot not contain t, or issues a `HREF` command for some URL build from constants, that too does not contain t.

When processing the script *script_rp_get_fragment* (Algorithm B.4), b retrieves the document location URL u and extracts the fragment of u which might contain t and processes a `FORM` command with the fragment in the body, and sends a `POST` request for $\langle \texttt{URL}, \texttt{S}, u.\texttt{host}, /\texttt{redirect\_ep}, \langle\rangle, \langle\rangle\rangle$. As r never leaks the private TLS key for its domains (as above), only r can decrypt and process this encrypted HTTP request m'_p containing the HTTP request m_p with $m_p.\texttt{body} \equiv u.\texttt{fragment}$.

The only point where r can process the request m_p is starting from Line 21 of Algorithm B.5 (as it is a `POST` request). There, r might retrieve $pdata \equiv m_p.\texttt{body}$ with $t \equiv pdata[\texttt{id\_token}]$ (Line 32) and store the ID token term as $S(r).\texttt{sessions}[sid][\texttt{id\_token}] \equiv t$ (Line 37) as well as the message m_p as $S(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}][\texttt{message}]$ (Line 36) that contains t in $m_p.\texttt{body}$.

The only point where the RP r might retrieve t from $S'(r).\texttt{sessions}[sid][\texttt{id\_token}]$ is Line 3 of Algorithm B.7. In this algorithm we see that r emits or stores no value that contains t. $S'(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}][\texttt{message}] \equiv request$ is only accessed by r in Algorithm B.9, but there r only uses $request.\texttt{nonce}$, while t might only be contained in $request.\texttt{body}$.

Thus, we have that a term t as given in the Lemma E.1.2 does not leak to a process p outside honest r, $b = \mathsf{browser}(i)$ and i. We have that $t \notin d_\emptyset(S(p))$ and the lemma holds. □

Lemma E.1.3 (DID Cache Integrity)
Let SIOID^n be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of SIOID^n, every configuration (S, E, N) in ρ, every Self-Issued Identity $id \in \mathsf{SIID} \cap \mathsf{DID}$ where $\mathsf{govenor}(id) = vdr$ is an honest VDR in S and every Same-Device RP r that is honest in S we have that $S(r).\texttt{didCache}[id] \equiv \langle\rangle$ (not set) or $doc \equiv S(r).\texttt{didCache}[id]$ with $doc \equiv \mathsf{resolveDoc}(id)$.

Proof. First note that we have $s^r_0.\texttt{didCache} \equiv \langle\rangle$, the DID Cache is empty in the initial state of every RP atomic DY process r. It follows $s^r_0.\texttt{didCache}[id] \equiv \langle\rangle$ for every $id \in \mathsf{DID} \cap \mathsf{SIID}$. If $S(r).\texttt{didCache}[id] \not\equiv \langle\rangle$, we see that r must have assigned a value at some point.

The only point where an honest r might change the content of $S(r).\texttt{didCache}[id]$ is Line 8 in Algorithm B.6. This Line is only executed, if r processes an HTTP response *resp* for a request *req* that r sent before, for which r stored $reference[\texttt{responseTo}] \equiv \texttt{DID\_RESOLVE}$ and $reference[\texttt{requestedDID}] \equiv resp.\texttt{body.did} \equiv id$. On receiving this response *resp* the RP r sets $S(r).\texttt{didCache}[id] \equiv resp.\texttt{body}$ (Lines 4 and 8 of Algorithm B.6). The only point where r assembles such a request for the stored reference is in Algorithm B.12.

In Algorithm B.12, the RP r assembles the request *req* for the stored reference (Line 5) with $dom \equiv req.\texttt{host} \equiv S(r).\texttt{didResolvers}[id.\texttt{method}]$ (Lines 2 and 4) with $req.\texttt{params} \equiv [\texttt{did} : id]$ (Line 3). Note that we have

$$dom \equiv S(r).\texttt{didResolvers}[id.\texttt{method}] \equiv s^r_0.\texttt{didResolvers}[id.\texttt{method}] \equiv \langle \mathsf{didResolvers}^r \rangle$$

as r is honest and never assigns to the DID resolver component of its state. We get that

$$dom \in \mathsf{dom}(\mathsf{method\_gov}(id.\mathtt{method})) = \mathsf{dom}(\mathsf{govenor}(id)) = \mathsf{dom}(vdr)$$

with $vdr = \mathsf{govenor}(id)$. This request req includes in an encrypted HTTP request $req' \equiv \mathsf{enc_a}(\langle req, rkey\rangle, S(r).\mathtt{keyMapping}[dom])$. Note that by the definition of the key mapping in the initial state and that r never assigns to it, we get that $S(r).\mathtt{keyMapping}[dom] \equiv \mathsf{pub}(\mathsf{tlskey}(dom))$. As vdr is honest up to S and does not leak the private TLS key nonce $\mathsf{tlskey}(dom)$ and neither r nor vdr leak the symmetric encryption key $rkey$ in the request, only vdr can decrypt the request to obtain req from r and only vdr can create the response $resp$.

The only possible response vdr may give is determined in Algorithm B.18 where vdr retrieves with a term $doc \in^{\langle\rangle} S(vdr).\mathtt{docs}$ such that $doc.\mathtt{did} \equiv req.\mathtt{parameters}[\mathtt{did}]$ (Lines 3 and 10). As an honest vdr never changes the stored documents component $\mathtt{docs}$ of its state, $S(vdr).\mathtt{docs} \equiv s_0^{vdr}.\mathtt{docs}$ and we have $doc \in \mathsf{DIDDoc}^{vdr}$ with $doc \equiv \mathsf{resolveDoc}(id)$ by the definition of the initial state of vdr. We see that vdr sends $resp$ in an encrypted HTTP response with $resp.\mathtt{body} \equiv doc$. We get that $S(r).\mathtt{didCache}[id] \equiv doc \equiv \mathsf{resolveDoc}(id)$. □

Lemma E.1.4 (Verification Key Integrity)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Self-Issued Identity $id \in \mathsf{SIID}$ for which we have that if $id \in \mathsf{DID}$ then $\mathsf{govenor}(id)$ is an honest VDR, every Same-Device RP r that is honest in S, every term sid such that $S(r).\mathtt{sessions}[sid][\mathtt{loggedInAs}] \equiv id$, we have that $S(r).\mathtt{sessions}[sid][\mathtt{id\_token}] \equiv \mathsf{sig}(t', k)$ for $k \in \mathsf{proof\_key}(id)$ and some term t' with $t'.\mathtt{token\_data}[\mathtt{sub}] \equiv id$.

Proof. If $S(r).\mathtt{sessions}[sid][\mathtt{loggedInAs}] \equiv id$ for some term sid then the RP r must have set this value before as $s_0^r.\mathtt{sessions} \equiv \langle\rangle$. We get that r must have executed Line 2 of Algorithm B.9 as this is the only point where r might set this value.

Algorithm B.9 might only be called in Algorithm B.7 and Algorithm B.8, and as Algorithm B.8 is only called in Algorithm B.7, Algorithm B.7 must have been executed before, at least up to Line 38.

From Algorithm B.7 we get that $S(r).\mathtt{sessions}[sid][\mathtt{id\_token}] \equiv t$ for a term t (Lines 2 and 3) with $\mathsf{extractmsg}(t).\mathtt{token\_data} \equiv data$ (Line 4) with $id \equiv data[\mathtt{sub}]$ such that $id \in \mathsf{DID}$ or $id \equiv \langle \mathtt{jkt}, z\rangle$ for a term id' (Lines 13, 14 and 19). We see that t is of the shape $t \equiv \mathsf{sig}(t', k)$ for a term k and a sequence of terms t'.

We first consider the case where $id \in \mathsf{DID}$. Note that by Lemma E.1.3 we have that $S(r).\mathtt{didCache}[id] \equiv \langle\rangle$ or $S(r).\mathtt{didCache}[id] \equiv \mathsf{resolveDoc}(id)$. Note that due to the check in Line 20 we have that $S(r).\mathtt{didCache}[id] \not\equiv \langle\rangle$. We get that $didDoc \equiv S(r).\mathtt{didCache}[id] \equiv \mathsf{resolveDoc}(id)$. The RP r retrieves $kid \equiv \mathsf{extractmsg}(t).\mathtt{kid}$ such that $kid \in \mathsf{DIDURL}$, $kid.\mathtt{did} \equiv id$ and $kid.\mathtt{key} \in didDoc.\mathtt{verification}$ (Lines 21 and 23). We have that $kid \in \mathsf{vkid}(id)$. The RP also retrieves a term $vKey \equiv didDoc.\mathtt{verification}[kid.\mathtt{key}]$ (Line 26). We have r must have executed Line 33 and get $\mathsf{checksig}(t, vKey) \equiv \mathsf{checksig}(\mathsf{sig}(t', k), v) \equiv \top$ for terms t' and k such that $vKey \equiv \mathsf{pub}(k)$. We have that $k \in \mathsf{proof\_key}(id)$ by definition of $\mathsf{proof\_key}$.

If id is a not a DID then it must hold that $id \equiv \langle \texttt{jkt}, id' \rangle$ (Line 14). with $id' \equiv \mathsf{hash}(vKey)$ (Line 16) for a term $vKey \equiv data[\texttt{sub\_jwk}]$ (Line 15) such that $\mathsf{checksig}(t, vKey) \equiv \mathsf{checksig}(\mathsf{sig}(t', k), vKey) \equiv \top$ (Line 33) for some nonce k. It follows $vKey \equiv \mathsf{pub}(k)$. We have $id \equiv \langle \texttt{jkt}, \mathsf{hash}(vKey) \rangle \equiv \langle \texttt{jkt}, \mathsf{hash}(\mathsf{pub}(k)) \rangle$. By the definition of the mapping $\mathsf{proof\_key}$ we get $k \in \mathsf{proof\_key}(id)$. □

Lemma E.1.5 (Authorization Endpoint Integrity)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $i \in \mathsf{SIOP}$ that is honest in S where $y = \mathsf{configProvider}(i)$ is an honest Configuration Provider in S, every term $sessionId$, every term g, and every URL $issId$ such that

1. $S(r).\texttt{sessions}[sessionId] \equiv g$,
2. $g[\texttt{issuer}] \equiv issId$, $issId.\texttt{host} \in \mathsf{dom}(i)$ and $issId.\texttt{protocol} \equiv \texttt{S}$,

we have that $issId.\texttt{host} \notin S(r).\texttt{oidcConfigCache}$ (not set) or there is a URL $auth\_ep$ with

$$S(r).\texttt{oidcConfigCache}[issId.\texttt{host}][\texttt{authorization\_endpoint}] \equiv auth\_ep,$$

$auth\_ep.\texttt{host} \in \mathsf{dom}(i)$ and $auth\_ep.\texttt{protocol} \equiv \texttt{S}$.

Proof. Let $g \equiv S(r).\texttt{sessions}[sessionId]$ for a state S with $g[\texttt{issuer}].\texttt{host} \in \mathsf{dom}(i)$ and $g[\texttt{issuer}].\texttt{protocol} \equiv \texttt{S}$ such that RP r, Self-Issued OP i and $y = \mathsf{configProvider}(i)$ are honest in S. For the initial state we have that $s_0^r.\texttt{oidcConfigCache} \equiv \langle\rangle$ and thus $issId.\texttt{host} \notin s_0^r.\texttt{oidcConfigCache}$.

If we have $g[\texttt{issuer}].\texttt{host} \in S(r).\texttt{oidcConfigCache}$, then r must have set the value before. The only place where r assigns to $S(r).\texttt{oidcConfigCache}$ is in Line 15 of Algorithm B.6. For the line to be executed and $S(r).\texttt{oidcConfigCache}[issId.\texttt{host}]$ for $issId \equiv g[\texttt{issuer}]$ to bet set, r must process a response $resp$ for a request req that r sent before with $reference[\texttt{session}] \equiv sessionId$ (Line 2) and $reference[\texttt{responseTo}] \equiv \texttt{CONFIG}$ (Line 10).

The only point where r might send a request req is in Algorithm B.13, Lines 4 to 8. There, r assembles an HTTP request with the host $req.\texttt{host} \equiv g[\texttt{issuer}].\texttt{host} \equiv issId.\texttt{host}$ (Line 4) and the path $/.\texttt{well} - \texttt{known}/\texttt{openid} - \texttt{configuration}$ (Line 5) and sends it using the algorithms from the generic HTTPS server model. We get that r sends an encrypted HTTP message $req' \equiv \mathsf{enc_a}(\langle req, key \rangle, k)$ for a nonce key and $k \equiv S(r).\texttt{keyMapping}[req.\texttt{host}] \equiv issId.\texttt{host}$. From the definition of the initial state s_0^r and that honest r never assigns to its key mapping, we get that $k \equiv \mathsf{pub}(\mathsf{tlskey}(issId.\texttt{host}))$.

In the initial state only $y \in \mathsf{CP}$ with $issId.\texttt{host} \in \mathsf{dom}(i) = \mathsf{dom}(y)$ knows the private TLS key $\mathsf{tlskey}(issId.\texttt{host})$ for decrypting this request and as $y = \mathsf{configProvider}(i)$ is honest in S, it never leaks the key. Thus, only y can decrypt the request, and obtain the nonce key needed to encrypt the response. As neither y nor r leak the nonce key to another process, and r never encrypts with key, only y may create the response $resp$ for the request req.

Algorithm B.14 describes how the honest Configuration Provider y answers the request. We see that y chooses a URL $auth\_ep$ with $auth\_ep.\texttt{host} \in \mathsf{dom}(y)$ and $auth\_ep.\texttt{protocol} \equiv \texttt{S}$ (Line 7) and includes the URL in the response body such that $resp.\texttt{body}[\texttt{authorization\_endpoint}] \equiv auth\_ep$ (Lines 8 and 11).

When processing this HTTP response *resp*, r stores the body in its state as

$$S(r).\texttt{oidcConfigCache}[\mathit{issId}.\texttt{host}] \equiv \mathit{resp}.\texttt{body}$$

(Lines 11 and 15 of Algorithm B.6). From this we get

$$S(r).\texttt{oidcConfigCache}[\mathit{issId}.\texttt{host}][\texttt{authorization\_endpoint}] \equiv \mathit{auth\_ep}$$

for a URL *auth_ep* with $\mathit{auth\_ep}.\texttt{host} \in \mathsf{dom}(y) = \mathsf{dom}(i)$ and $\mathit{auth\_ep}.\texttt{protocol} \equiv \texttt{S}$. □

Lemma E.1.6 (Confidentiality of request object reference)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every domain $h \in \mathsf{dom}(r)$, every $i \in \mathsf{SIOP}$ that is honest in S where $b = \mathsf{browser}(i)$ is an honest browser in S and $\mathsf{configProvider}(i)$ is an honest Configuration Provider in S, every nonce $\mathit{sessionId} \in \mathcal{N}$, every atomic DY process $p \in \mathcal{W} \setminus \{r, b, i\}$, every term $g \in \mathcal{T}_{\mathcal{N}}$, every cookie $c \equiv \langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle \mathit{sessionId}, \top, \top, \top\rangle\rangle$ such that

1. $S(r).\texttt{sessions}[\mathit{sessionId}] \equiv g$,
2. $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$, and
3. $g[\texttt{issuer}].\texttt{host} \in \mathsf{dom}(i)$ and $g[\texttt{issuer}].\texttt{protocol} \equiv \texttt{S}$,

we have that $\texttt{request\_uri} \notin g$ or there is a nonce *ref* such that

$$\mathit{ref} \equiv g[\texttt{request\_uri}].\texttt{parameters}[\texttt{ref}]$$

and $\mathit{ref} \notin d_\emptyset(S(p))$.

Proof. Let $g \equiv S(r).\texttt{sessions}[\mathit{sessionId}]$ for a state S that fulfills the requirements from the lemma. In the initial state we have $s_0^r.\texttt{sessions} \equiv \langle\rangle$, thus, if $\texttt{request\_uri} \in g$, r must have assigned a value before. The only point where r may assign to this value is in Line 47 of Algorithm B.13. There, r chooses a fresh nonce *ref* (Line 46), adds it URL *reqUri* such that $\mathit{reqUri}.\texttt{parameters}[\texttt{ref}] \equiv \mathit{ref}$, and stores $\mathit{reqUri} \equiv g[\texttt{request\_uri}]$ (Line 47). Note that $\mathit{reqUri}.\texttt{host} \in \mathsf{dom}(r)$, $\mathit{reqUri}.\texttt{protocol} \equiv \texttt{S}$ and $\mathit{reqUri}.\texttt{path} \equiv \texttt{/request\_ep}$ (Line 45). Note that r only retrieves $g[\texttt{request\_uri}]$ from its state in Line 5 of Algorithm B.11 where r only uses it to check it against another value and does not store or send a term that may contain *ref*.

If a cookie $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$ with $c.\texttt{name} \equiv \langle\texttt{\_\_Host}, \texttt{sessionId}\rangle$ for a domain $h \in \mathsf{dom}(r)$, we see that b must have stored this cookie before. From the browser model we get that b only might add the cookie c to $S(b).\texttt{cookies}[h]$ if b received a response m_r for an encrypted HTTP request $m_l' \equiv \mathsf{enc_a}(\langle m_l, \mathit{key}_0\rangle, S(b).\texttt{keyMapping}[h])$ with m_l an HTTP request, key_0 a nonce, $h \equiv m_l.\texttt{host}$ and $c \in^{\langle\rangle} m_r.\texttt{headers}[\texttt{Set-Cookie}]$. Note that due to the `__Host` prefix the request must have been sent using the HTTPS protocol, as otherwise the browser b would not store the cookie c in its state. From the definition of the initial state and that b never assigns to the key mapping we get that $S(b).\texttt{keyMapping}[h] \equiv \mathsf{pub}(\mathsf{tlskey}(h))$. In the initial state all processes except r only know $\mathsf{pub}(\mathsf{tlskey}(h))$ (and not the private key) and r never leaks $\mathsf{pub}(\mathsf{tlskey}(h))$ or key_0. From the Browser HTTPS Lemma we get that only r could have created the encrypted HTTP response for the request m_l'.

Note that Algorithm B.13 includes the only point where r might send a `Set-Cookie` header for a cookie with name $\langle$`__Host-sessionId`$\rangle$ in a response m_r. In this response m_r, r also includes *ref* in m_r.`headers[Location]` $\equiv$ *authEP* (Line 56) for a URL *authEP* with *authEP*.`parameters[request_uri]` $\equiv$ *reqUri* (Line 48). Using that *authEP*.`path` $\equiv$ *authEP*′.`path` and *authEP*.`protocol` $\equiv$ *authEP*′.`protocol` for

$$authEP' \equiv S(r).\texttt{oidcConfigCache}[g[\texttt{issuer}].\texttt{host}][\texttt{authorization\_endpoint}] \equiv authEP'$$

and Lemma E.1.5, we get that *authEP*.`host` $\in \mathsf{dom}(i)$ and *authEP*.`protocol` $\equiv$ `S`. Note that also only the b and r may learn key_0 and that r sends $m_r' \equiv \mathsf{enc_s}(m_r, key_0)$ as a response to m_l'.

When processing the HTTP response m_r, the browser b follows the redirect to the URL *authEP* in the location header and sends a request $m_i' \equiv \mathsf{enc_a}(\langle m_i, key_1\rangle, S(b).\texttt{keyMapping}[dom])$ for a nonce key_1 and an HTTP request m_i with $dom \equiv m_i.\texttt{host} \equiv authEP.\texttt{host}$ and $m_i.\texttt{method} \equiv \texttt{GET}$. Note that only i knows $\mathsf{tlskey}(dom)$ in the initial state and never leaks it to another process and $S(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(\mathsf{tlskey}(dom))$. Using Browser HTTPS Lemma, we get that only i may decrypt m_i' and obtain m_i.

When i processes the request m_i, which it always starts in Algorithm B.15, we have that i stores the message in its state as $S(i).\texttt{sessions}[isid][\texttt{startRequest}][\texttt{message}] \equiv m_i$ for a term *isid* (Line 12) and also retrieves the parameters $data \equiv m_i.\texttt{parameters}$ (Line 4) and sends an encrypted HTTP request $req' \equiv \mathsf{enc_a}(\langle req, key_2\rangle, S(i).\texttt{keyMapping}[h'])$ with key_2 a nonce, *req* an HTTP request with $req.\texttt{host} \equiv h'$, $req.\texttt{path} \equiv reqUri.\texttt{path} \equiv$ `/request_ep`, $req.\texttt{parameters}[\texttt{ref}] \equiv reqUri.\texttt{parameters}[\texttt{ref}] \equiv ref$ (the only point wher req' contains *ref*) and $h' \equiv data[\texttt{request\_uri}].\texttt{host} \equiv reqUri.\texttt{host} \in \mathsf{dom}(r)$ (Lines 14 to 17) as `request_uri` $\in$ *data* holds (Line 13) by the above. Note that only r can decrypt this request req' and obtain *req* as $S(i).\texttt{keyMapping}[h'] \equiv \mathsf{pub}(\mathsf{tlskey}(h'))$ (by the definition of the key mapping in the initial state to which i never assigns) and only r knows $\mathsf{tlskey}(h')$ in the initial state and never leaks it.

The only point where r may decrypt using this key is in the generic HTTPS server model main relation, from where r passes *req* to Algorithm B.5. As the path of the request *req* is `/request_ep`, r calls Algorithm B.11 with *req*. There, r does not store the message parameters, or send them in a message.

Now, we check if i might send out a message including the nonce *ref* at another point, when retrieving the term $S(i).\texttt{sessions}[isid][\texttt{startRequest}][\texttt{message}]$ that contains it. The only points where i may retrieve this value are Line 8 of Algorithm B.16 and Line 52 of Algorithm B.17. In Line 8 of Algorithm B.16, i might access the parameter dictionary of the stored request message m_i and store it in $S(i).\texttt{sessions}[isid'][\texttt{params}]$ (Line 29) for a term $isid'$. Then, i may retrieve the value of this term $S(i).\texttt{sessions}[sid][\texttt{params}]$ in Algorithm B.17, Line 2, but there i never accesses the key `request_uri` of the stored dictionary and thus does not leak *ref* from there. When retrieving m_i in Line 52 of Algorithm B.17, i only accesses $m_i.\texttt{nonce}$ and never uses *ref*.

Thus, b, i and r never leak the nonce *ref*. We have $ref \notin d_\emptyset(S(p))$ for every process $p \notin \{b, i, r\}$ and thus Lemma E.1.6 holds. □

Lemma E.1.7 (Confidentiality of the Nonce Parameter)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every domain $h \in \mathsf{dom}(r)$, every $i \in \mathsf{SIOP}$ that is honest in S with $b = \mathsf{browser}(i)$ is an

honest browser in S and $\mathsf{configProvider}(i)$ is an honest Configuration Provider in S, every nonce $sessionId \in \mathcal{N}$, every atomic DY process $p \in \mathcal{W} \setminus \{r, b, i\}$, every term $g \in \mathcal{T}_{\mathcal{N}}$, every cookie $c \equiv \langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle sessionId, \top, \top, \top\rangle\rangle$ such that

1. $S(r).\texttt{sessions}[sessionId] \equiv g$,
2. $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$, and
3. $g[\texttt{issuer}].\texttt{host} \in \mathsf{dom}(i)$ and $g[\texttt{issuer}].\texttt{protocol} \equiv \texttt{S}$,

we have that $g[\texttt{nonce}] \equiv \langle\rangle$ or $g[\texttt{nonce}] \notin d_\emptyset(S(p))$.

Proof. First note that for a browser b that is honest in S if $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$ for a cookie $c \equiv \langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle sid, \top, \top, \top\rangle\rangle$, a nonce sid, and a domain $h \in \mathsf{dom}(r)$ then b must have added this cookie c before. From the browser model we get that b only might add the cookie c to $S(b).\texttt{cookies}[h]$ if b received a response m_r for an encrypted HTTP request $m_l' \equiv \mathsf{enc_a}(\langle m_l, key_0\rangle, S(b).\texttt{keyMapping}[h])$ with m_l an HTTP request, $h \equiv m_l.\texttt{host}$ and $c \in^{\langle\rangle} m_r.\texttt{headers}[\texttt{Set-Cookie}]$. Note that due to the $\texttt{\_\_Host}$ prefix the request must have been sent using the HTTPS protocol, otherwise b would not store the cookie in its state. From the definition of the initial state and that b never assigns to the key mapping we get that $S(b).\texttt{keyMapping}[h] \equiv \mathsf{pub}(\mathsf{tlskey}(h))$. In the initial state all processes except r only know $\mathsf{tlskey}(h)$ (and not the private key) and r never leaks $\mathsf{tlskey}(h)$ or key_0. From the Browser HTTPS Lemma we get that only r could have created the encrypted HTTP response for the request m_l'.

Note that Algorithm B.13 includes the only point where r might send a `Set-Cookie` header for a cookie with name $\langle\texttt{\_\_Host}, \texttt{sessionId}\rangle$ in a response m_r. This algorithm also contains the only point where r might set $S(r).\texttt{sessions}[sessionId][\texttt{nonce}]$ (which must have happened if $S(r).\texttt{sessions}[sessionId][\texttt{nonce}] \not\equiv \langle\rangle$).

We now follow the path of a nonce send with a response for such a request m_l'. The only point where r might set $nonce \equiv S(r).\texttt{sessions}[sessionId][\texttt{nonce}]$ is in Lines 12 and 38 of Algorithm B.13 — there r stores a term $data$ containing a fresh nonce $nonce$ as $data[\texttt{nonce}] \equiv nonce$. Let $g \equiv S(r).\texttt{sessions}[sessionId]$. Note that the RP r also stores $nonce$ contained in $sigRequest \equiv g[sessionId][\texttt{requestObj}]$ such that $\mathsf{extractmsg}(sigRequest)[\texttt{nonce}] \equiv nonce$ (Lines 12 and 40 to 42). This nonce is included in the response message m_r in the location header as $loc \equiv m_r.\texttt{headers}[\texttt{Location}]$ with $loc.\texttt{parameters}[\texttt{nonce}] \equiv nonce$ and $loc.\texttt{parameters}[\texttt{request}] \equiv sigRequest$ (Lines 50, 52, 54 and 56). Note that for this message we also have $m_r.\texttt{parameters}[\texttt{client\_id}] \equiv client\_id$ where $client\_id$ is a URL with $client\_id.\texttt{host} \in \mathsf{dom}(r)$ and $client\_id.\texttt{protocol} \equiv \texttt{S}$ (Lines 11, 33, 35, 37 and 54), and we also have $\mathsf{extractmsg}(sigRequest)[\texttt{client\_id}] \equiv client\_id$ (Lines 37, 40 and 41). This message m_r with $m_r.\texttt{status} \equiv 303$ is sent in Line 60 of Algorithm B.13. From Lines 10, 30 and 56 we get that $loc.\texttt{host} \equiv auth\_ep.\texttt{host}$ and $loc.\texttt{protocol} \equiv auth\_ep.\texttt{protocol}$ with

$$auth\_ep \equiv S(r).\texttt{oidcConfigCache}[g[\texttt{issuer}].\texttt{host}][\texttt{authorization\_endpoint}].$$

By Lemma E.1.5 we have that $g[\texttt{issuer}].\texttt{host} \notin S(r).\texttt{oidcConfigCache}$ or

$$S(r).\texttt{oidcConfigCache}[g[\texttt{issuer}]][\texttt{authorization\_endpoint}] \equiv auth\_ep$$

for a URL *auth_ep* with $auth\_ep.\texttt{host} \in \mathsf{dom}(i)$ and $auth\_ep.\texttt{protocol} \equiv \texttt{S}$. If r responded with code 303 however $g[\texttt{issuer}].\texttt{host} \notin S(r).\texttt{oidcConfigCache}$ could not have been the case as then the lines above would not have been executed (Line 3). We get that $loc.\texttt{host} \in \mathsf{dom}(i)$ and $loc.\texttt{protocol} \equiv \texttt{S}$ for the Self-Issued OP i with $g[\texttt{issuer}].\texttt{host} \in \mathsf{dom}(i)$.

When the browser b processes the HTTP response m_r, we get from the browser model that b follows the redirect given in the location header i.e., we see that b sends an encrypted HTTP message $m_a' \equiv \mathsf{enc_a}(\langle m_a, key_1\rangle, S(b).\texttt{keyMapping}[loc.\texttt{host}])$ for a nonce key_1 and an HTTP request m_a with $m_a.\texttt{method} \equiv \texttt{GET}$ and $m_a.\texttt{parameters} \equiv m_r.\texttt{parameters}$. Note that again for the key mapping of the browser we have that

$$S(b).\texttt{keyMapping}[loc.\texttt{host}] \equiv \mathsf{pub}(\mathsf{tlskey}(loc.\texttt{host})).$$

In the initial state all processes except i only know the public key $\mathsf{pub}(\mathsf{tlskey}(loc.\texttt{host}))$ (and not the private key) and i never leaks $\mathsf{pub}(\mathsf{tlskey}(loc.\texttt{host}))$. Using the Browser HTTPS Lemma, we get that only i can decrypt the request m_a' and obtain m_a.

We see that i might only process such a request m_a in Algorithm B.15. We first consider how i processes the nonce *nonce* that is included in the parameters of the request sent to i, directly as $m_a.\texttt{parameters}[\texttt{nonce}] \equiv nonce$ or $m_a.\texttt{parameters}[\texttt{request}] \equiv sigRequest$. Note that we have $m_a.\texttt{parameters}[\texttt{client\_id}] \equiv client\_id$ with $client\_id \in \mathsf{dom}(r)$ and $client\_id.\texttt{protocol} \equiv \texttt{S}$ by the above. In Algorithm B.15, i may only store the parameters $m_a.\texttt{parameters}$ in its state and the message m_a. The parameters i stores as

$$S(i).\texttt{sessions}[isid][\texttt{params}] \equiv m_a.\texttt{parameters}$$

for some term *isid* executing Lines 4 and 40, or use values from $\mathsf{extractmsg}(sigRequest)$ additionally to $m_a.\texttt{parameters}$ when executing Lines 4, 19, 32, and 40. In Line 12, i stores the message m_a as

$$S(i).\texttt{sessions}[isid][\texttt{startRequest}][\texttt{message}] \equiv m_a.$$

Before we see where i might retrieve *nonce* and process it further, we check for other points where r might send a message containing *nonce*. We see that r might retrieve $S(r).\texttt{sessions}[sessionId][\texttt{nonce}]$ only in Line 36 of Algorithm B.7 and 8 of Algorithm B.8. There r does not store *nonce* to another place, or send a message containing *nonce*. The term $S(r).\texttt{sessions}[sessionId][\texttt{requestObj}]$ containing *nonce*, might only be retrieved by r in Algorithm B.11.

We have that the RP r only may execute Algorithm B.11 and retrieve the signed request from its state

$$S(r).\texttt{sessions}[sessionId][\texttt{requestObj}] \equiv g[\texttt{requestObj}] \equiv sigRequest$$

if r processes an HTTP request m_o with the path $m_o.\texttt{path} \equiv \texttt{/request\_ep}$ and $ref \equiv m_o.\texttt{parameters}[\texttt{ref}] \equiv g[\texttt{request\_uri}].\texttt{parameters}[\texttt{ref}]$ such that $g[\texttt{request\_uri}] \not\equiv \langle\rangle$. By Lemma E.1.6, we get that only i, b and r may know *ref*. Thus, only i, b or r could have created the request m_o. Note that r may only send requests for the paths $\texttt{/resolve}$ (Algorithm B.12) or $\texttt{/.well-known/openid-configuration}$ (Algorithm B.13) and thus could not have created m_o. We also have that b may only send such a request when processing a redirect, or when processing a script command. As the attacker script cannot provide *ref*, and the scripts r may provide do not add a parameter that may contain *ref*, and i and r never add a parameter $\texttt{ref}$ to a URL they send in a Location header, only i could have created the request m_o.

Note that i must have sent the request m_o in an encrypted HTTP request $m'_o \equiv \mathsf{enc_a}(\langle m_o, key_2\rangle, tk)$ for some nonce key_2 and $tk \equiv S(i).\texttt{keyMapping}[m_o.\texttt{host}]$ by the definition of the generic HTTPS server model that i uses and as i never assigns to its key mapping and by the definition of the initial state, we get $tk \equiv \mathsf{pub}(\mathsf{tlskey}(m_o.\texttt{host}))$. The private key $\mathsf{tlskey}(m_o.\texttt{host})$ only r knows in the initial state as $m_o.\texttt{host} \in \mathsf{dom}(r)$ and never leaks it. Thus, only r can decrypt the request, and obtain key_2. Neither r nor i leak key_2, and thus only r and i can decrypt the encrypted HTTP response $m'_b \equiv \mathsf{enc_s}(m_b, key_2)$ that r sends (Line 8 of Algorithm B.11). We get that only i may process the response m_b as r never uses key_2 to decrypt. Note that $m_b.\texttt{body} \equiv g[\texttt{requestObj}] \equiv sigRequest$ with $\mathsf{extractmsg}(sigRequest)[\texttt{nonce}] \equiv nonce$ and also $\mathsf{extractmsg}(sigRequest)[\texttt{client\_id}] \equiv client\_id$ and that $nonce$ is contained in no other component of m_b.

We have that the Self-Issued OP i may only process m_b in Algorithm B.16. There, i might retrieve the $m_b.\texttt{body}$ in Lines 4 and 14. We get that i might only store the nonce contained in $m_b.\texttt{body}$ as $S(i).\texttt{sessions}[isid][\texttt{registration}] \equiv m_b.\texttt{body}$ (Line 5) and as $S(i).\texttt{sessions}[isid][\texttt{params}] \equiv rdata$ (Line 29) with $rdata \equiv unsig \cup \mathsf{extractmsg}(m_b.\texttt{body})$ for a term $unsig$ (Lines 14, 15 and 28). We get that $nonce$ may be stored under $S(i).\texttt{sessions}[isid][\texttt{params}][\texttt{nonce}]$.

We now check where the Self-Issued OP i may retrieve $nonce$ from its state. From the above we know that the nonce $nonce$ is only contained in the state of i as the value of the components $S(i).\texttt{sessions}[isid][\texttt{registration}]$, $S(i).\texttt{sessions}[isid][\texttt{startRequest}][\texttt{message}]$ and $S(i).\texttt{sessions}[isid][\texttt{params}][\texttt{nonce}]$. $S(i).\texttt{sessions}[isid][\texttt{registration}]$ is only retrieved in Line 7 of Algorithm B.17. There, i does not store another value or send a message that may contain $nonce$. We see that i may retrieve $S(i).\texttt{sessions}[isid][\texttt{startRequest}][\texttt{message}]$ only in Line 8 of Algorithm B.16 and 52 of Algorithm B.17. In the latter case, i does not retrieve the parameters that contain $nonce$. Following Line 8 of Algorithm B.16 i may again retrieve the parameters (Line 10), and store them to $S(i).\texttt{sessions}[isid][\texttt{params}]$ (Lines 27 to 29). Then, again $S(i).\texttt{sessions}[isid][\texttt{params}][\texttt{nonce}]$.

It remains to check where i accesses $S(i).\texttt{sessions}[isid][\texttt{params}][\texttt{nonce}]$. The only point where i may do this is in algorithm Algorithm B.17, Line 2 together with 34 or 46. Note that i does not add a parameter `ref` to the redirection URL u and that r does not include a parameter `ref` in the redirection URL. In both cases i includes the nonce $nonce$ contained in the ID token $idt \equiv u.\texttt{fragment}[\texttt{id\_token}]$ (Lines 34 and 50) and the Verifiable Presentation $vpt \equiv u.\texttt{fragment}[\texttt{vp\_token}]$ (Lines 46 and 47) for a URL u in a response message m_u that i sends with $m_u.\texttt{headers}[\texttt{Location}] \equiv \mathsf{enc_a}(u, bkey)$ with $bkey \equiv S(i).\texttt{browserEncKey}$ (Lines 51 to 57). As i is honest in S and i never assigns to $S(i).\texttt{browserEncKey}$, we get from the definition of the initial state that $bkey \equiv \mathsf{pub}(\mathsf{localkey}(b))$. We also have that

$$\mathsf{extractmsg}(idt).\texttt{token\_data}[\texttt{aud}] \equiv S(i).\texttt{sessions}[isid][\texttt{params}][\texttt{client\_id}] \equiv client\_id$$

(Lines 33 and 50) with $client\_id.\texttt{host} \in \mathsf{dom}(r)$ and $client\_id.\texttt{protocol} \equiv \texttt{S}$ from the above. Using this and Lemma E.1.1, we get that $u.\texttt{host} \in \mathsf{dom}(r)$ and $u.\texttt{protocol} \equiv \texttt{S}$. As b is honest, only b knows the private key $\mathsf{localkey}(b)$ in the initial state and never leaks it. Thus, only b may decrypt $m_u.\texttt{headers}[\texttt{Location}]$, and the only place where b might do this, is when processing a redirect in Line 18 of Algorithm B.2.

Now, b sends an encrypted HTTP request $m_t' \equiv \mathsf{enc_a}(\langle m_t, key_3\rangle, rkey)$ for a nonce key_3, $rkey \equiv S(b).\texttt{keyMapping}[client\_id.\texttt{host}]$ and an HTTP request m_t with $m_t.\texttt{method} \equiv \texttt{GET}$. Again, by arguments as the above and the Browser HTTPS Lemma, only r can decrypt this request to obtain m_t and only r can create a response for m_t'. Note that since *idt* and *vpt* that contain *nonce* is only contained in the fragment of the URL in the location header, m_t does not contain *nonce*. We see that b may only retrieve it when processing a response.

The RP r might only process this `GET` request starting in Line 2, Line 17, Line 19 or Line 21 of Algorithm B.5. Processing from Line 2 r sends a response with the script string for *script_rp_index* in the body (Line 4). This script string b stores in a document when processing the response, together with the *nonce* in the fragment of the document location. While b retrieves the location when processing the script *script_rp_index* (Algorithm B.3), it does not access the fragment. Also, as the fragment is never included in the referrer header, *nonce* cannot leak in this case. When r starts processing from Line 17, r invokes Algorithm B.10 and may send $regis \equiv S(r).\texttt{sessions}[\texttt{rsid}'][\texttt{registration}]$ for some term $\texttt{rsid}'$ in the response body. We see that $S(r).\texttt{sessions}[\texttt{rsid}'][\texttt{registration}]$ must have been set before, and that that can only happen in Algorithm B.13 where r does not include a script string. We see that b may store a document with the `nonce` in the fragment of the location URL, but never sends this fragment, as this can only happen when processing a script — in this case b will stop as it cannot retrieve a valid script string from the body. When r starts processing from Line 19, r invokes Algorithm B.11 and may send $reqObj' \equiv S(r).\texttt{sessions}[\texttt{rsid}''][\texttt{requestObj}]$ for some term $\texttt{rsid}''$ in the response body. We see that $S(r).\texttt{sessions}[\texttt{rsid}''][\texttt{requestObj}]$ must have been set before, and that that can only happen in Algorithm B.13, Lines 40 to 42, from which we get that $reqObj' \equiv \mathsf{sig}(reqDat, sigKey)$ for some terms *reqDat* and *sigKey*. When processing this response body b stops, as $reqDat \not\equiv \langle t_1, t_2\rangle$ for some terms t_1, t_2.

It remains the case where r starts processing the request m_t in Line 21 of Algorithm B.5. There, r sends the script string for *script_rp_get_fragment* in the response body (Line 29). This script string b stores in a document when processing the response together with *idt* and *vpt* that contains *nonce* in the fragment of the document location. When processing the script *script_rp_get_fragment* (Algorithm B.4), b retrieves the document location (Line 1) and from it the fragment (Line 3) and places it in the *data* of a `FORM` command with URL *url* such that $url.\texttt{host} \equiv u.\texttt{host} \in \mathsf{dom}(r)$, $url.\texttt{path} \equiv \texttt{/redirect\_ep}$ and $url.\texttt{protocol} \equiv \texttt{S}$. Again, b uses the public TLS key from its key mapping for a private TLS key only r knows for the encrpyted HTTP message that b sends when processing the request, and thus by the Browser HTTPS Lemma only r can decrypt this message and obtain the contained HTTP message m_i, $m_i.\texttt{body}[\texttt{vp\_token}] \equiv vpt$ and $m_i.\texttt{body}[\texttt{id\_token}] \equiv idt$.

As $m_i.\texttt{path} \equiv url.\texttt{path} \equiv \texttt{/redirect\_ep}$ and $m_i.\texttt{method} \equiv \texttt{POST}$, r processes this request starting in Line 21 of Algorithm B.5. There, r retrieves the request body (Line 32) and stores the terms *idt* and *vpt* that contain the nonce as $S(r).\texttt{sessions}[rsid][\texttt{id\_token}] \equiv idt$ (Line 37) and $S(r).\texttt{sessions}[rsid][\texttt{vp\_token}] \equiv vpt$ (Line 39) for some term *rsid*. The RP r retrieves the terms $S(r).\texttt{sessions}[rsid][\texttt{id\_token}] \equiv idt$ only in Line 3 of Algorithm B.7, and $S(r).\texttt{sessions}[rsid][\texttt{vp\_token}] \equiv vpt$ only in Line 6 of Algorithm B.8. In both algorithms r only retrieves

$$nonce \equiv \mathsf{extractmsg}(idt).\texttt{token\_data}[\texttt{nonce}] \equiv \mathsf{extractmsg}(vpt)[\texttt{challenge}]$$

to check against values and does not store a term or send a message that contains *nonce*. We get that *nonce* does not leak to a process other than b, r and i. □

Lemma E.1.8 (Verifiable Presentations do not leak)
Let $\mathcal{SIOID}^n$ be a Same-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Verifiable Credential $vc \in \mathsf{VC}$ where $\mathsf{vc\_holder}(vc) = i$ is an honest Same-Device Self-Issued OP in S and $\mathsf{browser}(i)$ an honest browser in S, every Same-Device RP r that is honest in S, and every term vpt with $vpt \equiv \mathsf{sig}(data, k)$ for $k \in K^{\mathsf{VP}}_{\mathrm{sign}}$ with $data[\texttt{domain}].\texttt{host} \in \mathsf{dom}(r)$, $data[\texttt{domain}].\texttt{protocol} \equiv \mathsf{S}$, $data[\texttt{vc}] \equiv vc$ and $vc[\texttt{subject}] \equiv \mathsf{pub}(k)$, and every atomic DY process $p \in \mathcal{W} \setminus \{i, r, \mathsf{browser}(i)\}$ we have that $vpt \notin d_\emptyset(S(p))$.

Proof. Note that $\mathsf{vc\_holder}(vc) = i$ for a Verifiable Credential vc implies that we there is a $k \in K^{\mathsf{VP}}_{\mathrm{sign}}$ with $\mathsf{key\_holder}(k) = i$ such that $\mathsf{extractmsg}(vc)[\texttt{subject}] \equiv \mathsf{pub}(k)$. Thus, as $i = \mathsf{key\_holder}(k)$ is an honest Self-Issued OP up to S, only i knows k such that $\mathsf{extractmsg}(vpt)[\texttt{subject}] \equiv \mathsf{pub}(k)$ in the initial state (in the component $s^i_0.\texttt{credentials}$) and i never leaks such a key, only i can create a Verifiable Presentation term vpt with $vpt \equiv \mathsf{sig}(data, k)$ for $k \in K^{\mathsf{VP}}_{\mathrm{sign}}$ with $data[\texttt{domain}].\texttt{host} \in \mathsf{dom}(r)$, $data[\texttt{domain}].\texttt{protocol} \equiv \mathsf{S}$, $data[\texttt{vc}] \equiv vc$ and $vc[\texttt{subject}] \equiv \mathsf{pub}(k)$ where vpt.

The only point where an honest Self-Issued OP i may retrieve a key $k \in K^{\mathsf{VP}}_{\mathrm{sign}}$ and sign a term $data$ is in Line 47 of Algorithm B.17. There i retrieves $k \equiv record[\texttt{key}]$ with $record \in^{\langle\rangle} \langle \mathsf{VCW}^i \rangle \equiv s^i_0.\texttt{credentials}$ and $record[\texttt{vc}] \equiv vc$ (Line 45). Note that in Line 46, we have that i sets $data[\texttt{vc}] \equiv vc$, and also sets $data[\texttt{domain}] \equiv tokendata[\texttt{aud}]$ (Lines 33 and 46) where $tokendata \equiv \mathsf{extractmsg}(idt).\texttt{token\_data}$ for the ID token $idt \equiv redUri.\texttt{fragment}[\texttt{id\_token}]$ (Lines 51 and 54) that i sends in the location header set to $\mathsf{enc_a}(redUri, bkey)$ (Lines 53 to 56) with $bkey \equiv S(i).\texttt{browserEncKey} \equiv \mathsf{pub}(\mathsf{localkey}(\mathsf{browser}(i)))$ by the definition of the initial state and that i never assigns a new value to this component. Note that $redUri$ also is the only component of this in which i sends vpt as $redUri.\texttt{fragment}[\texttt{vp\_token}] \equiv vpt$ (Lines 48 and 54). Using Lemma E.1.1 we get that $redUri.\texttt{host} \in \mathsf{dom}(r)$ and $redUri.\texttt{protocol} \equiv \mathsf{S}$.

Only the browser $b = \mathsf{browser}(i)$ can decrypt the location header and obtain $redUri$, as only only b knows $\mathsf{localkey}(b)$ in the initial state and never leaks it. The only point where b decrypts using $S(b).\texttt{localCallKey} \equiv \mathsf{localkey}(b)$ is when b processes a redirect in a response. We get that b sends a message $m'_r \equiv \mathsf{enc_a}(\langle m_r, key_0 \rangle, rkey)$ for an HTTP message m_r and a nonce key_0 with $m_r.\texttt{method} \equiv \texttt{GET}$, $m_r.\texttt{host} \equiv redUri.\texttt{host} \in \mathsf{dom}(r)$ and $rkey \equiv S(b).\texttt{keyMappig}[redUri.\texttt{host}] \equiv \mathsf{pub}(\mathsf{tlskey}(redUri.\texttt{host}))$ due to the definition of the initial state and as honest b never assigns to the key mapping. As processes other than r only know $rkey$ and not $\mathsf{tlskey}(redUri.\texttt{host}$ in the initial state, r never leaks the private key, and r never leaks the symmetric key key_0 that b sends with the request, we get using Browser HTTPS Lemma that only r can decrypt m'_r to obtain m_r, and only r can create a response for m_r. Note that the term vpt is not included in the request m_r as vpt is only contained in the fragment component of the URL $redUri$ that is not included in the request. The URL is only stored in the pending requests of b, and when b processes a response that is not a redirect b may store it in $d.\texttt{location}$ for a document d in the state of b. The only point where r (or any other party) might obtain the fragment from $d.\texttt{location} \equiv redUri$ is when b processes a script that accesses the document location, and only r might provide a response with a script that b accepts by the above.

The only script strings r might send in a response, are the script string *script_rp_get_fragment* (Line 29 of Algorithm B.5) and the script string for *script_rp_index* (Line 4 of Algorithm B.5). When processing *script_rp_index* (Algorithm B.3), b accesses the document location but does not access the fragment of the location. Thus, this script can not result in *vpt* being leaked to another process. When b processes the script *script_rp_get_fragment* (Algorithm B.4), b retrieves the document location and processes the command $\langle \texttt{FORM}, url, \texttt{POST}, redUri.\texttt{fragment}, \bot \rangle$ with $url.\texttt{path} \equiv \texttt{/redirect\_ep}$, $url.\texttt{host} \equiv redUri.\texttt{host} \in \mathsf{dom}(r)$ and $url.\texttt{host}.\texttt{protocol} \equiv \texttt{S}$. We see that b sends a message $m_t' \equiv \mathsf{enc_a}(\langle m_t, key_1 \rangle, rkey)$ for an HTTP message m_t and a nonce key_1 with $m_t.\texttt{method} \equiv \texttt{POST}$, $m_t.\texttt{host} \equiv url.\texttt{host} \in \mathsf{dom}(r)$, $m_t.\texttt{path} \equiv url.\texttt{path} \equiv \texttt{/redirect\_ep}$, $rkey \equiv\equiv \mathsf{pub}(\mathsf{tlskey}(redUri.\texttt{host}))$ and $m_t.\texttt{body} \equiv redUri.\texttt{fragment}$ (the component that contain *vpt*). With arguments as above and Browser HTTPS Lemma we get that only r can decrypt this message and obtain m_t.

The RP r may only process m_t starting in Line 21 of Algorithm B.5 as $m_t.\texttt{path} \equiv \texttt{/redirect\_ep}$. There r stores $vpt \equiv m_t.\texttt{body}[\texttt{vp\_token}]$ in its state only such that $S(r).\texttt{session}[sid][\texttt{vp\_token}] \equiv vpt$ for some term *sid* (Lines 32 and 39) and r stores the full request m_t in Line 36 under $S(r).\texttt{session}[sid][\texttt{redirect\_ep\_req}][\texttt{message}] \equiv m_t$.

The only point where r retrieves $S(r).\texttt{session}[sid][\texttt{vp\_token}]$ is Line 6 of Algorithm B.8. There r neither stores a term or sends a message that may contain *vpt*. Only in Line 11 of Algorithm B.9 r may retrieve $S(r).\texttt{session}[sid][\texttt{redirect\_ep\_req}]$ but r does not access the body of m_t there that contains *vpt*. Thus, *vpt* does not leak here either.

We get that no term *vpt* with $vpt \equiv \mathsf{sig}(data, k)$ for $k \in K^{\mathsf{VP}}_{\mathsf{sign}}$ with $data[\texttt{domain}].\texttt{host} \in \mathsf{dom}(r)$, $data[\texttt{domain}].\texttt{protocol} \equiv \texttt{S}$, $data[\texttt{vc}] \equiv vc$ and $vc[\texttt{subject}] \equiv \mathsf{pub}(k)$ can leak to an atomic DY process outside honest r, $i = \mathsf{vc\_holder}(\texttt{vc})$ and $\mathsf{browser}(i)$. □

E.2 Same-Device Authentication

In this section we prove the authentication property. For this we use that in a run between honest RP, browser and Self-Issued OP no ID token for an identity held by the Self-Issued OP with the RP in the audience is leaked (Lemma E.1.2) and that the RP cannot be confused about the key to use to verify an ID token for an identity (Lemma E.1.4).

Using these properties, we get that an honest RP only creates a new service session associated with an identity if it obtains an ID token with the correct nonce and audience, as well as a valid signature created by the holder of the identity. No other party than the honest Self-Issued OP and the browser might obtain such an ID token, and the Self-Issued OP might only invoke the RP via the browser. Thus the RP only sends the service session nonce to the browser and the attacker cannot obtain a session for an identity of held by an honest Self-Issued OP.

Theorem E.2.1 (Same-Device Authentication)
For every Same-Device Self-Issued OpenID Provider web system with a network attacker $\mathcal{SIOID}^n$, every run ρ of $\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Same-Device RP $r \in \mathsf{RP}$ that is honest in S, every Self-Issued Identity $id \in \mathsf{SIID}$ where $\mathsf{id\_holder}(id) = i$ is an honest Same-Device Self-Issued OP in S and $\mathsf{browser}(i)$ an honest browser in S, every service session identified by some nonce *service* for the Self-Issued Identity *id* at r where it holds that if $id \in \mathsf{DID}$ then $\mathsf{govenor}(id)$ is an honest VDR in S, it holds that $service \notin d_\emptyset(S(att))$ where *att* is the network attacker.

Proof. By the definition of a service session identified by some nonce *service* for the Self-Issued Identity *id* at r in a configuration (S, E, N) we have that there is a nonce *sid* with $S(r).\texttt{sessions}[sid][\texttt{serviceSessionId}] \equiv service$ and $S(r).\texttt{sessions}[sid][\texttt{loggedInAs}] \equiv id$. As by the definition of the initial state we have that $s_0^r.\texttt{sessions} \equiv \langle\rangle$, an RP r that is honest in S must have set these values by executing Algorithm B.9 before, as this algorithm is the only place where this values might be set.

The RP r only calls Algorithm B.9 with *id* in the input if Algorithm B.7 was executed successfully at least up to Line 38 (optionally Algorithm B.8 is called that then passes *id* to Algorithm B.9). We get from Algorithm B.7 that it must hold that there are terms *idt* and *session* with $session \equiv S(r).\texttt{sessions}[sid]$ (Line 2), $idt \equiv session[\texttt{id\_token}]$ (Line 3), and $data \equiv \mathsf{extractmsg}(idt).\texttt{token\_data}$ (Line 4) such that

- $S(r).\texttt{sessions}[sid][\texttt{client\_id}] \in^{\langle\rangle} data[\texttt{aud}]$ and $data[\texttt{aud}] \not\equiv \langle\rangle$ (Line 9)
- $S(r).\texttt{sessions}[sid][\texttt{nonce}] \equiv data[\texttt{nonce}]$ and $data[\texttt{nonce}] \not\equiv \langle\rangle$ (Line 36)
- $data[\texttt{sub}] \equiv id$ (Line 13).

For this we must have $S(r).\texttt{sessions}[sid][\texttt{nonce}] \not\equiv \langle\rangle$ and $S(r).\texttt{sessions}[sid][\texttt{client\_id}] \not\equiv \langle\rangle$. This implies Line 38 of Algorithm B.13 was executed before, and we see that $S(r).\texttt{sessions}[sid][\texttt{client\_id}] \equiv client\_id \in \mathsf{URLs}$ with $client\_id \in \mathsf{URLs}$, $client\_id.\texttt{host} \in \mathsf{dom}(r)$ and $client\_id.\texttt{protocol} \equiv \mathsf{S}$ (Lines 11 and 33 to 38), and that $S(r).\texttt{sessions}[sid][\texttt{nonce}]$ is a nonce (Lines 12 and 38). It follows $data[\texttt{aud}] \in \mathsf{URLs}$ with $data[\texttt{aud}].\texttt{host} \in \mathsf{dom}(r)$ and $data[\texttt{nonce}]$ is a nonce.

Using Lemma E.1.4 we get that $S(r).\texttt{sessions}[sid][\texttt{id\_token}] \equiv idt \equiv \mathsf{sig}(t', \mathsf{proof\_key}(id))$ for a term t', i.e., we have that only terms signed with the correct key for *id* can result in a service session identified by some nonce *service* for the Self-Issued Identity *id* at r.

We see that r must have set the value $idt \equiv S(r).\texttt{sessions}[sid][\texttt{id\_token}]$ before. Thus, Line 37 of Algorithm B.5, the only point where r might do this, must have been executed. For this r to happen we have that r must have processed a request message *req* in Algorithm B.5 with $req.\texttt{host} \in \mathsf{dom}(r)$ (follows from the generic HTTPS server model) $req.\texttt{body}[\texttt{id\_token}] \equiv idt$ (Lines 32 and 37), $req.\texttt{method} \equiv \texttt{POST}$ (Line 31) and $req.\texttt{headers}[\texttt{Cookie}][\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle] \equiv sid$ (Line 22). Note that r stores the request as $S(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}][\texttt{message}] \equiv req$.

By Lemma E.1.2 we have that for all atomic DY processes p except b, r and i, we have that $idt \notin d_\emptyset(S(p))$. Thus, only b, i and r might have created the request *req*. As i and r never send a request with a `Cookie` header, only b might have sent the request *req*.

Note that we get from the browser model that b sends such requests only an encrypted HTTP request $req' \equiv \mathsf{enc_a}(\langle req, key\rangle, rk)$ for a nonce *key* and $rk \equiv S(b).\texttt{keyMapping}[req.\texttt{host}]$. By the definition of the key mapping of b in the initial state and that the browser b never assigns to it we get $rk \equiv \mathsf{pub}(\mathsf{tlskey}(req.\texttt{host}))$ No other process than r may know $\mathsf{tlskey}(req.\texttt{host})$ in the initial state, and r never leaks this nonce. Using the Browser HTTPS Lemma we get that only r can decrypt the request and obtain the nonce *key*. This nonce r stores as $key \equiv S(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}][\texttt{key}]$ (Line 11 of Algorithm B.5).

The only point where another party might learn *service* is, when r sends *service* in the header `Set-Cookie` in Line 13 of Algorithm B.9. There, r retrieves

$$req \equiv S(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}][\texttt{message}] \text{ (Line 11) and}$$

$$key \equiv S(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}][\texttt{key}] \text{ (Lines 11 and 12)}$$

and sends an encrypted HTTP response $resp' \equiv \mathsf{enc_s}(resp, key)$ where *resp* is an HTTP response with $resp.\texttt{nonce} \equiv req.\texttt{nonce}$ and $resp.\texttt{headers}[\texttt{Set-Cookie}] \equiv \langle cookie \rangle$ for a cookie $cookie \equiv \langle\langle \texttt{\_\_Host}, \texttt{serviceSessionId}\rangle, \langle service, \top, \top, \top\rangle\rangle$. Note that this is also the only point where r might retrieve $key \equiv S(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}][\texttt{key}]$ and thus *key* does not leak, and thus no party except b and r can decrypt the response. As r never uses *key* to decrypt, only b might decrypt this encrypted HTTP response $resp'$ and obtain *resp*.

When processing the response *resp*, b may store *service* in its cookies such that $cookie \in^{\langle\rangle} S(b).\texttt{cookies}[dom]$ for $dom \equiv req.\texttt{host} \in \mathsf{dom}(r)$.

We see that b may only retrieve the cookie *cookie* and with it the nonce *service* when sending an encrypted HTTPS message m' for an HTTP request message m for a URL u with $u.\texttt{protocol} \equiv \texttt{S}$ (due to the secure flag) and the domain $m.\texttt{host} \equiv dom \in \mathsf{dom}(r)$. Note that when processing a script b does not give it access to *cookie* due to the httpOnly flag. Due to the arguments about the key mapping of b from above, that r does not leak the private TLS key $\mathsf{tlskey}(dom)$ and using the Browser HTTPS Lemma we get that only r can decrypt such messages m' and obtain m. When processing a message with a `Cookie` header in Algorithm B.5, r may only store a term containing *service* in Line 15 or 36 and does not send a message that may contain it. When retrieving the stored messages in Line 11 of Algorithm B.9 or Line 8 of Algorithm B.6, r never accesses the headers and thus never stores another term or sends a message that may contain *service*.

We get that no other process than r and b can derive *service* from its state and thus Theorem E.2.1 holds. □

E.3 Same-Device Session Integrity

For the proof of session integrity in this section we use that the `nonce` parameter in a flow between honest parties is not leaked (Lemma E.1.7). Using this property, and that session cookies cannot be injected via unencrypted connections due to the use of the `__Host` prefix in the cookie name, we can show that session integrity is not violated as the RP checks each request to the redirection endpoint for the nonce associated with the session cookie. Thus, if r issues a service session cookie for b associated with the `nonce` parameter, b must have started an authentication flow at r for some atomic DY process i. If i is an honest Self-Issued OP and the associated Configuration Provider is honest, then only i can respond with an ID token with the correct `nonce` parameter and for this must have chosen one of identities i holds to include in the token.

Theorem E.3.1 (Same-Device Session Integrity)
For every Same-Device Self-Issued OpenID Provider web system with a network attacker $\mathcal{SIOID}^n$, every run ρ of $\mathcal{SIOID}^n$, every processing step Q in ρ with

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

for some configurations (S, E, N) and (S', E', N'), every browser $b \in \mathsf{B}$ that is honest in S, every Same-Device Relying Party $r \in \mathsf{RP}$ that is honest in S, every $id \in \mathsf{SIID}$, every nonce sid, where $\mathsf{loggedIn}_\rho^Q(b, r, id, sid)$ holds, we have that there is an atomic DY process i such that

1. there is a processing step Q' in ρ before Q such that $\mathsf{started}_\rho^{Q'}(b, r, i, sid)$, and
2. if i is an honest Same-Device Self-Issued OP in S with $\mathsf{browser}(i) = b$ and $\mathsf{configProvider}(i)$ is an honest Configuration Provider in S and then there is a processing step Q'' in ρ before Q such that $\mathsf{choseIdentity}_\rho^{Q''}(r, id, i, sid)$.

Proof. By the definition of $\mathsf{loggedIn}_\rho^Q(b, r, id, sid)$, we have that r sent an encrypted HTTP response $resp' \equiv \mathsf{enc_a}(resp, key_0)$ for an encrypted HTTP request $req' \equiv \mathsf{enc_a}(\langle req, key_0\rangle, rk)$ that b sent to r before such that there is a cookie $c \equiv \langle\langle \texttt{\_\_Host}, \texttt{serviceSessionId}\rangle, \langle service, \top, \top, \top\rangle\rangle$ and $c \in^{\langle\rangle} resp.\texttt{headers}[\texttt{Set-Cookie}]$ for some nonce $service$, an HTTP request req, an HTTP response $resp$, a nonce key_0 and a term rk. We also get that there is a term $session$ such that $S(r).\texttt{sessions}[sid] \equiv session$, $session[\texttt{serviceSessionId}] \equiv service$, and $session[\texttt{loggedInAs}] \equiv id$.

The only point where the RP r might send such an encrypted HTTP response $resp'$, is in Algorithm B.9, Line 13, if it holds that $req \equiv session[\texttt{redirect\_ep\_req}][\texttt{message}]$ for the corresponding HTTP request req. We get that r must have set $S(r).\texttt{sessions}[sid][\texttt{redirect\_ep\_req}]$ before, which can only happen if Line 36 of Algorithm B.5 was executed before. We get from Line 22 that $req.\texttt{headers}[\texttt{Cookie}][\langle\texttt{\_\_Host}, \texttt{sessionId}\rangle] \equiv sid$ and also $host \equiv req.\texttt{host} \in \mathsf{dom}(r)$ (from the generic HTTPS server model main relation).

User started login flow. Note that $\mathsf{loggedIn}_\rho^Q(b, r, id, sid)$ requires that the browser b sent the encrypted HTTP request req'. For b to send req such that

$$req.\texttt{headers}[\texttt{Cookie}][\langle\texttt{\_\_Host}, \texttt{sessionId}\rangle] \equiv sid,$$

there must be a cookie c' for $req.\texttt{host}$ in the cookie component of the state of b with $c'.\texttt{name} \equiv \langle\texttt{\_\_Host}, \texttt{sessionId}\rangle$ and $c'.\texttt{content.value} \equiv sid$. From the browser model we get that b must have processed an HTTP response $cresp$ for an encrypted HTTP request $creq'$ (note the __Host prefix of the cookie name requiring the HTTPS protocol) for the domain $host \equiv req.\texttt{host} \equiv creq.\texttt{host} \in \mathsf{dom}(r)$ with $c' \in^{\langle\rangle} cresp.\texttt{headers}[\texttt{Set-Cookie}]$. In the initial state for all processes except r the private TLS key $\mathsf{tlskey}(host)$ only appears as the public key in the state, r never leaks $\mathsf{tlskey}(host)$ and by the definition of the initial state of b and the browser model definition, b uses $S(b).\texttt{keyMapping}[host] \equiv \mathsf{pub}(\mathsf{tlskey}(host))$ to encrypt the request as $creq' \equiv \mathsf{enc_a}(\langle creq, key_1\rangle, \mathsf{pub}(\mathsf{tlskey}(host)))$ for an HTTP request $creq$ and a nonce key_1. Also, r never leaks the key key_1, and thus by Browser HTTPS Lemma only r can create the response $cresp$.

The only point where r might create such an HTTP response $cresp$ with the header such that $c' \in^{\langle\rangle} cresp.\texttt{headers}[\texttt{Set-Cookie}]$ is in Algorithm B.13 if

$$S(r).\texttt{sessions}[sid][\texttt{startRequest}][\texttt{message}] \equiv creq$$

(Lines 57 to 60). From this we also get $c' \equiv \langle\langle\texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle sessionId, \top, \top, \top\rangle\rangle$ (Line 57). We see that r must have stored the request $creq$ before, and the only point where r may have done this is Line 15 of Algorithm B.5. We get that $creq.\texttt{headers}[\texttt{Origin}] \equiv \langle creq.\texttt{host}, \texttt{S}\rangle$ (Line 7), $creq.\texttt{method} \equiv \texttt{POST}$ (Line 6) and $creq.\texttt{path} \equiv \texttt{/startLogin}$ (Line 6).

We see that b must have set the origin header of *creq* to $\langle host, \mathtt{S}\rangle$, with $host \in \mathsf{dom}(r)$. Such an origin header b only includes in the request when processing a `FORM` or a `XMLHTTPREQUEST` command from a script, or when processing a redirect response where the previous request contained an origin header. When processing a script command, b retrieves $origin \equiv \langle host, \mathtt{S}\rangle \equiv S(b).\overline{d}.\mathtt{origin}$ for a document pointer $\overline{d}$. As again only r may know the private key $\mathsf{tlskey}(host)$ for $S(b).\mathtt{keyMapping}[host] \equiv \mathsf{pub}(\mathsf{tlskey}(host))$, and r never leaks the symmetric key for encrypting the response, we get from Lemma 2 of [10] that b extracted the script in the document d from an encrypted HTTP response that was created by r. The only scripts r might include in a response to b are *script_rp_index* and *script_rp_get_fragment*. As *script_rp_get_fragment* (Algorithm B.4) may only output a `FORM` command for URLs with the path $/\mathtt{redirect\_ep} \not\equiv creq.\mathtt{path}$, and r never responds to a request to the path `/redirect_ep` with a redirect, only *script_rp_index* might cause b sending such a request. Thus, b must have executed *script_rp_index* (Algorithm B.3) and chose a domain $dom \in \mathsf{dom}(i)$ for some atomic DY process i in Line 5 of the script. It follows that $\mathsf{started}_\rho^{Q'}(b, r, i, sid)$ for some processing step Q' before Q holds.

User chose identity at Self-Issued OP. For r to set $session[\mathtt{loggedInAs}] \equiv id$ in Algorithm B.9, the RP r must have executed Algorithm B.7 at least up to Line 38 before. Thus,

- $session[\mathtt{client\_id}] \in^{\langle\rangle} data[\mathtt{aud}]$ (Line 9),
- $pnonce \equiv session[\mathtt{nonce}] \equiv data[\mathtt{nonce}]$ and $data[\mathtt{nonce}] \not\equiv \langle\rangle$ (Line 36) and
- $data[\mathtt{sub}] \equiv id$ (passed as input in call of Algorithm B.9)

for $data \equiv \mathsf{extractmsg}(idt).\mathtt{token\_data}$ (Line 4) and $idt \equiv session[\mathtt{id\_token}]$ (Lines 2 and 3). Note that r might only set $client\_id \equiv session[\mathtt{client\_id}]$ such that $client\_id.\mathtt{host} \in \mathsf{dom}(r)$ and $client\_id.\mathtt{host} \equiv \mathtt{S}$ (Lines 11 and 33 to 38 of Algorithm B.13). We have that $session[\mathtt{id\_token}]$ must have been set before, and can only be set by r in Algorithm B.5, Line 37, where r stores the value $idt \equiv req.\mathtt{body}[\mathtt{id\_token}]$ for the HTTP request *req*. It follows that the term *idt* must have been created by some process before.

Note that we have by the above that $c' \equiv \langle\langle \mathtt{\_\_Host}, \mathtt{sessionId}\rangle, \langle sessionId, \top, \top, \top\rangle\rangle$ and $c' \in^{\langle\rangle} S(b).\mathtt{cookies}[host]$ for a domain $host \in \mathsf{dom}(r)$, $pnonce \equiv S(r).\mathtt{sessions}[sid][\mathtt{nonce}] \equiv data[\mathtt{nonce}]$ with $data \equiv \mathsf{extractmsg}(idt).\mathtt{token\_data}$. If the process i is an honest Self-Issued OP with $\mathsf{browser}(i) = b$, and i and $\mathsf{configProvider}(i)$ are honest up to S we have by Lemma E.1.7 that no other process than b, i and r can know *pnonce* as RP r and the browser b are honest up to S. As r and b do not create a message containing a term shaped like *idt*, it follows that only i can create a message containing *idt* that contains the nonce *pnonce*.

The only point where i may create a message containing the ID token term *idt* is in Algorithm B.17, Line 56. We get that i sent a response message *tresp* with $tresp.\mathtt{headers}[\mathtt{Location}] \equiv \mathsf{enc_a}(redirect, bkey)$ with $redirect.\mathtt{fragment}[\mathtt{id\_token}] \equiv idt$ and $bkey \equiv S(i).\mathtt{browserEncKey}$. Note that by the definition of the initial state s_0^i and that honest i never assigns to $S(i).\mathtt{browserEncKey}$, we get that $bkey \equiv \mathsf{publocalkey}(\mathsf{browser}(i))$. By the above, we have that $data \equiv \mathsf{extractmsg}(idt).\mathtt{token\_data}$ with $pnonce \equiv data[\mathtt{nonce}]$, $client\_id \in^{\langle\rangle} data[\mathtt{aud}]$, $client\_id.\mathtt{host} \in \mathsf{dom}(r)$, $client\_id.\mathtt{host} \equiv \mathtt{S}$, $data[\mathtt{sub}] \equiv id$. We get that $\mathsf{choseIdentity}_\rho^{Q''}(r, id, i, sid)$ for a processing step Q'' before Q. Theorem E.3.1 holds. □

E.4 Same-Device Holder Binding

For the proof of Holder Binding we use that RP if Self-Issued OP and browser for a Verifiable Credential are honest, no other process might learn a Verifiable Presentation that the RP accepts (Lemma E.1.8) and Self-Issued OP and browser for a Self-Issued Identity are honest, no other process might learn an ID token that the RP accepts (Lemma E.1.2 and Lemma E.1.4). We then show that if an honest RP accepts a Verifiable Credential or a Self-Issued Identity into a session and the Self-Issued OP that is the holder of either is honest, then the same (honest) Self-Issued OP must have sent both via redirection in an honest browser. As an honest Self-Issued OP only creates ID tokens and Verifiable Presentations for Self-Issued Identity or Verifiable Credentials respectively of which the Self-Issued OP is the holder, it must be the holder of both.

Theorem E.4.1 (Same-Device Holder Binding)
For every Same-Device Self-Issued OpenID Provider web system with a network attacker $\mathcal{SIOID}^n$ for every run ρ of $\mathcal{SIOID}^n$, for every configuration (S, E, N) in ρ, every RP $r \in \mathsf{RP}$ that is honest in S, every Self-Issued Identity $id \in \mathsf{SIID}$, every service session identified by some nonce *service* for the Self-Issued Identity *id* at *r* where if $id \in \mathsf{DID}$ then $\mathsf{govenor}(id)$ is an honest VDR in S, every Verifiable Credential $vc \in \mathsf{VC}$, every service session identified by some nonce *service* with the Verifiable Credential *vc* at *r* such that

- $\mathsf{id\_holder}(id) = i$ is an honest Same-Device Self-Issued OP in S and $\mathsf{browser}(i)$ is an honest browser in S, or
- $\mathsf{vc\_holder}(vc) = y$ is an honest Same-Device Self-Issued OP in S and $\mathsf{browser}(y)$ is an honest browser in S,

it holds that $i = y$.

Proof. Let *r* be an RP that is honest up to S, $id \in \mathsf{SIID}$ with $\mathsf{id\_holder}(id) = i$ and $vc \in \mathsf{VC}$ with $\mathsf{vc\_holder}(vc) = y$. If we have a service session identified by some nonce *service* for the Self-Issued Identity *id* at an RP *r* and a service session identified by the nonce *service* with the Verifiable Credential *vc* at *r* we have that there is a term *sid* with

- $S(r).\mathtt{sessions}[sid][\mathtt{serviceSessionId}] \equiv service$,
- $S(r).\mathtt{sessions}[sid][\mathtt{credential}] \equiv vc$ and
- $S(r).\mathtt{sessions}[sid][\mathtt{loggendInAs}] \equiv id$

by definition. It follows that *r* must have executed Algorithm B.9 that contains the only points where *r* might set these values. We see that *r* must have invoked Algorithm B.9 from Algorithm B.8 (as $vc \not\equiv \bot$), and thus also executed Algorithm B.7 to invoke Algorithm B.8 .

As Algorithm B.8 was executed, we have that *r* must have set $S(r).\mathtt{sessions}[sid][\mathtt{vp\_token}] \equiv vp$ (Lines 2 and 6) for a term *vp* before with

- $\mathsf{extractmsg}(vp)[\mathtt{domain}] \equiv S(r).\mathtt{sessions}[sid][\mathtt{client\_id}]$ (Line 11) and
- $\mathsf{extractmsg}(vp)[\mathtt{vc}] \equiv vc$ (Line 14).

As the only point where r might have set $S(r).\texttt{sessions}[\texttt{client\_id}]$ is in Lines 33 to 38 of Algorithm B.13, it follows that $\mathsf{extractmsg}(vp)[\texttt{domain}] \equiv \langle \texttt{URL}, \texttt{S}, domain, path, \langle\rangle, \bot\rangle$ for $domain \in \mathsf{dom}(r)$ and a term $path$. We also get from Algorithm B.8, Lines 15, 16 and 20, that $\mathsf{checksig}(vp, \mathsf{extractmsg}(vc)[\texttt{subject}]) \equiv \top$. As $\mathsf{vc\_holder}(vc) \equiv y$ it must be that $\mathsf{extractmsg}(vc)[\texttt{subject}] \equiv \mathsf{pub}(k)$ for $k \in K^{\mathsf{VP}}_{\mathrm{sign}}$ with $\mathsf{key\_holder}(k) = y$.

As Algorithm B.7 was executed, where r retrieves an ID token term idt we have that the RP r must have also set $S(r).\texttt{sessions}[sid][\texttt{id\_token}] \equiv idt$ (Lines 2 and 3) with $data \equiv \mathsf{extractmsg}(idt).\texttt{token\_data}$ (Line 4) and $S(r).\texttt{sessions}[sid][\texttt{client\_id}] \in^{\langle\rangle} data[\texttt{aud}]$ (Line 9). As for vp above it follows that $\langle \texttt{URL}, \texttt{S}, domain, path, \langle\rangle, \bot\rangle \in^{\langle\rangle} data[\texttt{aud}]$ for $domain \in \mathsf{dom}(r)$ and a term $path$. By Lemma E.1.4 we have that $idt \equiv \mathsf{sig}(t', k)$ for a term t' and $k \in \mathsf{proof\_key}(id)$.

The only point where r might have set such values $S(r).\texttt{sessions}[sid][\texttt{vp\_token}]$ and $S(r).\texttt{sessions}[sid][\texttt{id\_token}]$ is in Algorithm B.5 when r starts processing a message m_t from Line 21 and executes Lines 37 and 39. For r to store to the session identified by the term sid, r must process a message m_t with a matching cookie in the headers i.e., we get from Line 22 that

$$m_t.\texttt{headers}[\texttt{Cookie}]\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle \equiv sid,$$

and from Line 31 that $m_t.\texttt{method} \equiv \texttt{POST}$, a dictionary in the body such that $m_t.\texttt{body}[\texttt{id\_token}] \equiv idt$ and $m_t.\texttt{body}[\texttt{vp\_token}] \equiv vp$.

Let i be a Self-Issued OP that is honest up to S such that $\mathsf{browser}(i) = b$ is honest up to S and $\mathsf{id\_holder}(id) = i$ or $\mathsf{vc\_holder}(vc) = i$. If $i = \mathsf{id\_holder}(id)$ we get using Lemma E.1.2 that only r, i and b can know a term $idt \equiv \mathsf{sig}(z, k)$ for a term z and $k \in \mathsf{proof\_key}(id)$ with $data \equiv z.\texttt{token\_data}$ and $data[\texttt{aud}].\texttt{host} \in \mathsf{dom}(r)$, $data[\texttt{aud}].\texttt{protocol} \equiv \texttt{S}$, and $data[\texttt{sub}] \equiv id$ for the case where $\mathsf{id\_holder}(id) = i$. For the case $\mathsf{vc\_holder}(vc) = i$ we have that only i, b and r might learn a term vp with $vp \equiv \mathsf{sig}(z, k)$ for $k \in K^{\mathsf{VP}}_{\mathrm{sign}}$ and $z[\texttt{domain}].\texttt{host} \in \mathsf{dom}(r)$, $z[\texttt{domain}].\texttt{protocol} \equiv \texttt{S}$, $z[\texttt{vc}] \equiv vc$ and $vc[\texttt{subject}] \equiv \mathsf{pub}(k)$ by Lemma E.1.8. Thus, only b, r and i could have created the message m_t containing both the ID token idt and the Verifiable Presentation vp.

As i and r never send a message with a cookie header, b must have sent the request message m_t. By the definition of b we have that b only might send a message m_t with $m_t.\texttt{method} \equiv \texttt{POST}$ if b processes a 307 redirect, or if b processes a script command `FORM` or `XMLHTTPREQUEST`. As the POST request resulting from a 307 redirect would need to contain the idt and vp in the body, there then must have been a request before containing idt and vp in the body. As only b, i and r may know both idt and vp and never send a 307 redirect this cannot be the case. Thus, b must have processed a script resulting in a command `FORM` or `XMLHTTPREQUEST` that initiated the request message m. The attacker cannot derive both idt and vp due to the above and thus the attacker script can not initiate this request. We have that only the two scripts *script_rp_index* and *script_rp_get_fragment* can result in b sending the request m_t. The script *script_rp_index* (Algorithm B.3) cannot result in m_t as the form command it may output contains a body that is independent of possible inputs. Thus, only the output of script *script_rp_get_fragment* may have resulted in b sending the request m_t. As $m_t.\texttt{host} \in \mathsf{dom}(r)$ and m_t must have been sent using HTTPS (else r would reject the request by the generic HTTPS server model main relation), we see that the document location of the script must be for a domain of r that was set when r send the script in a previous response. The only point where r may send the script string for *script_rp_get_fragment* is Line 29 of Algorithm B.5. We see that b must have sent a request $m'_r \equiv \mathsf{enc_a}(\langle m_r, key_0\rangle, rkey)$ for an HTTP message

m_r and a nonce key_0 with $m_r.\texttt{method} \equiv \texttt{GET}$ (Line 27), $m_r.\texttt{path} \equiv \texttt{/redirect\_ep}$ (Line 21), $m_r.\texttt{host} \equiv m_t.\texttt{host} \in \mathsf{dom}(r)$ and $rkey \equiv S(b).\texttt{keyMappig}[m_r.\texttt{host}] \equiv \mathsf{pub}(\mathsf{tlskey}(m_r.\texttt{host}))$ by the defintion of the initial state and that b never assigns to the key mapping.

For b to send such a request m'_r with $m_r.\texttt{method} \equiv \texttt{GET}$, b must have either executed a script for a URL with *idt* and *vc* in the fragment or received a redirect response. As the attacker cannot derive both *idt* and *vp*, the attacker script can not result in such a GET request and the attacker cannot send the redirect response. We also have that *script_rp_get_fragment* (Algorithm B.4) only outputs a `FORM` command for a `POST` request. If *script_rp_index* is executed only the `HREF` command may result in a `GET` request. Then however the fragment is set to some string and cannot contain terms of the structure as *idt* and *vp*.

Thus, for b to make such a request m'_r for a URL *url* with *idt* and *vp* contained in $url.\texttt{fragment}$ with $url.\texttt{host} \equiv m_r.\texttt{host}$ and $url.\texttt{protocol} \equiv \texttt{S}$, b must have received a response m_i with a location header from i, b or r. The only points where one of the processes might send a response with a location header are for r in Algorithm B.13 and for i in Algorithm B.17. Note, however that r in Algorithm B.13 does not add value under the keys `id_token` or *vp_token* to the URL in the redirect. We have that i must have created the message m_i with the location header containing both *idt* and *vp* to b.

As i is honest in S we have that the only point where i might include the ID token for identity *id* and a Verifiable Presentation *vp* with $\mathsf{extractmsg}(vp)[\texttt{vc}] \equiv vc$ in a message is Lines 48 and 51 to Algorithm B.17. In this algorithm i retrieves *vc* in Line 45 such that $record \in^{\langle\rangle} S(i).\texttt{credentials}$ and $record[\texttt{vc}] \equiv vc$. As i is honest in S and i never assigns to the credential component of the state, we have that $S(i).\texttt{credentials} \equiv s_0^i.\texttt{credentials} \equiv \langle \mathsf{VCW} \rangle^i$. From $record \in^{\langle\rangle} \langle \mathsf{VCW} \rangle^i$ with $record[\texttt{vc}] \equiv vc$ we get that $\mathsf{vc\_holder}(vc) \equiv i$. We also have that i only might include *idt* under the key `id_token` in a dictionary in Line 51 of Algorithm B.17. From Lines 32 and 50 we get that $\mathsf{extractmsg}(idt).\texttt{tokendata}[\texttt{sub}] \equiv id$ with $id \equiv id\_record[\texttt{id}]$ for a term $id\_record \in^{\langle\rangle} S(i).\texttt{identities}$ (Lines 23 to 28) with $S(i).\texttt{identities} \equiv s_0^i.\texttt{identities} \equiv \langle \mathsf{ID\_Records}^i \rangle$ as i never assigns to the identities component of the state. By the definition of $\mathsf{ID\_Records}^i$ we get that $id \in \mathsf{SIID}^i$ and thus $\mathsf{id\_holder}(id) = i$.

Thus, we have $i = \mathsf{id\_holder}(id) = \mathsf{vc\_holder}(vc)$. □

E.5 General Properties of Cross-Device Self-Issued OPs

In this section we show some general properties of the Cross-Device model, before we use them to show that the authentication property holds in the following section.

Lemma E.5.1 (Relying Party Cookie not leaked)
Let $\chi SIOI\mathcal{D}^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\chi SIOI\mathcal{D}^n$, every configuration (S, E, N) in ρ, every $r \in \mathsf{xRP}$ that is honest in S, every $b \in \mathsf{xB}$ that is an honest browser in S, every atomic DY process $p \in \mathcal{W} \setminus \{b, r\}$, every $h \in \mathsf{dom}(r)$, every term *sid*, with $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$ such that $c.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId} \rangle$ and $c.\texttt{content}.\texttt{value} \equiv sid$, we have that $sid \notin d_\emptyset(S(p))$.

Proof. If we have a cookie $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$ such that $c.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ and $c.\texttt{content.value} \equiv sid$ with $h \in \mathsf{dom}(r)$ for a browser b and an RP r that are honest in S, we have that b must have stored the cookie in its state at some point prior to S. The only point where b might store c in its cookies is Line 4 of the algorithm PROCESSRESPONSE of the browser model when processing the response *resp* for some request *req* for the domain $req.\texttt{host} \equiv h$ with $c \in^{\langle\rangle} resp.\texttt{headers}[\texttt{Set-Cookie}]$. We get that the request *req* must have been contained in an encrypted HTTP request *req*′ due to $c.\texttt{name}.1 \equiv \texttt{\_\_Host}$ and the definition of AddCookie used. For encrypting such a request *req*, b uses the public key for h in its key mapping, that is $S(b).\texttt{keyMapping} \equiv \mathsf{pub}(\mathsf{tlskey}(h))$, by the definition of the initial state of b and that the honest browser b never modifies this key mapping i.e., $req' \equiv \mathsf{enc_a}(\langle req, key_0\rangle, \mathsf{pub}(\mathsf{tlskey}(h)))$ for a nonce key_0. Only r knows this key $\mathsf{tlskey}(h)$ in the initial state and never leaks it or the symmetric key key_0 for the response that b sends with the request. Using the Browser HTTPS Lemma, we see that r must have created the response *resp*.

The only point where r might include a `Set-Cookie` header for a cookie with name $\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ to form such a response *resp* for *req* is in Lines 39 to 41 of Algorithm C.5. From this we get

$$\langle c\rangle \equiv resp.\texttt{headers}[\texttt{Set-Cookie}] \equiv \langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle sid, \top, \top, \top\rangle\rangle.$$

Note that before executing Algorithm C.5, r must have executed Lines 20 and 21 of Algorithm C.4 to initiate *sid* with a fresh nonce (*sid* cannot be trivially derived). The message *resp* is the only one where r might include the nonce *sid*.

On receiving this response message *resp*, b stores the cookie c containing the nonce *sid* in $S(b).\texttt{cookies}[h]$ (as above) and may store the `Set-Cookie` header in a document in its state (Line 35 of PROCESSRESPONSE) but in the latter case the `Set-Cookie` header is not accessed again as the document is only passed to scripts after the headers are removed (using the Clean algorithm from the browser model).

The only points where b might retrieve c from $S(b).\texttt{cookies}[h]$ is Line 4 of HTTP_SEND when sending a message m with $m.\texttt{host} \equiv h$ using HTTPS as $c.\texttt{content.secure} \equiv \top$. Note that Line 3 of RUNSCRIPT is not possible, since $c.\texttt{content.httpOnly} \equiv \top$. Following Line 4 of the browser algorithm HTTP_SEND, b includes $sid \equiv c.\texttt{content.value}$ in a cookie header for the name $c.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$. This message m is again encrypted using the key $\mathsf{pub}(\mathsf{tlskey}(h))$ from the key mapping in the state of b, and only r can decrypt such messages as only r knows $\mathsf{tlskey}(h)$. The only point where r stores a term containing *sid* is in Line 21 where r stores m to $S(r).\texttt{sessions}[sid'][\texttt{startRequest}][\texttt{message}]$, in other places r only uses *sid* to retrieve values from its state, but never stores or sends it. The only point where r might retrieve $m \equiv S(r).\texttt{sessions}[sid'][\texttt{startRequest}][\texttt{message}]$ is in Lines 40 and 41, but there r only uses $m.\texttt{nonce}$ and not $m.\texttt{headers}$ that may contain *sid*.

Thus, only b and r may know *sid*. □

Lemma E.5.2 (Stub Tokens do not leak)
Let $\chi\mathcal{SIOID}^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\chi\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every $i \in \mathsf{xSIOP}$ that is honest in S where all $b \in \mathsf{userBrowser}(i)$ are honest browsers in S and $u = \mathsf{trustedStub}(i)$ is an honest Cross-Device Stub in S, every $t \in \mathsf{StubToken}^i$, every atomic DY process $p \in \mathcal{W}\backslash(\{i, u\} \cup \mathsf{userBrowser}(i))$ we have that $t \notin d_\emptyset(S(p))$.

Proof. Note that in the initial state $t \in \mathsf{StubToken}^i$ may only be known to the Cross-Device Self-Issued OP i contained in $s_0^i.\texttt{stub\_tokens} \equiv \langle \mathsf{StubToken}^i \rangle$ and browsers $b \in \mathsf{userBrowser}(i)$ contained in $s_0^b.\texttt{secrets} \equiv \langle \mathsf{Secrets}^b \rangle$ with

$$\mathsf{Secrets}^b \equiv \{\langle\langle d, \mathtt{S}\rangle, t\rangle \mid i = \mathsf{user}(b), d \in \mathsf{dom}(u), t \equiv \mathsf{browserStubToken}^i(b)\}$$

where $u = \mathsf{trustedStub}(i)$ and $\mathsf{browserStubToken}^i(b) \in \mathsf{StubToken}^i$ by definition of the initial states of the processes in the web system $\chi\mathcal{SIOID}^n$. The only point where i might retrieve a stub token $t \in \mathsf{StubToken}^i$ contained in $S(i).\texttt{stub\_tokens}$ from its state is Line 9 of Algorithm C.12. There, i does not store t anywhere else in its state, and does not send a message that contains t.

We know check how a browser b may access $S(b).\texttt{secrets}$ for states S and see that b only retrieves the secrets components from its state in Line 7 of Algorithm C.1 (and might set $S(b).\texttt{secrets} \equiv \langle\rangle$ in Line 47 of the browser main algorithm). In Line 7 of Algorithm C.1, b retrieves $\mathit{secrets} \equiv S(b).\texttt{secrets}[o]$ for $o \equiv S(b).\overline{d}.\texttt{origin}$ for some document pointer term $\overline{d}$. As we only care for the case where $t \in \mathsf{StubToken}^i$ is contained in *secrets*, we have that $S(b).\overline{d}.\texttt{origin} \equiv o \equiv \langle \mathit{domain}, \mathtt{S}\rangle$ for $\mathit{domain} \in \mathsf{dom}(u)$ with $u = \mathsf{trustedStub}(i)$ by the above as an honest browser b does not assign new value to its secrets.

Note that RUNSCRIPT is only called in Line 12 of the browser main algorithm. From this we get $\overline{d} \equiv \overline{w} +^{\langle\rangle} \texttt{activedocument}$ for some $\overline{w}$ such that $S(b).\overline{w} = w$ and $w \in^{\langle\rangle} S(b).\texttt{windows}$ (Line 10 and 11 of the browser main algorithm and the definition of $\mathsf{Subwindows}(S(b))$). As $k \equiv S(b).\texttt{keyMapping}[\mathit{domain}] \equiv s_0^b.\texttt{keyMapping} \equiv \langle \mathsf{keyMapping} \rangle$ and thus $k \equiv \mathsf{pub}(\mathsf{tlskey}(\mathit{domain}))$ and only u knows $\mathsf{tlskey}(\mathit{domain})$ (as $\mathit{domain} \in \mathsf{dom}(u)$) in the initial state, u never leaks k and also neither b nor u leak the symmetric key key_1 for encrypting the response, we have that u must have created the encrypted HTTP response $\mathit{sresp}' \equiv \mathsf{enc_s}(\mathit{sresp}, \mathit{key}_1)$ from which b extracted the script in the document $d \equiv S(b).\overline{d}$ in Line 35 of the PROCESSRESPONSE algorithm by Lemma 2 of [10] and stored the document d to its state.

Note that the only scripts that u might send contained in an HTTP response message *sresp* are *script_stub_token_form* and *script_render_qr*. If u sent *script_render_qr*, then the outputs *out* of applying the script relation are determined independently of *secrets* that contains t (Algorithm C.7 and Algorithm C.1). In this case b neither sends a message that may contain t or stores t to another part of its state.

In the case where the Cross-Device Stub u sent the script string for the script *script_stub_token_form* (Algorithm C.8), we have that the browser b may retrieve a stub token t from *secrets* and obtains a command term $\langle \texttt{FORM}, \mathit{url}, \texttt{POST}, \mathit{formData}, \bot\rangle$ with $\mathit{formData}[\texttt{stub\_token}] \equiv t$ and $\mathit{url} \equiv \langle \texttt{URL}, \mathtt{S}, \mathit{domain}, /\texttt{token}, \langle\rangle, \langle\rangle\rangle$ (Algorithm C.8 and Algorithm C.1) as we have that $\mathit{domain} \equiv S(b).\overline{d}.\texttt{location.host}$ by the definition of the function GETURL. Note that no other components of the output of applying the script relation may contain t. In Line 51 of Algorithm C.1 we have that b then assembles an HTTP request req_t of the shape $\mathit{req}_t \equiv \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \mathit{domain}, /\texttt{token}, \langle\rangle, \langle\rangle, \mathit{formData}\rangle$ for a nonce n_1. Note that as the method is POST we have that b sets *formData* as the body of the request and that b includes an origin header with value $S(b).\overline{d}.\texttt{origin} \equiv o \equiv \langle \mathit{domain}, \mathtt{S}\rangle$ (Line 17 and 49 of Algorithm C.1 and Line 7 of HTTP_SEND). We also see that b sets the reference *ref* for this request req_t such that $\mathit{ref}.1 \equiv \texttt{REQ}$.

Then, b sends this request req_t as an encrypted HTTP request $\mathsf{enc_a}(\langle req_t, key_2\rangle, k)$ using $k \equiv \mathsf{pub}(\mathsf{tlskey}(domain))$ for encryption (by the same argument as above) and again only u knows k and can decrypt this request to obtain req_t and the nonce key_2 for encrypting the response. The Cross-Device Stub u then proceeds to process req_t starting at Line 31 of Algorithm C.9 as $req_t.\texttt{path} \equiv \texttt{/token}$ and $req_t.\texttt{headers}[\texttt{Origin}] \equiv \langle domain, \texttt{S}\rangle \equiv \langle req_t.\texttt{host}, \texttt{S}\rangle$.

There u retrieves $t \equiv req_t.\texttt{body}[\texttt{stub\_token}]$ (Line 38), adds t to a term *stub_req* as $stub\_req.\texttt{parameters} \equiv stub\_req.\texttt{parameters} \cup [\texttt{stub\_token} : t]$ (Line 39) and sends an encrypted HTTP response message $resp'_t \equiv \mathsf{enc_a}(resp_t, key_2)$ with script string `script_render_qr` and initial script state *stub_req* in the body $resp_t.\texttt{body}$ of the HTTP response $resp_t$. As neither b nor u leak key_2 to another process, and u does not store key_2 in its state, only b might decrypt $resp'_t$. The honest browser b may only decrypt using this key in the browser main algorithm and b calls PROCESSRESPONSE with the HTTP response $resp_t$.

Note that in the browser algorithm PROCESSRESPONSE the browser b might only access the response body containing t in Lines 31, 33, 34 and 47. As the response however is for the request req_t with reference such that $ref.1 \equiv \texttt{REQ}$, Line 47 cannot be executed as then $ref.1 \equiv \texttt{XHR}$ must have been the case (Line 28 and 44). In Line 31 b only checks that the body is of a certain shape, and then retrieves `script_render_qr` in Line 33 and *stub_req* in Line 34. In Line 35, b forms a document with `script_render_qr` as the script string and *stub_req* as the script state and adds the document to some window in its state.

The only point where honest b might retrieve the script state *stub_req* in some document d is in Line 9 of Algorithm C.1, together with the relation for the string $d.\texttt{script} \equiv \texttt{script\_render\_qr}$ in the previous line. In the following lines b applies the relation of the script and obtains the command $command \equiv \langle \texttt{ShowQR}, stub\_req\rangle$ — no other component than *stub_req* of the output may contain t.

The browser b then processes this command *command* starting at Line 95 of Algorithm C.1. There b sends a message $m_q \equiv \mathsf{enc_a}(qr, key)$ (Line 98) with $qr \equiv \langle \texttt{QR}, stub\_req\rangle$ (Line 96) and *key* such that $\langle addr, key\rangle \in^{\langle\rangle} S(b).\texttt{qrchannels} \equiv s_0^b.\texttt{qrchannels}$ (Line 97). It follows that $key \equiv \mathsf{pub}(\mathsf{qrkey}(\mathsf{user}(b)))$ by the definition of $s_0^b.\texttt{qrchannels}$ and since $b \in \mathsf{userBrowser}(i)$ we have that $\mathsf{user}(b) = i$ and thus $key \equiv \mathsf{pub}(\mathsf{qrkey}(i))$.

In the initial state only the Cross-Device Self-Issued OP i (that is honest in S) knows the nonce $\mathsf{qrkey}(i) \in K_{\mathrm{QR}} \subset \mathcal{N}$ and i never leaks this nonce. Thus, only i might decrypt this message and retrieve *stub_req*. By definition of i we get that i may only decrypt using $\mathsf{qrkey}(i) \equiv S(i).\texttt{qrkey}$ in Lines 2 and 3 of Algorithm C.11. There, i passes *stub_req* to Algorithm C.12 as the input *content*. Remember that *stub_req* is a term such that $stub\_req.\texttt{parameters}[\texttt{stub\_token}] \equiv t$ and t is contained in no other component in the term. In Algorithm C.12, the Cross-Device Self-Issued OP i retrieves t to check against components of its own state (Lines 5 and 9) and stores $data \equiv stub\_req.\texttt{parameters}$ in $S(i).\texttt{sessions}[sid][\texttt{params}]$ (Lines 5 and 20). The Cross-Device Self-Issued OP i does not store t in another component of its state and does not send a message containing t in Algorithm C.12. The only point where i might retrieve $data \equiv S(i).\texttt{sessions}[sid][\texttt{params}]$ is Line 2 of Algorithm C.13. There, i never accesses the key `stub_token` of *data* and never includes the full dictionary term *data* in a message or stores it to its state.

We get that i does not emit a message that contains t and with this we get that only $i, u = \mathsf{trustedStub}(i)$ and $b \in \mathsf{userBrowser}(i)$ can learn a stub token $t \in \mathsf{StubToken}^i$. □

Lemma E.5.3 (Stub State does not leak)
Let $\chi\mathcal{SIOID}^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\chi\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every $b \in \mathsf{xB}$ that is an honest browser in S, every $r \in \mathsf{xRP}$ that is an honest RP in S, every $u \in \mathsf{xStub}$ that is an honest Cross-Device Stub in S, every domain $h \in \mathsf{dom}(u)$, every atomic DY process $p \in \mathcal{W} \setminus \{b, r, u\}$, every nonce *usid*, every nonce *state*, every cookie c, with

- $u.\texttt{sessions}[usid][\texttt{state}] \equiv state$
- $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$, $c.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ and $c.\texttt{content.value} \equiv usid$ and
- $S(u).\texttt{sessions}[usid][\texttt{request}].\texttt{parameters}[\texttt{redirect\_uri}] \equiv redirect\_uri \in \mathsf{URLs}$ with $redirect\_uri.\texttt{host} \in \mathsf{dom}(r)$ and $redirect\_uri.\texttt{protocol} \equiv \mathsf{S}$,

we have that $state \notin d_\emptyset(S(p))$.

Proof. For $S(u).\texttt{sessions}[usid][\texttt{state}] \equiv state$ to hold for a nonce *state* and a term *usid* we see that the Cross-Device Stub u must have set this value first. Thus, we have that u must have executed Line 7 of Algorithm C.9 where u chooses a fresh nonce for *state* and also for *usid*. We follow the path of the nonce *state* in the protocol flow.

The Cross-Device Stub u only retrieves *state* from $u.\texttt{sessions}[usid][\texttt{state}]$ in Line 25 and does not store it elsewhere or send a message including it. The only other use of the nonce *state* by u is in the `Location` header of the HTTP response message m sent in Line 16 of Algorithm C.9 that contains a URL *redirect_uri* with $redirect\_uri.\texttt{parameters}[\texttt{cstub}] \equiv state$. This message m is also the only one with which u might send *usid*, which u does in the header as $m.\texttt{headers}[\texttt{Set-Cookie}] \equiv \langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle usid, \top, \top, \top\rangle\rangle$. Note that $u.\texttt{sessions}[usid][\texttt{request}].\texttt{parameters}[\texttt{redirect\_uri}] \equiv redirect\_uri$ with $redirect\_uri.\texttt{host} \in \mathsf{dom}(r)$ and $redirect\_uri.\texttt{protocol} \equiv \mathsf{S}$ and

$$redirect\_uri.\texttt{parameters}[\texttt{stub\_fin}] \equiv \langle \texttt{URL}, \mathsf{S}, d, \texttt{/stub/fin}, \langle\rangle, \bot\rangle$$

for a domain $d \in \mathsf{dom}(u)$.

Note that we have that there is a cookie c that is stored by the browser b as $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$ with $c.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ and $c.\texttt{content.value} \equiv usid$ for a domain $h \in \mathsf{dom}(u)$. For b to store this cookie in its state, it must have processed a `Set-Cookie` header in a response message for a request using the protocol HTTPS (note the `__Host` cookie name prefix) for the domain h and added the cookie c. As only u knows $\mathsf{tlskey}(h)$, for which b uses the public key $\mathsf{pub}(\mathsf{tlskey}(h) \equiv S(b).\texttt{keyMapping}[h]$, to encrypt the request in the initial state and never leaks it, and u does leak not leak the symmetric key key_1 for encrypting the response to another process, we get from the Browser HTTPS Lemma that u must have created the response message. The only point where u sends such an encrypted response message is, when u sends the encrypted HTTP response message $m' \equiv \mathsf{enc_a}(m, key_1)$. Only b and u know the symmetric key to decrypt this message, but u never processes a response, thus only b may process this response message m.

When the browser b processes the response message m, b retrieves $m.\texttt{headers}[\texttt{Location}] \equiv redirect\_uri$ and sends an encrypted HTTP request $m'_r \equiv \mathsf{enc_a}(\langle m_r, key_2\rangle, rkey)$ for a nonce key_2 with $dom \equiv m_r.\texttt{host} \equiv redirect\_uri.\texttt{host} \in \mathsf{dom}(r)$, $m_r.\texttt{parameters} \equiv redirect\_uri.\texttt{parameters}$, $redirect\_uri.\texttt{protocol} \equiv \mathsf{S}$ and $rkey \equiv \mathsf{pub}(\mathsf{tlskey}(dom))$ as b is

honest in S. Only r can decrypt this message and obtain m_r, we get using the Browser HTTPS Lemma as r is the only process that knows $\mathsf{tlskey}(dom)$ in the initial state and never leaks this nonce as r is honest in S.

On receiving the request m'_r, r processes the request m_r starting at Line 2, 12, 23 or 25 of Algorithm C.4. When r starts processing at Line 2 or 23 we see that r never accesses $m_r.\texttt{parameters[cstub]}$ or stores the value. When processing the request from Line 12 onward, we see that r stores the message m_r in Line 21. This request message m_r might then be retrieved in Line 40 of Algorithm C.5 where r again does not access the parameter `cstub`. It remains the case where r processes m_r starting from 25 of Algorithm C.4. In the case where r does not invoke Algorithm C.2 with m_r as input, $m_r.\texttt{parameters}$ is not accessed or stored. If m_r is passed to Algorithm C.2, r retrieves $state \equiv m_r.\texttt{parameters[cstub]}$ and sends an encrypted HTTP response message $m'_s \equiv \mathsf{enc_s}(m_s, key_2)$ (Line 16) with m_s an HTTP response with $m_s.\texttt{headers[Location]} \equiv stub\_fin'$ for a URL $stub\_fin' \equiv \langle \texttt{URL}, \texttt{S}, d, \texttt{/stub/fin}, params, \bot \rangle$ for $d \in \mathsf{dom}(u)$ ($m_s.\texttt{parameters[stub\_fin]}$ with modified parameters, see Lines 10 to 14) and a term $params$ with $params[\texttt{cstub}] \equiv m_r.\texttt{parameters[cstub]} \equiv state$. As only b and r know key_2 (as the processes are honest in S they never leak the symmetric key key_2), only b might decrypt the response (by the definition of the generic HTTPS server model main relation r will not use this key to decrypt a response message).

When the browser b processes the response message m_s it retrieves the `Location` header and sends an encrypted HTTP request m'_f. As the header $m_s.\texttt{headers[Location]}$ contains the URL $stub\_fin'$ with $d \equiv stub\_fin'.\texttt{host} \in \mathsf{dom}(u)$ and $stub\_fin'.\texttt{protocol} \equiv \texttt{S}$, we see that b sends an encrypted HTTP request $m'_f \equiv \mathsf{enc_a}(\langle m_f, key_3 \rangle, ukey\mathsf{pub}(\mathsf{tlskey}(d)))$ for a nonce key_3 with $m_f.\texttt{host} \equiv d$, $m_f.\texttt{path} \equiv stub\_fin'.\texttt{path} \equiv \texttt{/stub/fin}$ where $state$ is only contained in $m_f.\texttt{parameters[cstub]}$. Only u can decrypt this message as u is the only process that knows the private key $\mathsf{tlskey}(d)$ in the initial state and never leaks $\mathsf{tlskey}(d)$, and b retrieves the public key $ukey \equiv S(b).\texttt{keyMapping}[d] \equiv \mathsf{pub}(\mathsf{tlskey}(d))$ from its key mapping. The Cross-Device Stub u may only process this request m_f starting at Line 17 of Algorithm C.9 as $m_f.\texttt{path} \equiv \texttt{/stub/fin}$. There, u does not store $m_f.\texttt{parameters[cstub]} \equiv state$ or send a message containing $state$ and only sends a script in the response for which u also sets the `ReferrerPolicy` header to `origin`. Note that using the Browser HTTPS Lemma we get that only u may provide this response as u never leaks the symmetric key key_3 for the response contained in m_f to another process.

The browser b, on receiving this response might store the request URL $stub\_fin'$ (including $state$ in the parameters) as the location in a document in its state, together with the headers from the response (Line 35 of PROCESSRESPONSE of the browser model). This location b might retrieve in Algorithm C.1, Lines 2 or 15, or when choosing reload when b processes a `TRIGGER` message (in the browser main relation). When processing reload, b may retrieve $stub\_fin'$ from the document location and send an encrypted request like m'_f for the URL $stub\_fin'$, for which again the above applies. When b retrieves the location of a document in Line 15 of Algorithm C.1, b also retrieves the headers of the document (Line 16) that contain the header `ReferrerPolicy` set to `origin`. Thus, b does not include the parameters of $stub\_fin'$ in any `Referer` header that b might send and $state$ does not leak in this case. When the cleaned tree from Line 2 of Algorithm C.1 may contain the location URL $stub\_fin'$ only when b processes a script for a document with origin for a domain of u and the HTTPS protocol. However, by Lemma 2 from [10] we get that only u can provide this

script in an HTTPS response. The only scripts u may provide are *script_render_qr* (Algorithm C.7) and *script_stub_token_form* (Algorithm C.8). When processing these scripts b does not retrieve the parameters of the document of location that may contain *state*.

Thus, we have $state \notin d_\emptyset(S(p))$ for any atomic DY process $p \in \mathcal{W} \setminus \{b, r, u\}$. □

Lemma E.5.4 (Session Sync)
Let $\chi SIOID^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\chi SIOID^n$, every configuration (S, E, N) in ρ, every $b \in \mathsf{xB}$ that is an honest browser in S, every $r \in \mathsf{xRP}$ that is an honest RP in S every $u \in \mathsf{xStub}$ that is an honest Cross-Device Stub in S, every domain $h \in \mathsf{dom}(u)$, every nonce *usid*, every term *pnonce*, every cookie c, such that

- $c \in^{\langle\rangle} S(b).\mathtt{cookies}[h]$ with $c.\mathtt{name} \equiv \langle \mathtt{\_\_Host}, \mathtt{sessionId} \rangle$ and $c.\mathtt{content.value} \equiv usid$,
- $S(u).\mathtt{sessions}[usid][\mathtt{request}].\mathtt{parameters}[\mathtt{redirect\_uri}] \equiv redirect\_uri \in \mathsf{URLs}$ with $redirect\_uri.\mathtt{host} \in \mathsf{dom}(r)$ and $redirect\_uri.\mathtt{protocol} \equiv \mathsf{S}$,
- $S(u).\mathtt{sessions}[usid][\mathtt{request}].\mathtt{parameters}[\mathtt{nonce}] \equiv pnonce$ and
- $S(u).\mathtt{sessions}[usid][\mathtt{sync}] \equiv \mathtt{ok}$,

we have that there is a nonce *lsid*, a cookie c' and a domain $h' \in \mathsf{dom}(r)$, such that $c' \in^{\langle\rangle} S(b).\mathtt{cookies}[h']$

- $c' \in^{\langle\rangle} S(b).\mathtt{cookies}[h']$ with $c'.\mathtt{name} \equiv \langle \mathtt{\_\_Host}, \mathtt{sessionId} \rangle$ and $c'.\mathtt{content.value} \equiv lsid$ and
- $S(r).\mathtt{sessions}[lsid][\mathtt{nonce}] \equiv pnonce$.

Proof. If we have $S(u).\mathtt{sessions}[usid][\mathtt{sync}] \equiv \mathtt{ok}$ for a Cross-Device Stub u that is honest in S and a term *usid* then u must have executed set $S(u).\mathtt{sessions}[usid][\mathtt{sync}]$ before. This can only happen when Line 28 of Algorithm C.9 is executed.

From this follows that u must have received an encrypted HTTP request message for the HTTP request m_f and processed the request m_f starting at Line 17 of Algorithm C.9. We get that $m_f.\mathtt{path} \equiv \mathtt{/stub/fin}$ and $m_f.\mathtt{parameters}[\mathtt{xdev\_sync}] \equiv \mathtt{ok}$ (Line 17), $m_f.\mathtt{headers}[\mathtt{Cookie}][\langle \mathtt{\_\_Host}, \mathtt{sessionId} \rangle] \equiv usid$ (Lines 18 and 28),

$$pnonce \equiv m_f.\mathtt{parameters}[\mathtt{cnonce}] \equiv S(u).\mathtt{sessions}[usid][\mathtt{request}].\mathtt{parameters}[\mathtt{nonce}]$$

and $state \equiv m_f.\mathtt{parameters}[\mathtt{cstub}] \equiv S(u).\mathtt{sessions}[usid][\mathtt{state}]$ (Lines 22 to 25). Using the precondition of the Lemma and applying Lemma E.5.3, we get that $state \notin d_\emptyset(S(p))$ for any atomic DY process $p \in \mathcal{W} \setminus \{b, r, u\}$. Thus, only b, r and u could have created the request m_f. Note that r and u never include a cookie header in a message they send, thus b must have created the request m_f.

We have that b might only send such an encrypted HTTP request message for the HTTP request m_f if the algorithm HTTP_SEND of the browser model has been executed before, and this algorithm is only invoked (with suitable parameters) if

- b executed a script and obtained an `HREF`, `IFRAME`, `FORM` or `XMLHTTPREQUEST` command in the output (see Algorithm C.1) that contained the nonce *state* and term *pnonce* in the URL component *url* with the path $url.\texttt{path} \equiv \texttt{/stub/fin}$,
- if b processed an HTTP response in the algorithm PROCESSRESPONSE with a location header that contained a URL *url* with $url.\texttt{parameters} \equiv m_f.\texttt{parameters}$ (in particular $url.\texttt{parameters[cstub]} \equiv state$, $url.\texttt{parameters[cnonce]} \equiv pnonce$ and $url.\texttt{parameters[xdev\_sync]} \equiv ok$), or
- if b received a `TRIGGER` message and chose `urlbar` or `reload` in Line 8 of the main algorithm of the browser model.

If `urlbar` is chosen in Line 8 of the main algorithm of the browser model, however, the browser b chooses the request components from string constants and the request cannot contain *state*. If the browser chooses `reload` when processing a `TRIGGER` message, then b retrieves the location of a document in its state and sends a request for host, path and parameters of this URL. This document however only exists if b received a response for a request for the location URL before, processed it in the algorithm PROCESSRESPONSE of the browser model and stored the document in the state. We get that b must have sent a request like m_f at another point (as well).

We now show that a script cannot result in b sending m_f. Note that b executing the attacker script cannot result in a command that that contains *state*, as the attacker cannot know *state* and thus the attacker script cannot include it in any command it may output. Thus, the only scripts that may result in b sending m_f are *script_rp_index*, *script_stub_token_form* or *script_render_qr*. The script *script_rp_index* (Algorithm B.3) outputs either in a `FORM` command for a URL with path $\texttt{/startLogin} \not\equiv \texttt{/stub/fin}$ or in a `HREF` command where the parameters a picked from $[\mathbb{S} \times \mathbb{S}]$ and thus cannot contain the nonce *state*. We see that the processing of the script *script_rp_index* cannot result in m_f. The script *script_stub_token_form* (Algorithm C.8) may only output a `FORM` command for a URL with path $\texttt{/token} \not\equiv \texttt{/stub/fin}$ and thus cannot result in m_f. If b applies the relation of *script_render_qr*, b might only obtain a `ShowQR` command or an `HREF` command where the parameters a picked from $[\mathbb{S} \times \mathbb{S}]$ and thus cannot contain the nonce *state*. Again, *script_render_qr* cannot result in b sending m_f. Thus, the case where the command in a script caused b to send m_f is impossible.

We get that b must have received an HTTP response message m_r sent by b, r or u (the only processes that may know the nonce *state* that must be contained in m_r) with $m_r.\texttt{headers[Location]} \equiv url$ and $url.\texttt{parameters} \equiv m_f.\texttt{parameters}$. Note that b never sends a response message, and thus r or u must have sent m_r.

The only point where u might send a response message m_r with such a location header is in Line 16 of Algorithm C.9. There, u retrieves a term $url' \equiv req.\texttt{parameters[redirect\_uri]}$ for a term *req* in Line 9 and sets $url'.\texttt{parameters[xdev\_sync]} \equiv \texttt{request}$ (Line 11). This URL url' is then included in the location header of the response. We have that $url'.\texttt{parameters[xdev\_sync]} \not\equiv \texttt{ok}$ and this response message m_r can not result in the redirect.

There are two points where r might send a message m_r with such a location header, Line 16 of Algorithm C.2 and Line 41 of Algorithm C.5. We first show that the latter is impossible. If r executed Line 41 of Algorithm C.5, we have that r sends a URL *stub_ep* in the location header that is equal to $stub\_ep' \equiv S(r).\texttt{oidcConfigCache}[host][\texttt{stub\_start}]$ (Lines 10, 36 and 38) for some term *host* except for the parameter `qr_request` added in Line 37. If the URL *stub_ep*

contains *state* in the parameters, so must have $S(r).\texttt{oidcConfigCache}[\mathit{host}][\texttt{stub\_start}]$ that must have been set before by r. The only point where r might set this component of its state is Line 15 of Algorithm B.6 and we get by Lines 11 and 15, that r must have received an encrypted HTTP response containing an HTTP response m_c with

$$m_c.\texttt{body}[\texttt{stub\_start}] \equiv S(r).\texttt{oidcConfigCache}[\mathit{host}][\texttt{stub\_start}] \equiv \mathit{stub\_ep}.$$

This response must have been created by b, r or u as only these processes can know the nonce *state*. However, b never sends any (encrypted) HTTP response, and the Cross-Device Stub u may only send an empty body or a body of the form $\langle t_1, t_2\rangle$ with $t_1 \in \{\texttt{script\_stub\_token\_form}, \texttt{script\_render\_qr}\}$ and t_2 some term. If r sends an (encrypted) HTTP response we have that the body is empty (Line 16 and Line 41), contains a script string without input (Line 11) or the body is a term $\mathit{reqObj} \equiv S(r).\texttt{sessions}[\mathit{sid}][\texttt{registration}]$ (Lines 5, 7 and 8 of Algorithm B.10) for some term *sid*. This term r however must have stored in its state before, for which r could only do in Line 21 or Lines 23 and 33. In both cases the dictionary *reqObj* does not contain a key `stub_start` (Line 14) and thus does not contain *state* in a URL value at this position. We have that the message from Line 41 of Algorithm C.5 cannot be m_r.

It remains the case where r executed Line 16 of Algorithm C.2. If r executed Line 16 of Algorithm C.2, we get that r must have executed Line 27 of Algorithm C.4 as it is the only point where Algorithm C.2 may be invoked. We see that r processed an HTTP request m_s and get that $m_s.\texttt{headers}[\texttt{Cookie}][\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle] \equiv \mathit{lsid}$, $S(r).\texttt{sessions}[\mathit{lsid}][\texttt{nonce}] \equiv \mathit{cnonce}$ and $\mathit{cnonce} \equiv m_s.\texttt{parameters}[\texttt{cnonce}]$ (Lines 2, 6 and 7) from Algorithm C.2. There, r also sets the URL *url* in the location header of m_r to $m_s.\texttt{parameters}[\texttt{stub\_fin}]$, sets the parameters $\mathit{url}.\texttt{parameters}[\texttt{cstub}] \equiv m_s.\texttt{parameters}[\texttt{cstub}] \equiv \mathit{state}$, $\mathit{url}.\texttt{parameters}[\texttt{cnonce}] \equiv \mathit{cnonce} \equiv \mathit{pnonce}$ and $\mathit{url}.\texttt{parameters}[\texttt{xdev\_sync}] \equiv \texttt{ok}$ (Line 12 and following) and modifies the URL obtained from m_s in no other way. We get that $S(r).\texttt{sessions}[\mathit{lsid}][\texttt{nonce}] \equiv \mathit{pnonce}$ and also that the request message m_s that r received must have contained *state*. As m_s contains a cookie header, only r, u and b may know the nonce *state* and r and u never send a cookie header in a message, only b could have created m_s.

The only point where b might create an HTTP request m_s with a suitable cookie header

$$m_s.\texttt{headers}[\texttt{Cookie}][\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle] \equiv \mathit{lsid}$$

is if there is a cookie $c' \in^{\langle\rangle} S(b).\texttt{cookies}[m_s.\texttt{host}]$ with $c'.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ and $c'.\texttt{value} \equiv \mathit{lsid}$ (see the algorithm HTTP_SEND of the browser model). Note that by the definition of main relation of the generic HTTPS server model, r only processes m_s if $\langle m_s.\texttt{host}, k_r\rangle \in^{\langle\rangle} S(r).\texttt{tlskeys}$ for some term k_r and by the definition of the initial state of r and that honest r never changes this component of its state, we get $m_s.\texttt{host} \in \mathsf{dom}(r)$. We get that Lemma E.5.4 holds. □

Lemma E.5.5 (Cross-Device Redirection Endpoint Integrity)
Let $\chi\mathcal{SIOID}^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\chi\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Self-Issued Identity $id \in \mathsf{SIID}$ where $\mathsf{id\_holder}(id) = i$ is an honest Cross-Device Self-Issued OP in S and every RP $r \in \mathsf{xRP}$ that is honest in S, and every ID Token t with $t \equiv \mathsf{sig}(z, k)$ for a term z and $k \in \mathsf{proof\_key}(id)$ and $\mathit{data} \equiv z.\texttt{token\_data}$ with $\mathit{data}[\texttt{aud}].\texttt{host} \in \mathsf{dom}(r)$, $\mathit{data}[\texttt{aud}].\texttt{protocol} \equiv \texttt{S}$ and $\mathit{data}[\texttt{sub}] \equiv id$, we have that if i sends an encrypted HTTP request req' with $req' \equiv \mathsf{enc_a}(\langle req, key\rangle, reqkey)$ for a nonce *key*, a term *rkey* and an HTTP request *req* with $req.\texttt{body}[\texttt{id\_token}] \equiv t$, then we have $req.\texttt{host} \in \mathsf{dom}(r)$, $req.\texttt{method} \equiv \texttt{POST}$ and $S(i).\texttt{keyMapping}[req.\texttt{host}] \equiv reqkey$.

Proof. Note that when i sends an encrypted HTTP request $req' \equiv \mathsf{enc_a}(\langle req, key_0\rangle, rkey)$ for a nonce key_0, term $rkey$ and an HTTP request req with $req.\mathtt{body} \not\equiv \langle\rangle$, we get that $rkey \equiv S(i).\mathtt{keyMapping}[req.\mathtt{host}]$ from the generic HTTPS server model. We also see that i must have invoked the algorithm HTTP_SIMPLE_SEND of the generic HTTPS server model with req as the message in the input. The only point where i might call HTTP_SIMPLE_SEND with such a message req is in Line 54 of Algorithm C.13.

From Algorithm C.13 we get $req.\mathtt{body}[\mathtt{id\_token}] \equiv t$, $t \equiv \mathsf{sig}(z, k)$ for terms z and k, $data \equiv z.\mathtt{token\_data}$ such that

$$aud \equiv data[\mathtt{aud}] \equiv S(i).\mathtt{sessions}[sid][\mathtt{params}][\mathtt{client\_id}]$$

for $aud.\mathtt{host} \in \mathsf{dom}(r)$ and $aud.\mathtt{protocol} \equiv \mathtt{S}$ (Lines 2, 33 and 50 to 54). We also get from Lines 2, 52 and 53 that

$$req.\mathtt{host} \equiv S(i).\mathtt{sessions}[sid][\mathtt{params}][\mathtt{redirect\_uri}].\mathtt{host}$$

and that $req.\mathtt{method} \equiv \mathtt{POST}$.

We have that i must have set $S(i).\mathtt{sessions}[sid][\mathtt{params}]$ before and the only point where i might have done this is in Line 20 of Algorithm C.12. From this algorithm, Lines 14 and 17, we get

$$\begin{aligned} client\_id &\equiv S(i).\mathtt{sessions}[sid][\mathtt{params}][\mathtt{redirect\_uri}] \\ &\equiv S(i).\mathtt{sessions}[sid][\mathtt{params}][\mathtt{client\_id}] \end{aligned}$$ □

and thus

$$req.\mathtt{host} \equiv client\_id.\mathtt{host} \equiv aud.\mathtt{host} \in \mathsf{dom}(r).$$

Lemma E.5.6 (Cross-Device ID Tokens do not leak)
Let $\chi\mathcal{SIOID}^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\chi\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Self-Issued Identity $id \in \mathsf{SIID}$ where $\mathsf{id\_holder}(id) = i$ is an honest Cross-Device Self-Issued OP in S and every Cross-Device RP r that is honest in S, and every ID Token idt with $idt \equiv \mathsf{sig}(z, k)$ for a dictionary term z and $k \in \mathsf{proof\_key}(id)$ and $data \equiv z.\mathtt{token\_data}$ with $data[\mathtt{aud}].\mathtt{host} \in \mathsf{dom}(r)$, $data[\mathtt{aud}].\mathtt{protocol} \equiv \mathtt{S}$, $data[\mathtt{sub}] \equiv id$ and every atomic DY process $p \in \mathcal{W} \setminus \{i, r\}$, we have that $t \notin d_\emptyset(S(p))$.

Proof. Only $\mathsf{id\_holder}(id) = i$ knows signing keys $k \in \mathsf{proof\_key}(id)$ in the initial state. As i is honest in S, it never emits such a key k to another process. Therefore, if $idt \equiv \mathsf{sig}(z, k) \in d_\emptyset(S(p))$ for a term z and a process $p \neq i$, the term idt must have been created by i.

The only point where i might use a $k \in \mathsf{proof\_key}(id)$ to sign a term z such that $t \equiv \mathsf{sig}(z, k)$ is Line 50 of Algorithm C.13. We see that i there sends an encrypted request containing an HTTP request $req' \equiv \mathsf{enc_a}(\langle req, key\rangle, rkey)$ for a nonce key and an HTTP request req with $req.\mathtt{body}[\mathtt{id\_token}] \equiv t$ and a term $rkey$. By Lemma E.5.5 we have that if $idt \equiv \mathsf{sig}(z, k)$ for a term z and $k \in \mathsf{proof\_key}(id)$ and $data \equiv z.\mathtt{token\_data}$ with $data[\mathtt{aud}].\mathtt{host} \in \mathsf{dom}(r)$, $data[\mathtt{aud}].\mathtt{protocol} \equiv \mathtt{S}$ and $data[\mathtt{sub}] \equiv id$, it holds that $req.\mathtt{host} \in \mathsf{dom}(r)$, $req.\mathtt{method} \equiv \mathtt{POST}$ and $S(i).\mathtt{keyMapping}[req.\mathtt{host}] \equiv rkey$.

As i is honest up to S, we have that $S(i).\texttt{keyMapping} \equiv s_0^i.\texttt{keyMapping} \equiv \langle \mathsf{keyMapping} \rangle$ and thus $k \equiv \mathsf{pub}(\mathsf{tlskey}(h))$ with $h \equiv req.\texttt{host} \in \mathsf{dom}(r)$. As r is honest up to S, only r knows $\mathsf{tlskey}(h)$ in the initial state and r never leaks it, only r might decrypt the encrypted HTTP request req'. The only point where r may decrypt using this key is in the generic HTTPS server model main algorithm that then invokes Algorithm C.4 with req and key.

Then, r processes req in Algorithm C.4, starting at Line 2, 12, 23 or 25. If r starts processing at Line 2, r does not store $req.\texttt{body}$ and does not include it in a message — r only responds with the script *script_rp_index*. When processing a request starting at Line 12, r verifies $req.\texttt{body} \in \mathsf{URLs}$ which is false as $req.\texttt{body}[\texttt{id\_token}] \equiv idt$ and thus r executes stop in this case. If r starts processing at 23, r starts executing Algorithm B.10, but stops as $req.\texttt{method} \equiv \texttt{POST}$.

If r processes req starting from Line 25, r might invoke Algorithm C.2 with req as input (Line 27) or store the ID token idt in $S(r).\texttt{sessions}[sid][\texttt{id\_token}]$ for some term sid (Line 35). In Algorithm C.2 r never accesses $req.\texttt{body}$ or includes it in a message, thus this algorithm cannot result in idt being leaked to another process.

The only point where r may retrieve $idt \equiv S(r).\texttt{sessions}[sid][\texttt{id\_token}]$ is Line 3 of Algorithm B.7. There, r does not store idt again or send a message containing idt.

Thus, only r and $i = \mathsf{id\_holder}(id)$ might know an ID token idt with $t \equiv \mathsf{sig}(z, k)$ for a dictionary term z and $k \in \mathsf{proof\_key}(id)$ and $data \equiv z.\texttt{token\_data}$ with $data[\texttt{aud}].\texttt{host} \in \mathsf{dom}(r)$, $data[\texttt{aud}].\texttt{protocol} \equiv \mathsf{S}$, $data[\texttt{sub}] \equiv id$.

We have that $t \notin d_\emptyset(S(p))$ for all atomic DY processes $p \in \mathcal{W} \setminus \{i, r\}$. □

Lemma E.5.7 (Verification Key Integrity for Cross-Device SIOP)
Let $\chi\mathcal{SIOID}^n$ be a Cross-Device Self-Issued OpenID Provider web system with a network attacker. For any run ρ of $\chi\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every Self-Issued Identity $id \in \mathsf{SIID}$ and if $id \in \mathsf{DID}$ then $\mathsf{govenor}(id)$ is an honest VDR, every RP $r \in \mathsf{xRP}$ that is honest in S, if $S(r).\texttt{sessions}[sid][\texttt{loggedInAs}] \equiv id$ for some term sid, then $S(r).\texttt{sessions}[sid][\texttt{id\_token}] \equiv \mathsf{sig}(t', k)$ for $k \in \mathsf{proof\_key}(id)$ and some term t' with $t'.\texttt{token\_data} \equiv data$ with $data[\texttt{sub}] \equiv id$.

Proof. The proof for this lemma is analogous to Lemma E.1.4 (using a similarly modified Lemma E.1.3). Note that the definition of VDR in $\chi\mathcal{SIOID}^n$ is the same as in $\mathcal{SIOID}^n$, also the relevant algorithms given as Algorithm B.6, Algorithm B.7 and Algorithm B.8 are the same for the Cross-Device RP. The different relevant algorithm is only Algorithm C.3 that replaces Algorithm B.9. In this algorithm however, the relevant lines are the same. □

E.6 Cross-Device Authentication

Theorem E.6.1 (Cross-Device Authentication)
For every Cross-Device Self-Issued OpenID Provider web system with a network attacker $\chi\mathcal{SIOID}^n$, every run ρ of $\chi\mathcal{SIOID}^n$, every configuration (S, E, N) in ρ, every RP $r \in \mathsf{xRP}$ that is honest in S, every Self-Issued Identity $id \in \mathsf{SIID}$ with $\mathsf{id\_holder}(id) = i$ an honest Cross-Device Self-Issued OP

in S where all $b \in \mathsf{userBrowser}(i)$ are honest browsers in S and $\mathsf{trustedStub}(i)$ is an honest Cross-Device Stub in S, every service session identified by some nonce *ssid* for the Self-Issued Identity *id* at r and if $id \in \mathsf{DID}$ then $\mathsf{govenor}(id)$ is an honest VDR in S, it holds that $ssid \notin d_{\emptyset}(S(att))$ where *att* is the network attacker.

Proof. By the definition of a service session identified by some nonce *ssid* for the Self-Issued Identity *id* at r in a configuration (S, E, N), we have that there is a nonce *lsid* with $S(r).\mathtt{sessions}[lsid][\mathtt{serviceSessionId}] \equiv ssid$ and $S(r).\mathtt{sessions}[lsid][\mathtt{loggedInAs}] \equiv id$. We have that r might only set $S(r).\mathtt{sessions}[lsid][\mathtt{serviceSessionId}] \equiv ssid$ in Algorithm C.3, Line 3, where r chooses a fresh nonce *ssid*, and sets $S(r).\mathtt{sessions}[lsid][\mathtt{loggedInAs}]$ to the first parameter passed to the algorithm. It follows that Algorithm B.7 must have been executed, at least up to Line 38 (optionally Algorithm B.7 invokes Algorithm B.8 which then invokes Algorithm C.3).

For r to set $S(r).\mathtt{sessions}[lsid][\mathtt{loggedInAs}]$ to *id*, we get from Algorithm B.7 that it must hold that there is a term *idt* with $idt \equiv S(r).\mathtt{sessions}[lsid][\mathtt{id\_token}]$ (Lines 2 and 3), $data \equiv \mathsf{extractmsg}(idt).\mathtt{token\_data}$ (Line 4) such that $S(r).\mathtt{sessions}[lsid][\mathtt{client\_id}] \in^{\langle\rangle} data[\mathtt{aud}]$ and $data[\mathtt{aud}] \not\equiv \langle\rangle$ (Line 9) as otherwise stop is executed. For this we must have $S(r).\mathtt{sessions}[lsid][\mathtt{client\_id}] \not\equiv \langle\rangle$. This implies Line 33 of Algorithm C.5 was executed before, and we see that $S(r).\mathtt{sessions}[lsid][\mathtt{client\_id}] \in \mathsf{URLs}$ with $S(r).\mathtt{sessions}[lsid][\mathtt{client\_id}].\mathtt{host} \in \mathsf{dom}(r)$. It follows that $data[\mathtt{aud}] \in \mathsf{URLs}$ with $data[\mathtt{aud}].\mathtt{host} \in \mathsf{dom}(r)$. As Algorithm C.3 needs to have *id* in the input, we get from Line 13 of Algorithm B.7 that $data[\mathtt{sub}] \equiv id$. Using Lemma E.5.7, we get that $idt \equiv \mathsf{sig}(t', k)$ for $k \in \mathsf{proof\_key}(id)$ and some term t'.

This term *idt* must have been set in a state before S and the only point where r might set $S(r).\mathtt{sessions}[lsid][\mathtt{id\_token}] \equiv idt$ is Line 35 of Algorithm C.4. We get from Line 33 that r must have processed a request message m with $m.\mathtt{path} \equiv \mathtt{/redirect\_ep}$, $m.\mathtt{methods} \equiv \mathtt{POST}$, $m.\mathtt{body}[\mathtt{id\_token}] \equiv idt$ and

$$pnonce \equiv \mathsf{extractmsg}(idt).\mathtt{token\_data}[\mathtt{nonce}] \equiv S(r).\mathtt{sessions}[lsid][\mathtt{nonce}].$$

Using Lemma E.5.6 we get, that $\mathsf{id\_holder}(id) = i$ must have created this message m as r never sends a request message with path $\mathtt{/redirect\_ep}$ and no other process might know *idt*.

The only point where the Self-Issued OP i might assemble such a request m is in Line 53 of Algorithm C.13, if there exist terms *isess* and *isid* such that $isess \equiv S(i).\mathtt{sessions}[isid]$, $isess[\mathtt{params}][\mathtt{redirect\_uri}].\mathtt{host} \in \mathsf{dom}(r)$ and $isess[\mathtt{params}][\mathtt{nonce}] \equiv pnonce$.

For the Cross-Device Self-Issued OP i to execute Algorithm C.13, we see that i must have executed Algorithm C.12, as only in Lines 13 and 20 of this algorithm the Cross-Device Self-Issued OP i might set $S(i).\mathtt{sessions}[isid]$. We have that i must have processed a term $content \in \mathsf{URLs}$ as input of Algorithm C.12 with $req\_data \equiv content.\mathtt{parameters}$ such that $req\_data[\mathtt{nonce}] \equiv isess[\mathtt{params}][\mathtt{nonce}] \equiv pnonce$, $req\_data[\mathtt{redirect\_uri}] \equiv isess[\mathtt{params}][\mathtt{redirect\_uri}]$ and thus $req\_data[\mathtt{redirect\_uri}].\mathtt{host} \in \mathsf{dom}(r)$ and $req\_data[\mathtt{redirect\_uri}].\mathtt{protocol} \equiv \mathtt{S}$ (Line 14). We also get from Line 9 that

$$stub\_token \equiv req\_data[\mathtt{stub\_token}] \in^{\langle\rangle} S(i).\mathtt{stub\_tokens}.$$

As i is honest in S and thus $S(i).\texttt{stub\_tokens} \equiv s_0^i.\texttt{stub\_tokens}$, we get with the definition of the initial state of i that *stub_token* $\in \mathsf{StubToken}^i$. For i to process *content* in Algorithm C.12 we have that Lines 2 to 4 of Algorithm C.11 must have been executed. Thus, i must have received a QR message $m_{\texttt{qr}} \equiv \mathsf{enc_a}(\langle \texttt{QR}, \mathit{content}\rangle, \mathsf{pub}(S(i).\texttt{qrkey}))$ with $S(i).\texttt{qrkey} \equiv s_0^i.\texttt{qrkey} \equiv \mathsf{qrkey}(i)$ as i is honest is S and never changes this component of its state.

By Lemma E.5.2, we have that only the Cross-Device Stub $u = \mathsf{trustedStub}(i)$ and browsers $b \in \mathsf{userBrowser}(i)$ can know *stub_token* other than i and thus only these processes can create $m_{\texttt{qr}}$. As u is an honest Cross-Device Stub up to S and i is an honest Cross-Device Self-Issued OP up to S, we have that neither may send a QR code message $m_{\texttt{qr}}$. Thus, a browser $b \in \mathsf{userBrowser}(i)$ (that is honest in S) must have sent the QR code message $m_{\texttt{qr}}$. The only point where b might send a QR code in such a message, is when processing the command $\langle \texttt{ShowQR}, \mathit{content}\rangle$ in a script (Lines 95 to 98 of Algorithm C.1). Note that the command $\langle \texttt{ShowQR}, \mathit{content}\rangle$ cannot be in the output from the attacker script, as the attacker cannot know the stub token *stub_token* contained in *content*. We have that b must have obtained the command from the script *script_render_qr* (Algorithm C.7). There, b places sets *content* $\equiv$ *scriptstate*. From Algorithm C.1, we get that *scriptstate* $\equiv S(b).\overline{d}.\texttt{scriptstate}$ for a document d that b must have stored in its state before. This value could only have been set by b in Line 14 (set to the script state after executing the script, there b could not have set content if it had not been set before) or Line 60 (which can not be the case as the attacker script cannot derive *content* and no other script may output the command SETSCRIPTSTATE) in Algorithm C.1, or Line 34 and following of the browser algorithm PROCESSRESPONSE. In latter case b must have received a response containing the script string for *script_render_qr* and *content* as the initial script state. Of $u = \mathsf{trustedStub}(i)$, b and i who may know *stub_token* that is contained in *content*, only u might create such a response message. We have that u must have responded to a request m_t from b with the script *script_render_qr* and *content* as the script state in Algorithm C.9, Line 41.

If u executed Line 41 of Algorithm C.9 and send the script and *content*, then there must be a term *usid* such that $S(u).\texttt{sessions}[\mathit{usid}][\texttt{sync}] \equiv \texttt{ok}$ (Line 34), $\mathit{usid} \equiv m_t.\texttt{headers}[\texttt{Cookie}][\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle]$ (Line 32) and

$$\mathit{req\_data} \equiv \mathit{content}.\texttt{parameters} \equiv \mathit{request}.\texttt{parameters} \cup [\texttt{stub\_token} : \mathit{tok}]$$

for a term $\mathit{tok} \equiv$ *stub_token* (Line 39) and $\mathit{request} \equiv S(u).\texttt{sessions}[\mathit{usid}][\texttt{request}]$ (Lines 33 and 37). Note that this with the above properties of *req_data* gives us $\mathit{request}[\texttt{nonce}] \equiv \mathit{pnonce}$, $\mathit{request}[\texttt{redirect\_uri}].\texttt{host} \in \mathsf{dom}(r)$ and $\mathit{request}[\texttt{redirect\_uri}].\texttt{protocol} \equiv \texttt{S}$.

The browser b only sends such a cookie header with name $\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ and value *usid* in a request m_t if there is a cookie $c \in^{\langle\rangle} S(b).\texttt{cookies}[h]$ for a domain $h \in \mathsf{dom}(u)$ with $c.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ and $c.\texttt{content.value} \equiv \mathit{usid}$ (see algorithm HTTP_SEND of the browser model).

Using Lemma E.5.4 we get that there is a nonce lsid', a cookie c' and a domain $h' \in \mathsf{dom}(r)$ such that

- $c' \in^{\langle\rangle} S(b).\texttt{cookies}[h']$ with $c'.\texttt{name} \equiv \langle \texttt{\_\_Host}, \texttt{sessionId}\rangle$ and $c'.\texttt{content.value} \equiv \mathit{lsid}$ and
- $S(r).\texttt{sessions}[\mathit{lsid}'][\texttt{nonce}] \equiv \mathit{pnonce}$.

Note that since r picks fresh session identifier nonces *lsid* (Line 21 of Algorithm C.4) and a fresh associated request parameter nonce *pnonce* (Lines 12 and 33 of Algorithm C.5) in each execution, we get that from

$$S(r).\texttt{sessions}[lsid'][\texttt{nonce}] \equiv pnonce \equiv S(r).\texttt{sessions}[lsid][\texttt{pnonce}]$$

it follows that $lsid' \equiv lsid$.

The only point where the RP r may send a message containing the nonce *ssid* retrieved from $S(r).\texttt{sessions}[lsid][\texttt{serviceSessionId}]$ (Lines 6 to 11) is in Algorithm C.4 where r includes *ssid* as the cookie

$$cookie \equiv \langle \texttt{\_\_Host}, \texttt{serviceSessionId} \rangle, \langle ssid, \top, \top, \top \rangle$$

in the `Set-Cookie` header of a response to an HTTP request m_i with $m_i.\texttt{host} \in \mathsf{dom}(r)$. For this request it must hold that $m_i.\texttt{headers}[\texttt{Cookie}][\langle \texttt{\_\_Host}, \texttt{sessionId} \rangle] \equiv lsid$ (Line 5). As by Lemma E.5.1 only b and r can know *lsid* and r never sends a cookie header, b must have sent this HTTP request m_i encrypted towards r in an encrypted HTTP message $m_i' \equiv \mathsf{enc_a}(\langle m_i, key_1 \rangle, rkey)$ for a nonce key_1 and a public key term $rkey \equiv S(b).\texttt{keyMapping}[d] \equiv \mathsf{pub}(\mathsf{tlskey}(dom))$ (initial state and b never assigns to the key mapping) for $dom \equiv m_i.\texttt{host} \in \mathsf{dom}(r)$. We see that r encrypts the response with the symmetric key key_1 and r does not leak key_1 to another process. Thus, only b can decrypt the response message and retrieve the value of the `Set-Cookie` header containing the term *cookie*. The only point where b might retrieve *cookie* is in Line 3 of the algorithm PROCESSRESPONSE from the browser model. In the following line b might only store the cookie to $S(b).\texttt{cookies}[dom]$ and may store the `Set-Cookie` header in a document in its state (Line 35 of PROCESSRESPONSE) but in the latter case the `Set-Cookie` header is not accessed again as the document is only passed to scripts after the headers are removed and b never accesses a `Set-Cookie` header of a document otherwise.

The only points where b might retrieve *cookie* from $S(b).\texttt{cookies}[dom]$ is Line 4 of HTTP_SEND from the browser model. In Algorithm C.1 we have that b does not use this cookie, since $cookie.\texttt{httpOnly} \equiv \top$, and Line 52 of the browser main relation can not be the case as b is honest in S. In algorithm HTTP_SEND, Line 4, b only includes *cookie* in an HTTP request *creq* if b sends a message with host *dom* and the HTTPS protocol where $cookie.\texttt{content.value} \equiv ssid$ is the value of cookie header for the name $\langle \texttt{\_\_Host}, \texttt{serviceSessionId} \rangle$. Such messages only r can decrypt as only r knows the nonce $\mathsf{tlskey}(dom)$ in the initial state and never leaks this value, and b again uses the corresponding public key $\mathsf{pub}(\mathsf{tlskey}(dom))$ from its key mapping. As r never retrieves a cookie with name $\langle \texttt{\_\_Host}, \texttt{serviceSessionId} \rangle$ from a message, no process other than b and r might learn *ssid*.

We have that $ssid \notin d_\emptyset(S(att))$ where *att* is the network attacker. □

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature