Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Formal Security Analysis of the Shibboleth Web Single Sign On System Using a Comprehensive Model of the Web

David Sommer

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Dipl.-Inf. Guido Schmitz
Commenced:	November 5, 2019
Completed:	June 23, 2020

Abstract

SAML is an open standard that enables information exchange regarding authorization and authentication. A prominent implementation of SAML is Shibboleth. Typically, Shibboleth is used for Web Single Sign On Systems (Web SSO), which enables users to authenticate at multiple services (Relying Parties) using a central identity service (Identity Provider). In this master thesis, a comprehensive formal security analysis for Shibboleth based on the "Web Infrastructure Model" (WIM) is presented. The Web Infrastructure Model is the most comprehensive formal model of the web up to today. The security of Shibboleth with regard to Authentication and Session Integrity is formally proven by using the WIM. During the analysis, some security flaws of Shibboleth were uncovered, which are described in this thesis. Furthermore, mitigations against these attacks are proposed and used for proving the security of Shibboleth. A novel security property, called Single Logout Consistency (SLO Consistency) is defined, which describes the security of the logout flow. This thesis shows that Shibboleth satisfies SLO Consistency as well.

Kurzfassung

SAML ist ein offener Standard, welcher den Austausch von Informationen über Authorisierung und Authentifizierung ermöglicht. Eine prominente SAML-Implementierung stellt hierbei Shibboleth dar. Typischerweise wird Shibboleth für Web Single Sign On (Web SSO) verwendet. Dabei können Nutzer sich bei verschiedenen Diensten (Relying Parties) authentifizieren, indem sie einen zentralen Dienst (Identity Provider) nutzen. In dieser Masterarbeit wird eine umfassende formale Sicherheitsanalyse für Shibboleth basierend auf dem „Web Infrastructure Model“ (WIM) präsentiert. Das WIM ist derzeit das umfangreichste formale Modell des Webs. Die Sicherheit von Shibboleth bezüglich der Sicherheitseigenschaften Authentication und Session Integrity wird formal bewiesen mithilfe des WIMs. Durch die Analyse wurden einige Sicherheitslücken in Shibboleth entdeckt, welche ebenfalls in dieser Arbeit beschrieben werden. Außerdem wird aufgezeigt wie die beschriebenen Angriffe vermieden werden können und die vorgeschlagenen Korrekturen werden verwendet, um die Sicherheit von Shibboleth zu beweisen. Eine neuartige Sicherheitseigenschaft, genannt Single Logout Consistency (SLO Consistency) wird definiert, welche die Sicherheit eines Logout Flows beschreibt. In dieser Thesis wird bewiesen, dass Shibboleth diese Sicherheitseigenschaft ebenfalls erfüllt.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Authenticating and authorizing users is an important component of almost any modern web application. This requires that the implementers of web application have to provide the means to register, authenticate, change passwords, regain access after losing the password, etc. Great care has to be taken when implementing this functionality, since mistakes might open severe security vulnerabilities within the application. Another problem with this approach is that the user is forced to register at each service separately. This comes with its own share of problem such as password reuse, which enables credential stuffing attacks.

One hope on the way to building secure web applications lies in delegated authentication, where a web app no longer stores user information itself but instead relies on another web app, called Identity Provider (IdP). This way of web authentication is called Web Single Sign On (Web SSO). Implementing this requires a well-defined protocol by which the IdP and web applications wanting to use the IdP (called Service Provider, or SP) can communicate.

The Security Assertion Markup Language (SAML) provides such a protocol. It defines message formats, authentication and authorization flows and how they should translate to HTTP messages. SAML is complemented by software implementing this standard, which enables the SP to implement the standard by simply using an existing library. One prominent implementation is Shibboleth, which provides an implementation of an IdP and a library to use for SPs. Within SAML, an SSO flow works as follows: The user starts at a SP which wants to authenticate the user, for instance because she is trying to access a protected resource. The user then gets redirected to an IdP of his choice where she obtains an Assertion. An Assertion is a special document which can be used by the user to convince the SP that she is indeed authenticated. Presenting the Assertion to the SP therefore completes the SSO flow and the user is able to proceed at the SP as an authenticated user.

In this thesis, Shibboleth is formally analyzed by using a Dolev-Yao model that describes most of the important parts of web infrastructure, namely browsers, DNS servers and web servers. It is called the Web Infrastructure Model (WIM) [8]. Within this model, it is possible to precisely define the desired security properties of Shibboleth and to formally prove them.

Deriving this formal model of Shibboleth and proving the desired security properties for it is one main contribution of this thesis. The second main contribution is deriving a novel security property (called SLO Consistency) and proving that Shibboleth satisfies this property.

The aforementioned security properties include Authentication, Session Integrity and Single Logout Consistency (SLO Consistency). The first two describe how a secure login flow can be achieved, whereas SLO Consistency describes how a secure logout flow is achieved. More specifically, Authentication requires that an attacker is unable to log in under the identity of an honest user. Session Integrity requires that if a user is logged in with an identity from a specific IdP, she has to have previously expressed her intent to login at this specific IdP with this specific identity. A more formal definition of these security properties is given in [8][Section 3.4]. A Shibboleth-specific refinement of these security properties is given in Section B.4.

SLO Consistency is a security property which defines security of a Web SSO system regarding the logout flow. After being authenticated at one or multiple SPs by using one identity at one IdP, the user might want to terminate her sessions at all SPs. Instead of requiring the user to terminate her session at each SP separately, Shibboleth provides a more convenient way of achieving this goal. The user simply visits the IdP and expresses her intent to log out. The IdP then communicates with all SPs to terminate the respective sessions of the user. After doing so, an overview page is presented to the user which lists all the services that the IdP has contacted and whether terminating the session was successful for each. SLO Consistency then ensures that this overview page matches the actual state of the system as closely as possible. This is achieved by requiring that if a user has authenticated with a specific SP, this SP shows up in the overview. Furthermore it is required that if the overview page reports a successful logout for an SP, the user's session at this SP is actually terminated. SLO Consistency is described in more detail in Section 5.1.2 and refined for Shibboleth in Section B.4. This definition of a Security Property is novel, as no previous work has been done in this direction before.

While proving the security properties, some attacks on Shibboleth were discovered. These are described in Section 6.1 of this thesis. Mitigations for these attacks are proposed as well and were used for proving the security properties. A high-level sketch of the proofs can be found in Section 6.2, the full proof is given in the appendix. The resulting proofs show that Shibboleth can be secure within a very detailed model of the Web infrastructure.

In the following, an overview of SAML and Shibboleth will be given. Then, the WIM is presented and the model of Shibboleth within the WIM is outlined, followed by a more thorough description of the SLO Consistency security property. Lastly the results of the security analysis of Shibboleth are presented and a quick outlook will be given.

2 SAML and Shibboleth

The *Security Assertion Markup Language* (SAML) is an OASIS standard for exchanging information about authentication and authorization. Shibboleth is a prominent implementation of the SAML standard and is widely used in practice. At the time of writing, the latest revision of the SAML standard is SAML 2.0 and the latest version of Shibboleth is Shibboleth 4.0 which will be used for the analysis. Most of the analysis relies on the behaviour of applications as defined in the SAML standard. Implementation details that are not specified in the standard are modeled as observed for Shibboleth.
SAML defines message exchange formats, message formats and protocols for several use cases. Among these, Web SSO is the most prominent which is why this thesis focuses on this use case. In Web SSO, the user tries to access a protected resource or triggers any other action that requires the Service Provider to authenticate the user. The Service Provider then redirects the user to the IdP, where she authenticates and returns to the Service Provider with an Assertion from the IdP to prove that she is indeed authenticated. The Service Provider then starts a service session for the user. This service session allows the user to access protected resources without having to authenticate again.
The SAML standard consists of multiple documents. The central document for SAML is SAML core ([2]). It describes the messages exchanged and processing rules for these messages. It is complemented by Bindings and Profiles, which are described in the following. *Bindings* specify how XML messages used by SAML translate into HTTP and SOAP messages. They can be found in [9]. *Profiles* specify message flows for the different protocols defined by SAML. They can be found in [6].

2.1 General

2.1.1 Terms

SAML uses multiple terms in a very specific way, some of which are described in the following.

Service Provider (SP) A Service Provider (also called Relying Party (RP)) is some web service that wants to authenticate users, determine authorization for them or get some other user-related information. In the context of SAML, this party is also called *SAML Requester* as it typically creates SAML Requests that are answered by an IdP.

Identity Provider (IdP) An Identity Provider is a web service that is able to provide Assertions about authentication and authorization or to provide some other user-related information. In the context of SAML, this party is also called *SAML Responder* as it typically creates SAML Responses that are consumed by a SP.

Subject A Subject (also called Principal) is an authenticated user. It contains a *NameID*, which could be a username or just an opaque string used between the SP and IdP to uniquely identify the subject.

EntityID An EntityID is a URI used for identifying SPs and IdPs. This URI usually resolves to a location where one is able to find more information about the service.

2.1.2 Messages

SAML defines multiple messages that are to be used in the various protocols. Since all messages use XML, messages and their contents are XML elements. The SAML core document precisely defines each of the messages using XML Schema Definition ([10]). In the following, they are informally described with less relevant details omitted.

Assertions

Assertions form the core of the SAML protocol for authentication. They contain statements of the form "Subject X is authenticated by A, is authorized to do B on C and has attributes D", called Authentication Statements, Authorization Statements and Attributes respectively. It therefore contains information regarding Authentication, Authorization and some Attributes. Since this thesis investigates Web SSO, authentication statements are of particular interest. Assertions are typically contained in messages sent by the IdP. They are signed by the IdP to prevent other parties from generating them. Usually, an Assertion is the only requirement for logging in at a Service Provider, meaning anyone able to present a valid Assertion gets logged in. Listing 2.1 shows a simplified example of a SAML Assertion. This Assertion was issued by an IdP with the EntityID `http://idp.example.com/metadata.php` as can be seen from the `Issuer` element. It authenticates the Subject with a NameID of `ce3d2948b4cf20146dee0a0b3dd6f69b6cf86f62d7`. It is intended for the SP with the EntityID `http://sp.example.com/demo1/metadata.php`, which can be seen from the `Conditions` element. It contains an Authentication Statement which indicates a Session Index that the Subject obtained during authentication at the IdP. This Session Index can later be used to reference the IdP session. Details about this are found in Section 5. This Assertion would typically be signed using XML Signatures (Here, they are omitted for simplicity). XML Signatures are a mechanism to include cryptographic signatures within XML documents. Details are given in [3]. They have their own share of problems (e.g. [7]), but these are not within the scope the thesis.

```
  <Assertion ID="d71a3a8e9fcc45c9e9d248ef7049393fc8f04e5f75" Version="
2.0">
    <Issuer>http://idp.example.com/metadata.php</Issuer>
    <Subject>
      <NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-
format:transient">
        ce3d2948b4cf20146dee0a0b3dd6f69b6cf86f62d7
      </NameID>
    </Subject>
    <Conditions NotBefore="2014-07-17T01:01:18Z">
```

```
      <AudienceRestriction>
      <Audience>http://sp.example.com/demo1/metadata.php</Audience>
      </AudienceRestriction>
    </Conditions>
    <AuthnStatement SessionIndex="
be9967abd904ddcae3c0eb4189adbe3f71e327cf93">
      <AuthnContext>
        <AuthnContextClassRef>
          urn:oasis:names:tc:SAML:2.0:ac:classes:Password
        </AuthnContextClassRef>
      <AuthnContext>
    </AuthnStatement>
  </Assertion>
```

Listing 2.1: A simplified SAML Assertion containing an Authentication Statement

Authentication Request

An Authentication Request is used by a SP to query authentication information from the IdP which is typically the case for SSO. It specifies some constraints for the Assertion that the IdP should return. For the analysis a very simple version of the Authentication Request was used. In fact, an Authentication Request is not even required, as SAML 1.1 did not even have the concept of a SP-initiated SSO flow. Instead it relied on a mechanism that is called *Unsolicited SSO* in SAML 2. In Unsolicited SSO, the user starts an SS0 flow at the IdP by authenticating there. Since the Web Infrastructure Model mandates a very strict definition of Session Integrity, Unsolicited SSO had to be forbidden to prove Session Integrity. Unfortunately, unsolicited SSO is a rather central feature of SAML and Shibboleth, which means that implementing session integrity comes with a non-negligible cost.
Listing 2.2 contains a simplified SAML Authentication Request. It was issued by a Service Provider with the EntityID `http://sp.example.com/demo1/metadata.php`. The IdP is instructed to return the SAML Response to `http://sp.example.com/demo1/index.php?acs` using the HTTP-POST Binding, which is described in Section 2.2.2.

```
  <AuthnRequest ID="ONELOGIN_809707f0030a5d00620c9d9df97f627afe9dcc24"
        Version="2.0"
        Destination="http://idp.example.com/SSOService.php"
        ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST
"
        AssertionConsumerServiceURL="http://sp.example.com/demo1/index.
php?acs">
    <Issuer>
      http://sp.example.com/demo1/metadata.php
    </Issuer>
  </AuthnRequest>
```

Listing 2.2: A simplified SAML Authentication Request

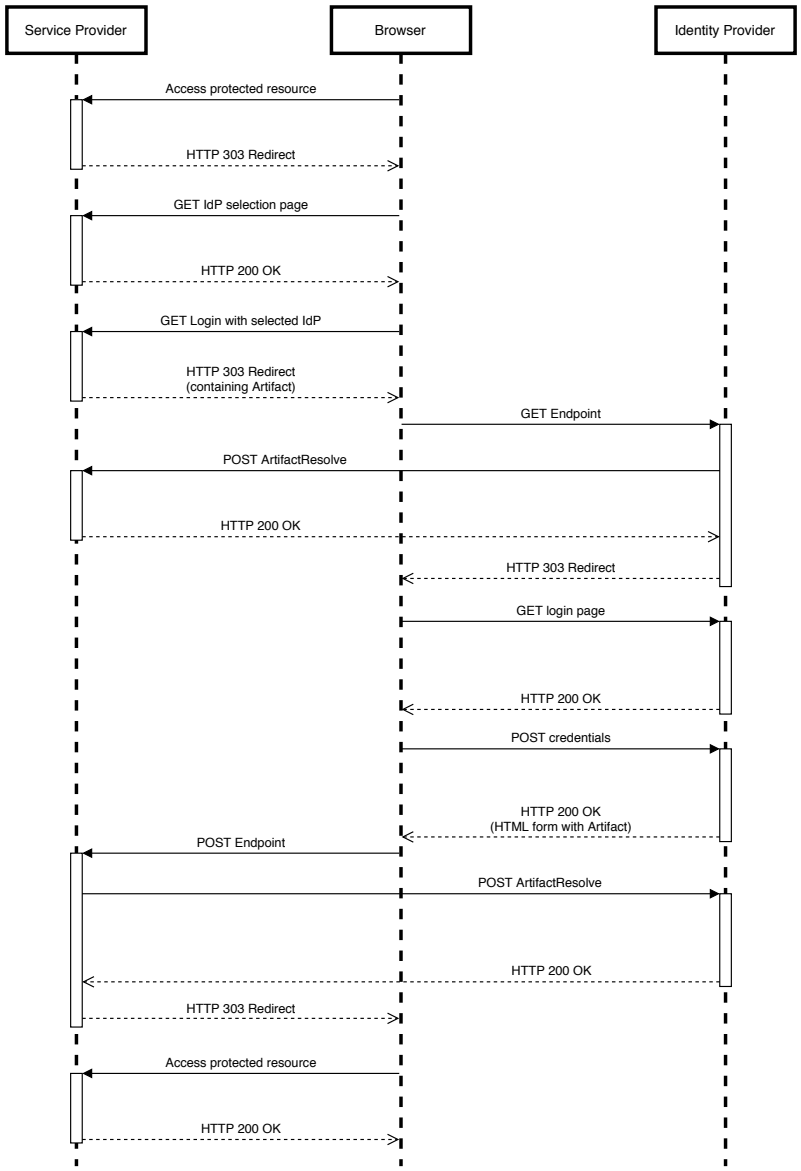

Figure 2.1: SSO Flow with Artifact Binding
An SSO flow which uses the Artifact with HTTP Redirect Binding for transferring the SAML Request and the Artifact with HTTP POST Binding for transferring the SAML Response.

Response

A Response either contains an Assertion or an error, describing why the Assertion could not be returned.

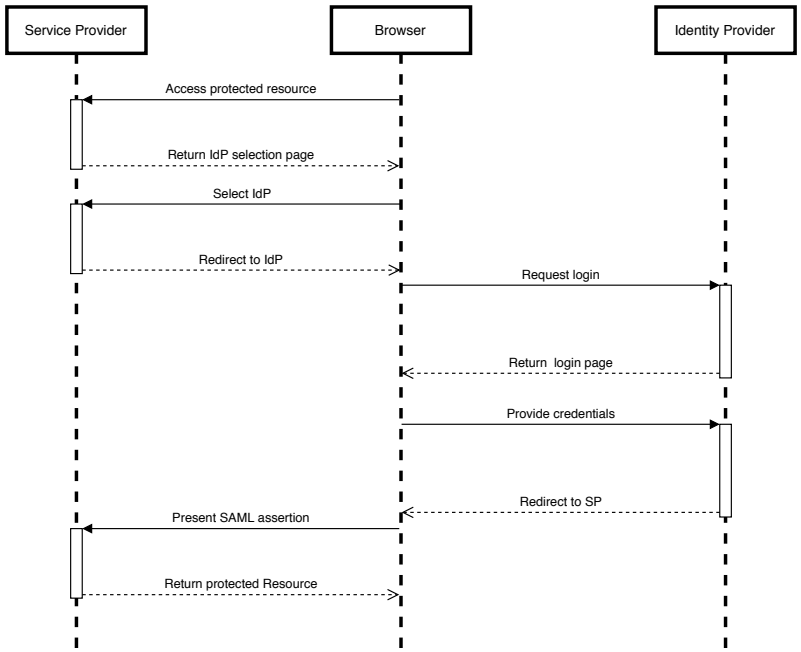

Figure 2.2: Overview of a SAML SSO Flow.

2.2 Web SSO Flow

The high-level overview of a user logging in is illustrated in Figure 2.2. The user first tries to access some protected resource. She is then presented a choice of multiple Identity Providers, of which she selects one. The SP then redirects her to the selected Identity Provider, sending the IdP an Authentication Request. The user is then asked by the IdP to provide valid credentials. After providing these, the user is redirected by the IdP back to the Service Provider with a SAML Response. By presenting the SAML Response to the Service Provider, the user is able to log in.
The way in which this is implemented depends on the *binding* used. Figure 2.3 shows one exemplary flow that uses HTTP redirects to forward the user from the SP to the IdP and uses an HTTP Form to redirect her back. This first way of redirecting is called the *HTTP-Redirect Binding*.

2.2.1 HTTP-Redirect Binding

The HTTP-Redirect Binding works by answering an HTTP(S) Request with an HTTP(S) Response containing a status code of 303 (See Other) or 302 (Found), with the `Location` field set to the target location. The payload (e.g. the Authentication Request or SAML Response) is then encoded into a URL parameter.

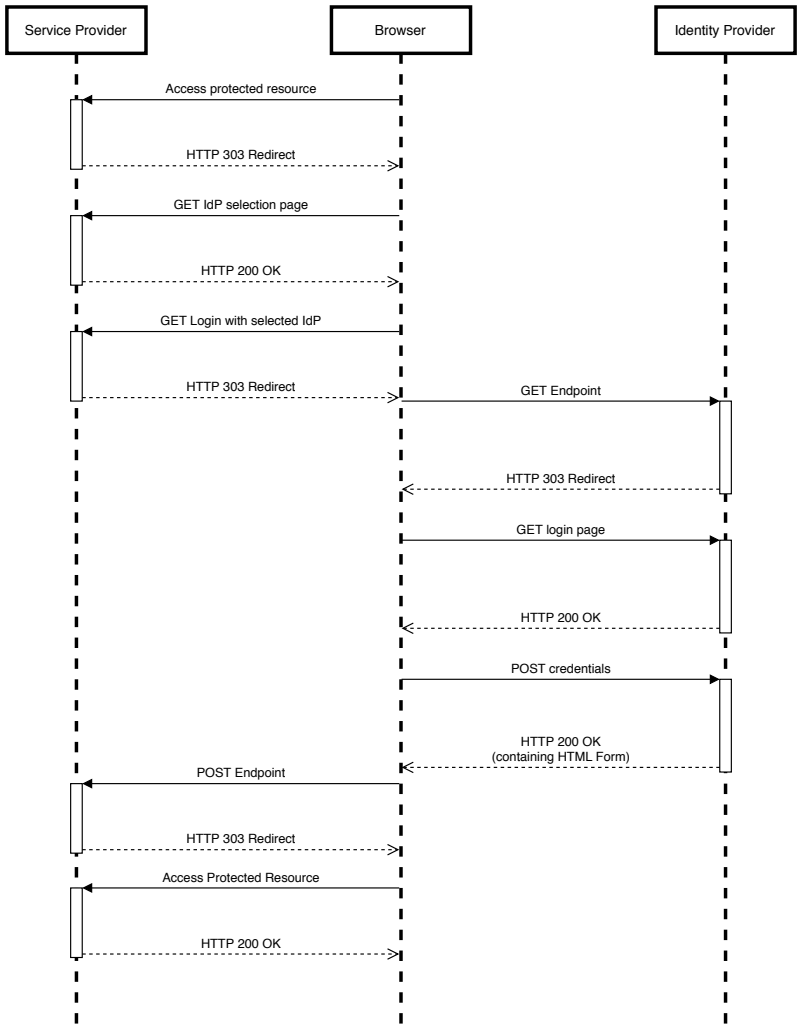

Figure 2.3: Exemplary SSO Flow
An Exemplary SSO Flow. For the SAML Request, the HTTP-Redirect Binding is used, for the SAML Response the HTTP-POST Binding is used.

HTTP Redirects

HTTP Redirects provide a way for a web server to redirect the browser to a different location. This works as follows: The browser tries to access a resource by sending an HTTP Request. The server answers the Request with a HTTP Response containing a special HTTP status code and a URL within the `Location` header. The HTTP status code 302 is defined in RFC 1945 ([1]) and mandated that the resulting Request must use the URL that is provided in the `Location` header. The resulting Request must be otherwise identical to the Request that created the HTTP 302 Response. Traditionally, many browsers violated the standard by issuing a GET Request after receiving a HTTP 302 Response for an HTTP POST Request. Therefore, two new status codes, namely 303

and 307, were defined with 303 mandating a GET Request and 307 mandating a POST Request. However, for backward compatibility reasons, SAML still allows a 302 Redirect for the HTTP-Redirect Binding.

Advantages and Disadvantages

This approach has the advantage that no POST Request is required, which enables the receiving party to have a more strict same-site setting. Furthermore, no Javascript is required to perform this kind of Request. One disadvantage is the limit that some browsers enforce on the length of URLs. This is the reason that the SAML standard forbids sending SAML Responses using this Binding, as they are typically rather big. Another disadvantage is the information leak of possibly sensitive information from the browser history. This problem is investigated by this thesis and some problems are presented in Chapter 6.1.

2.2.2 HTTP-POST Binding

The HTTP-Post Binding works by answering an HTTP(S) Request with an HTTP(S) Response containing an HTML form. This HTML form has its `method` set to `POST`. The payload is stored in hidden fields. An exemplary HTML form for the HTTP-POST Binding can be seen in Listing 2.3.

```
<form action="http://sp.example.org/Shibboleth.sso/SAML2/POST" method="
post">
  <input type="hidden" name="RelayState" value="..."/>
  <input type="hidden" name="SAMLResponse" value="..."/>
  <noscript>
    <input type="submit" value="Continue"/>
  </noscript>
</form>
```

Listing 2.3: HTML form for the HTTP-POST Binding

The form is then sent (automatically using Javascript or manually by using a visible button on the form) to the recipient.

Advantages and Disadvantages

This way of redirecting information creates a POST Request which eliminates both shortcomings of the previous approach. However, some sites might block POST Requests from different origins. For an automatic redirect, the user furthermore has to allow Javascript. The HTTP-POST and HTTP-Redirect Binding are illustrated in Figure 2.3

2.2.3 HTTP-Artifact Binding

The HTTP-Artifact Binding does not transfer the payload directly, but instead transfers an `Artifact`, which the recipient can *resolve* by issuing a direct request to the sender, which then returns the actual payload. The Artifact has a special format that contains an opaque identifier to uniquely identify the payload, along with the EntityID of the sender. The recipient looks up the endpoint on which the sender accepts Artifact Resolution Requests, to resolve the Artifact into the payload. The recipient then sends an `ArtifactResolutionRequest` containing the identifier of the Artifact to the original sender. The sender responds with the actual payload. The communication between recipient and sender for resolving the Artifact is enabled by using SOAP messages. Transferring the Artifact is done by either using the HTTP-Artifact or HTTP-POST Binding. Figure 2.1 illustrates the usage.

Advantages and Disadvantages

The advantage of the HTTP-Artifact Binding is that the payload is hidden from the user, which might be beneficial for some use cases. Obviously, providing a secure back-channel requires more configuration effort, as one has to either provide a secure network or exchange asymmetric key pairs between SP and IdP.

2.3 Security Measures

To prevent malicious use of the protocol, various security measures are in place. Some of these are opt-in, others are required by the SAML standard for a conforming implementation and are therefore always enabled in Shibboleth. In the following, these security measures are presented.

2.3.1 HTTPS

Although SAML does not strictly require the messages to be transferred via HTTPS, any meaningful Shibboleth installation should have HTTPS enabled. A network attacker would be able to simply record the password of a user in transit, if the IdP did not use HTTPS. If the SP does not use HTTPS, the attacker can record the session cookie to hijack the session of the logged-in user. Using HTTPS for Web SSO with Shibboleth is required to provide essential security to the protocol. The model of Shibboleth therefore assumes that every SP and every IdP only accepts HTTPS connections. Furthermore it is assumed that IdPs and SPs send a `Strict-Transport-Security` header with each HTTPS Response. This is important to avoid TLS stripping attacks and to ensure the integrity of cookies.

2.3.2 Message signing

Message signing is an important part of the SAML protocol. For some messages, signing is required in order to provide unilateral authentication and protect the integrity of the message. One important message that is required to be signed is the Assertion contained in the SAML Response sent by the IdP. The Artifact Resolution message returned by the SP must be signed as well. For the model of Shibboleth it is furthermore assumed that Artifact Resolution Requests to an IdP must be signed as well. This deviates slightly from the actual default behaviour of IdPs which only require messages to be signed if they are sent on port 443. This behaviour is based on the Idea that traffic on ports other than 443 can rely on a secure channel (for instance by being inside a secure network). As the analysis always assumes a network attacker, this does not hold.

2.3.3 Message expiration

SAML messages typically contain fields which limit the timespan during which the message is valid. For instance, SAML Assertions contain a `NotOnOrAfter` field, which is a timestamp for specifying the point in time after which the Assertion may not be used anymore. This is a security best practice but is not considered in the analysis, as it is assumed that the attacker is able to perform the attack within the required time frame.

2.3.4 Message replay

SAML Assertions contain an identifier which uniquely identifies the Assertion. This is used by Shibboleth to prevent an attacker to reuse an Assertion that was already used to create a session. This is again a security best practice. None of the attacks are prevented by this measure and the security proof for Shibboleth does not rely on this behaviour. It is therefore omitted in the model of Shibboleth.

3 The Web Infrastructure Model

This chapter gives a high-level overview over the Web Infrastructure Model as presented in [8]. This model has already been used for analysing other SSO protocols like OAuth 2.0 [4], OpenID Connect [5] or BrowserID [8]. It is the most comprehensive formal model of the web infrastructure to date. It facilitates studying Web protocols and enables proofs which show that a protocol is secure within this detailed model.

3.1 The WIM Dolev-Yao Model

Dolev-Yao models are formal models used to study cryptographic protocols and to prove security properties for them. The Dolev-Yao model for the Web Infrastructure Model consists of processes that listen to and create events. They are connected through a common network that enables the processes to communicate via events. Events consist of a receiver address, a sender address and a message.[1] They are usually written as $\langle a, f, m \rangle$. Processes are able to perform computations on the events by manipulating the terms contained in the events according to well-defined rules. After completing a so-called *processing step*, a process emits a set of new events. These events can then be consumed by the processes that are the recipients of the new events. Formally, events are terms defined over a signature Σ. This signature consists of constants like IP addresses, $\top$, $\bot$, $\Diamond$, ASCII strings, function symbols for public keys, symmetric encryption, asymmetric encryption and signatures, sequences of terms (denoted by e.g. $\langle t_1, t_2, t_3, t_4 \rangle$) and projection symbols $\pi_i(\cdot)$. There is an *equational theory* associated with these terms that defines an equivalence relation on the terms. This equational theory is described in Figure 3.1. Two terms are equivalent if they are equivalent with regard to the equational theory. One writes $x \equiv y$ for two terms that are equivalent. For instance, it holds that $\pi_2(\mathrm{dec_s}(\mathrm{enc_s}(\langle a, b \rangle, k), k)) \equiv b$.

3.2 Browsers, Servers and the Web

The WIM models web browsers, HTTP(S) messages, IP addresses and DNS servers by giving a concrete definition of the processes and message formats. Since the WIM is a framework that is independent of any specific web application, Web application servers are

[1] Unfortunately, the term *event* is also used in another way in the WIM. In the definition of the Session Integrity security property, events are special processing steps. The definition of the SLO Consistency security property also uses events of this second kind. Which definition of events is meant, usually becomes clear from the context.

$$\mathrm{dec_a}(\mathrm{enc_a}(x, \mathsf{pub}(y)), y) = x \tag{3.1}$$
$$\mathrm{dec_s}(\mathrm{enc_s}(x, y), y) = x \tag{3.2}$$
$$\mathrm{checksig}(\mathrm{sig}(x, y), \mathsf{pub}(y)) = \top \tag{3.3}$$
$$\mathrm{extractmsg}(\mathrm{sig}(x, y)) = x \tag{3.4}$$
$$\pi_i(\langle x_1, \ldots, x_n \rangle) = x_i \text{ if } 1 \leq i \leq n \tag{3.5}$$
$$\pi_j(\langle x_1, \ldots, x_n \rangle) = \Diamond \text{ if } j \notin \{1, \ldots, n\} \tag{3.6}$$

Figure 3.1: The equational theory associated with the WIM

not explicitly defined by the WIM, but instead are defined for the specific web application that is to be analyzed. In the following, the most important concepts of the WIM are briefly explained.

3.2.1 HTTP Requests and Responses

An HTTP Request is represented as a term consisting of a nonce, an HTTP method, a host name, a path, URL parameters, HTTP headers and a message body. An HTTP Response is represented as a term consisting of a nonce, an HTTP status code, HTTP headers and a message body. HTTPS messages are modeled by (asymetrically) encrypting the Request with the public key of the receiver and sending a nonce along with the Request. The Response is then (symmetrically) encrypted with the nonce contained in the Request.

3.2.2 Browsers

Browsers are at the core of the Web Infrastructure Model. They receive scripts contained in HTTP(S) messages and execute them. These scripts may represent simple HTML web pages consisting of a few links or complex javascript applications. Users are modeled as a part of the Browser. User actions are modeled by (nondeterministically) choosing an action and executing it.

3.2.3 Attackers

The Web Infrastructure Model defines two types of attackers: Network attackers and web attackers. In the analysis of Shibboleth, a network attacker is assumed (which is more powerful than a web attacker). This attacker is a special process that is able to control the network by re-routing or manipulating messages. It is modeled in the WIM as a process that (nondeterministically) chooses any possible action to execute next.

4 A Formal Model of Shibboleth

Shibboleth is modeled as a Web SSO system as defined in [8, chapter 3]. There, a generic template to formalize SSO protocols in the WIM is outlined. Most importantly, the relevant security properties (Authentication and Session Integrity) are defined in terms of a Web SSO system. Some of the definitions have to be refined for Shibboleth and Session Integrity has to be slightly weakened for Shibboleth, which will be mentioned in the respective sections.
Shibboleth consists of Browsers, Identity Processes, Service Providers and Network attackers. Each of these are modeled as atomic DY processes. Browsers are mostly equivalent to the definition given in the WIM, but have been extended to capture the browser history that may leak sensitive information when the browser is closed. In the appendix A.2, the details are given. Network attackers are also modeled exactly as described in [8]. They simply output everything they can possibly derive and are able to manipulate messages. Modelling Service Providers and Identity Providers is specific to Shibboleth and will be presented in the following two sections.

4.1 Service Providers

A Service Provider is an atomic DY process that wants to authenticate users using SSO. To start an SSO flow, the user visits `/secure`. This is meant to model the user accessing a secure resource. The Service Provider then answers the Request with a static site where the user can select an IdP that she wants to use for SSO. Formally, this is modeled by a script that nondeterministically chooses an IdP from a list of IdPs. This script then sends a Request to `/Shibboleth.sso/Login` with the selected IdP as a URL parameter. The Service Provider then nondeterministically chooses a Binding to use to forward the user with the Authentication Request to the IdP. In Shibboleth, the Binding for the Authentication Request is normally configured statically. However, in the model this configuration option is implemented as this nondeterministic choice. Note that this enables the Service Provider to use different Bindings for different SSO sessions, which a normal Shibboleth Service Provider would not do, therefore this is an over-approximation of the actual Service Provider. However, none of the attacks found rely on this behavior.[1]
Shibboleth supports HTTP-Redirect, HTTP-POST, HTTP-Artifact with Redirect and HTTP-Artifact with POST as Bindings for transferring the Authentication Request. In the model, only the first two Bindings are considered. The reasons for this decision are

[1] Over-approximations of real-world behavior in the model can be found in multiple places. They only make the resulting proof stronger. Since the attacks are tested against the Shibboleth implementation, each over-approximation does not affect the attack in a meaningful way.

given in 4.4.
Before sending the HTTPS Response to forward the user to the IdP, the SP creates a session for the user, which contains the Authentication Request sent and the EntityID of the Identity Provider used. This is not part of Shibboleth, but is required to mitigate the attacks presented in Section 6.1. When receiving a SAML Response, some properties of the Response are checked and, if they are fulfilled, a *service session* is created. An HTTPS Response containing a cookie linked to the service session is returned. A more detailed definition of Service Providers is given in Section B.2.8

4.2 Identity Providers

An Identity Provider is an atomic DY process that enables SPs to perform SSO. Depending on the Binding used, the user issues a GET Request to `/profile/SAML2/Redirect/SSO` or a POST Request to `/profile/SAML2/POST/SSO`. The IdP then starts a session for the user and redirects her to a site with the same URL with a 303 redirect, thus making the user perform a GET Request. In the first HTTPS Response the IdP sends to the user, a cookie `JSID` is set. This cookie ensures the integrity of the login flow at the IdP and enables the IdP to associate the SAML Request to the user session. Note that this closely mimics the actual behavior observed at the Shibboleth IdP. The Identity Provider then provides a login form where the user may log in. This login form must be protected against cross-site Requests. For the model, a simple `Origin` header check was used. After logging in, the Identity Provider redirects the user together with a SAML Response back to the SP. The way in which this is done depends on the Binding specified in the SAML Request. In the HTTPS Response sent out, the `shib_session` cookie is set. This cookie represents the authenticated session of the user at the IdP. It is used for enabling the shortened SSO flow described below.
If the user has been authenticated at the IdP previously, a shortened SSO flow is started. After arriving at the IdP, the user is directly granted a new Assertion and redirected back to the SP (depending on the Binding specified in the SAML Request). A more detailed definition of Identity Providers is given in Section B.2.9

4.3 Single Logout

Single Logout is triggered at the Identity Provider. The user accesses `/profile/Logout` and the Identity Provider, after terminating the IdP session, returns a script called *script_slo*. This script then opens a series of IFrames which are labeled with a Service Provider each. Initially, any of the IFrames points to `/profile/PropagateLogout?key=`j at the domain of the IdP where j is the index of the Service Provider inside the current IdP session. The IdP then forwards a Logout Request to the SP using the HTTP-Redirect Binding. The path of the endpoint at the SP is `/Shibboleth.sso/SLO/Redirect`. After terminating the user session, the SP forwards a Logout Response to `/profile/SAML2/Redirect/SLO` at the IdP. Depending on the content of the Logout Response, the IdP then returns a script, called *script_slo_result* with a status (stored within the scriptstate of the script) of $\top$ or $\bot$. A

logout is considered successful, iff the script *script_slo_result* is loaded with a status of $\top$ for a given Service Provider.
Shibboleth also supports using the HTTP-POST, HTTP-Artifact with Redirect and HTTP-Artifact with HTTP-POST Binding for forwarding both, the Logout Request and the Logout Response. However, in the analysis they were omitted for the reasons given in Section 4.4.

4.4 SAML Bindings

Modelling SAML Bindings is challenging because they are orthogonal to the SSO flow, and any proof has to consider any combination of Bindings used. Bindings are used for transferring the Authentication Request and for transferring the SAML Response. For the Authentication Request, any Binding is allowed. However, only the HTTP-POST and HTTP-Redirect Binding were included in the model. The reason for this is that the HTTP-Artifact Binding provides stronger security than the HTTP-POST or HTTP-Redirect Binding, if correctly configured. Usually, the Binding to be used is statically configured for a SP. This configuration is modelled by letting the SP decide (non-deterministically) which Binding to use for the Authentication Request.
For the SAML Response, only the HTTP-POST and HTTP-Artifact Binding are allowed. Here, only HTTP-POST and HTTP-Artifact with HTTP-Redirect were included in the model (HTTP-Artifact with HTTP-POST was omitted). The reason for omitting HTTP-Artifact with HTTP-POST is the same as for the Authentication Request. The Binding to be used for transferring the SAML Response is determined by the Authentication Request (see Section 2.1.2). In Shibboleth the SP determines the Binding to use for receiving the SAML Response by examining some static configuration. This was again modelled by the SP non-deterministically choosing a Binding to use.
For Single Logout, only the HTTP-Redirect Binding was considered to simplify the proof. The security of Single Logout does not seem to be affected by the choice of the Binding used.

4.5 Security Properties

Three Security Properties were considered for Shibboleth, namely Authentication, Session Integrity and SLO Consistency.

Authentication requires that the attacker must be unable to impersonate an honest user at an SP. This property is formulated in terms of a Service Session with a Service Token where the system has to ensure that the attacker is unable to get hold of a service token of an honest user. Service Sessions and Service Tokens have to be refined for any concrete SSO system that is analysed. For Shibboleth, the service session is stored in the state of the SP process and the service token is handed to the browser of the user by means of a cookie, called `shib_session`.

Session Integrity ensures that the user is not logged in under an identity that she either does not own or for which she did not express her intent to use it. Formally this is done by defining some important events in a Web SSO flow, which are briefly described in the following. A start of an SSO flow has to be defined. This is required for having a notion of a series of processing steps that make up an SSO flow. Next, the selection of an IdP has to be defined. This event captures the point in time when the user clicks on the IdP she wishes to use on a web page which presents her with the choice of multiple IdPs. Next, the selection of an identity is defined. This event is defined as the user submitting her username to the system. Lastly, the successful login at the SP is defined as the point in time when a service session is created for the user and the user just received the service token.

Note that again, these events have to be refined for each Web SSO system that is analysed. The refinements for Shibboleth are found in Section B.4. After defining *connected processing steps* (for which the details are omitted in this thesis, but can be found in [8]), we can now define Session Integrity as follows: If the user (using the browser b) is logged in at a SP p with an identity id from an IdP i, it must hold true that:

1. There has been an SSO flow between b and p
2. There exists a previous processing step in which the user selected the IdP i
3. There exists a previous processing step in which the user selected the identity id
4. The user owns id

Note that this is only a high-level overview of these first two security properties. The details are given in [8]. In the following, the third security property, called SLO Consistency is defined.

5 Single Logout

Terminating sessions is an important part of ensuring the overall integrity of a secure web application. Nevertheless, the security of Single Logout protocols has not been formally analysed previously. Therefore, this thesis presents a generic template to formalize Single Logout protocols, defines some security properties which must hold for the protocol and shows that Shibboleth actually fulfils these properties. Shibboleth 3.0 therefore introduced a proprietary extension to SAML, called *Simple Logout*, which is a mechanism to terminate all sessions of a user near-simultaneously. This mechanism is described in the following.

When a user logs in at a SP using an IdP, she creates a session at the IdP. If she then decides to log in at another SP with the same IdP, this IdP will recognize the session and directly authenticate the user. This (shortened) SSO flow is illustrated by Figure 5.1. The IdP tracks all SP sessions that are created from a IdP session. Furthermore, it sends a reference to the IdP session inside the `Assertion` to the SPs (by using the `SessionIndex` attribute inside the `<AuthnStatement>` element), called *IdP session alias*.
If the user decides to terminate her session at the IdP, the IdP presents the option to terminate all SP sessions that were generated during the IdP session. If the user opts in, the IdP attempts to terminate all sessions and presents an overview to the user that informs her which sessions were successfully terminated. An example of such an overview is found in Figure 5.2.

SAML 2.0 defines a mechanism called *Single Logout* (SLO). Single Logout is different from Simple Logout in that the logout flow is initiated at the SP instead of being initiated at the IdP. Furthermore, after completing the logout flow, the user is redirected to the SP. Since the security property which will be formalized in the following is not affected by the difference between Single Logout and Simple Logout in any way, Simple Logout is used for the analysis and in the following, the two terms are used interchangeably.
Care has to be taken when implementing SLO, as the information presented in the overview must match the actual state of the SSO system. For instance, one might imagine a scenario in which the IdP reports a successful logout at a SP to the user but was unable to terminate the session at the SP. In another scenario the IdP assumes that a session with a SP was already terminated before the user initiated the logout but the session still exists. It would then not even list the SP in the overview of terminated sessions, leaving the user assuming that the session was terminated already.

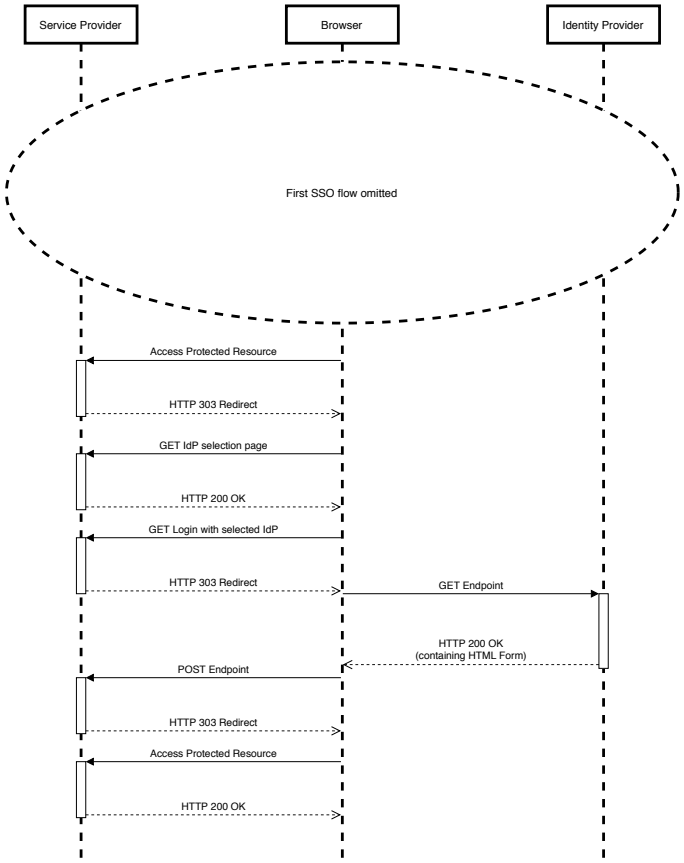

Figure 5.1: A Shortened SSO flow
For the SAML Request, the HTTP-Redirect Binding is used, for the SAML Response, the HTTP-POST Binding is used.

5.1 Formalizing Single Logout

A general template for formalizing Single Logout is given in the following. For defining the security properties of Single Logout, some important events and predicates are introduced.

5.1.1 Important Events and Predicates

Firstly, the IdP session and its reference has to be defined.

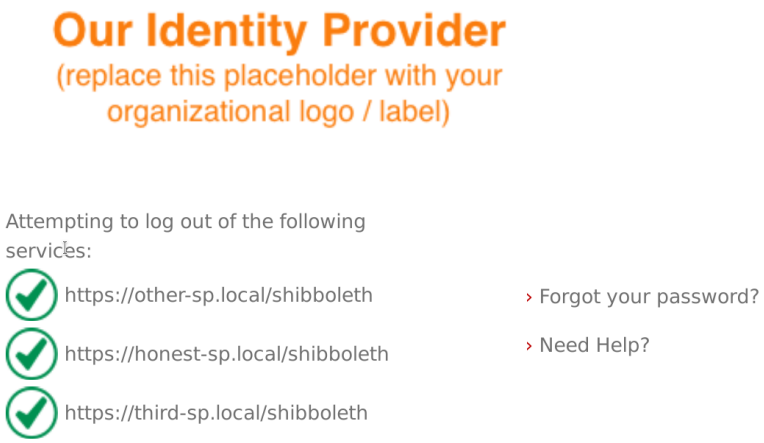

Figure 5.2: Single Logout overview page
This Overview is shown to the user after attempting to log her out of all service that were accessed during the session.

Definition 1 (Session Reference)**.** Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be processing step, i an IdP, b a browser and x an arbitrary term.
Write $\mathsf{secondaryLoggedIn}(Q, b, x, i)$ iff in Q, the browser b has a session with i that is referenced by x. ◇

Next, the predicates related to the overview page are defined.

Definition 2 (Logout Summary)**.** Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be processing step, i an IdP and b a browser. Write $\mathsf{logoutSummary}(Q, b, i)$ iff in Q, a logout summary page for i is presented to b. ◇

Definition 3 (Positive Result)**.** Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be processing step, i an IdP, b a browser and p a Service Provider. Write $\mathsf{positiveResult}(Q, b, i, p)$ iff in Q, a successful logout result is presented to b by i for the Service Provider p. ◇

Definition 4 (Negative Result)**.** Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be processing step, i an IdP, b a browser and p a Service Provider. Write $\mathsf{negativeResult}(Q, b, i, p)$ iff in Q, an unsuccessful logout result is presented to b by i for the Service Provider p. ◇

Furthermore, the termination of sessions is defined in the following.

Definition 5 (Session Tracking at SP)**.** Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be processing step, p a SP, i an IdP and x an arbitrary term. Write $\mathsf{authenticatedWith}(Q, p, i, x)$ iff in Q the SP p has an active session which is authenticated by i with the IdP session alias x. ◇

Definition 6 (Session Termination at IdP)**.** Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be processing step, i an IdP and x an arbitrary term. Write $\mathsf{sessionTerminatedIdP}(Q, x, i)$ iff in Q the IdP i does not have any active session with the IdP session alias x. ◇

5.1.2 SLO Consistency

SLO Consistency is defined in the following.

Definition 7 (SLO Consistency)**.** Let $\mathcal{WS}$ be an SSO Web system. $\mathcal{WS}$ is called *secure w.r.t. SLO Consistency* iff for every run ρ of $\mathcal{WS}$, every processing step Q_β, every IdP i that is honest in Q_β and every browser b that is honest in Q_β with $\mathsf{logoutSummary}(Q_\beta, b, i)$, the following holds true:
If there exists a processing step $Q'_\alpha \in \rho$ and a term x such that $\mathsf{secondaryLoggedIn}(Q_\alpha, b, x, i)$, for all Service Providers p which are honest in Q_β, exactly one of the following statements is true

1. There exists no processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$.
2. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{negativeResult}(Q_\beta, b, i, p)$.
3. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{positiveResult}(Q_\beta, b, i, p)$ and not $\mathsf{authenticatedWith}(Q_\beta, p, i, x)$.

Furthermore, it holds that $\mathsf{sessionTerminatedIdP}(Q_\beta, x, i)$. ◇

6 Analysis of Shibboleth

6.1 Attacks on Shibboleth

During the analysis some attacks on Shibboleth were uncovered. They were disclosed to the Shibboleth developers via the `security@shibboleth.net` contact address. They confirmed the attacks, however did decide not to act upon them. The reason for not acting upon the Response Injection Attack was that this is an attack on the underlying SAML standard. Fixing this attack would require to violate the standard and therefore be incompatible to most if not all other SAML implementations.
The reason for not acting upon the Leaked Assertion Artifact Attack was that the scenario in which this attack occurs (a shared browser environment) is not a scenario in which Shibboleth tries to be secure.
In the following, the attacks are presented in detail.

6.1.1 Leaked Assertion Artifact

If the Assertion is transferred using the HTTP-Artifact Binding, and the HTTP-Redirect Binding is used to transfer the Artifact, the Artifact may leak to the attacker if she gains access to the browser after it is closed, since the Artifact is stored in the browser history. The attacker can use this Artifact to log in under the identity of an honest user. Artifacts transferred by HTTP-Redirect are transferred by encoding them inside an URL query string. This query string is stored inside the browser history. If the attacker gets access to the browser because the browser is running on a public computer (e.g. in a library), the Artifact leaks. Figure 6.1 illustrates the attack.

Proposed Fix

By maintaining a Session on the SP using a cookie with a lifetime set to `Session`, one can ensure that after the user closes the browser the Artifact is no longer valid.

6.1.2 Response Injection Attack

The attacker can start an SSO flow at an IdP to obtain an Assertion (inside a SAML Response) for an arbitrary SP. She can then use this SAML Response to perform a cross-site Request within the browser of the victim to log the victim into her account. This might enable the attacker to track the behaviour of the victim at the SP. Figure 6.2 illustrates the attack.

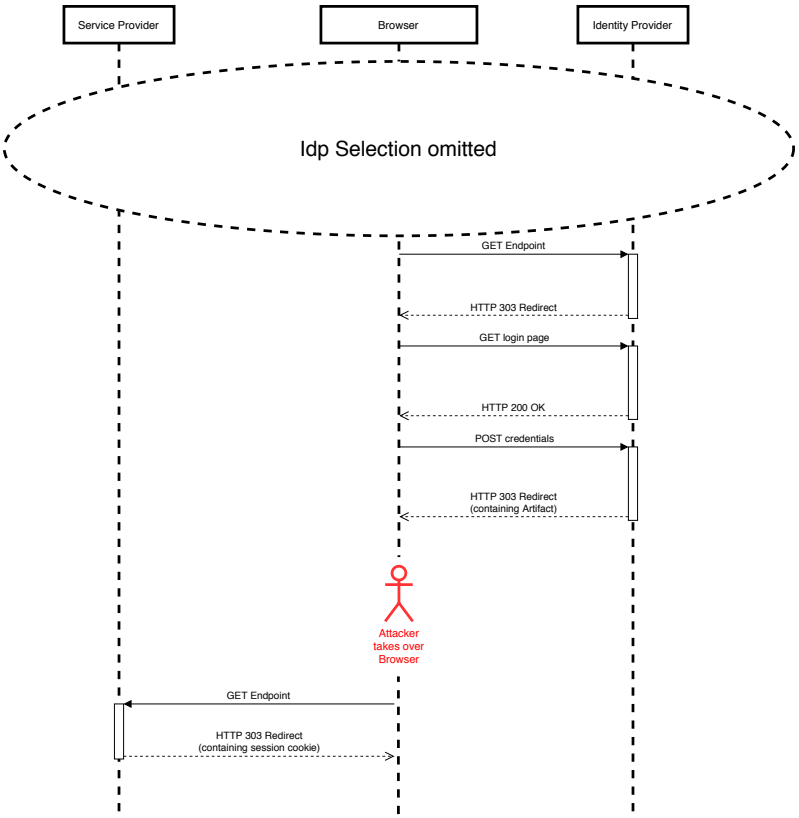

Figure 6.1: Leaked Artifact Attack

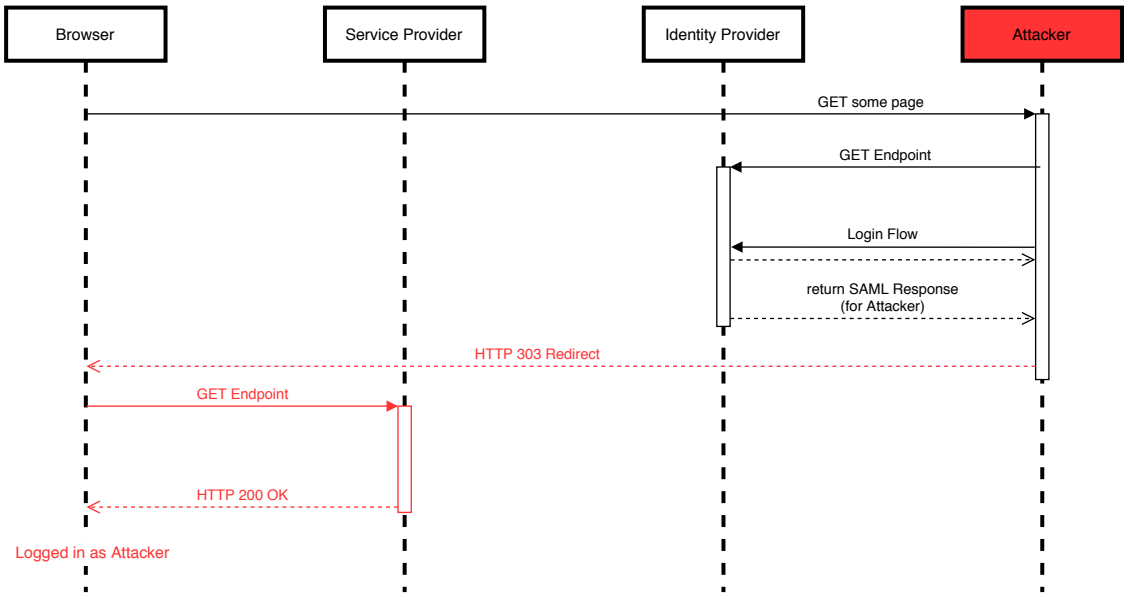

Figure 6.2: Response Injection Attack

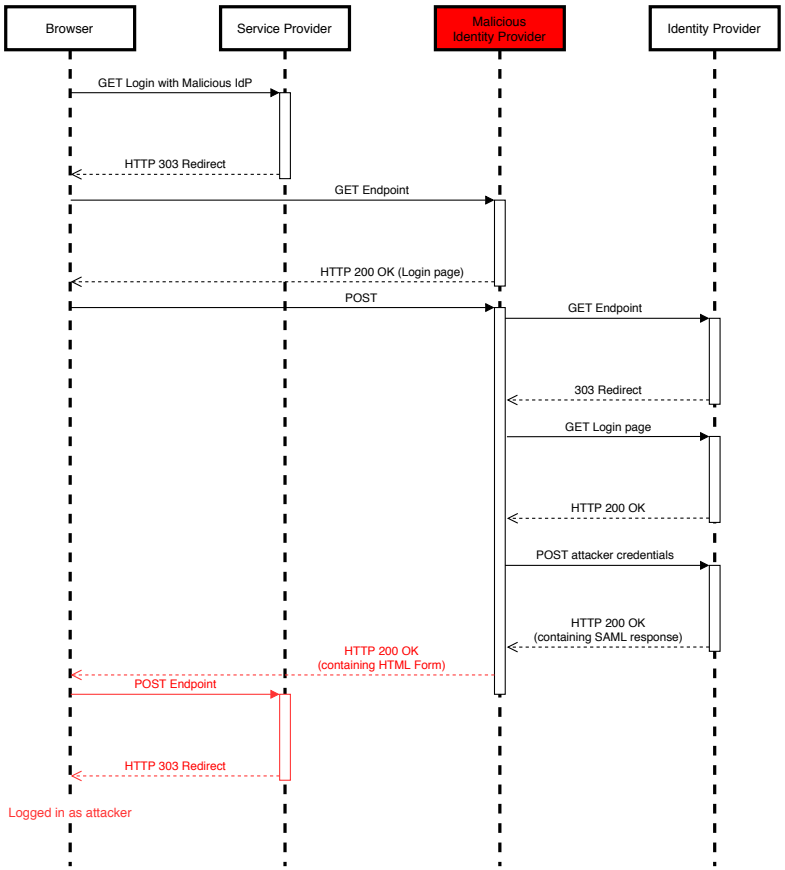

Figure 6.3: Identity Provider Spoofing

Proposed Fix

Again, this issue can be resolved by keeping a session on the SP. In addition, however, Unsolicited SSO (starting an SSO flow at the IdP) has to be forbidden. If a user has not started a session at a SP, this SP will not log the user in even if the user presents a valid SAML Response.

6.1.3 Identity Provider Spoofing

If the user chooses to log in at a malicious IdP, this IdP is able to obtain a valid SAML Response from any other IdP and to return this SAML Response as its own. This enables the malicious IdP to log the user in under an identity that is not governed by the IdP itself and owned by the attacker. This attack enables the attacker to track the behaviour of the victim on the SP. Note that this attack is still possible if the proposed fix for the Response Injection Attack is in place. Figure 6.3 illustrates the attack.

Proposed Fix

In addition to the measures that have to be taken to react to the Response Injection Attack (maintaining a session on the SP), the session at the SP also has to be linked to the EntityID of the IdP that is selected for login.

6.2 Proving Security Properties for Shibboleth

With the proposed fixes in place, Shibboleth is secure w.r.t. Authentication, Session Integrity and SLO Consistency. In the following, sketches of each of the proofs will be given.

6.2.1 Proving Authentication for Shibboleth

The proof for the Authentication property for Shibboleth starts at the point in time where the service session was created and investigates how the service session was created. This can only be done by presenting a valid Assertion to the SP. Next, it is shown that only the IdP which is mentioned in the Assertion is actually able to create it. Since the Assertion can be delivered to the SP by means of different Bindings, certain lemmata are proven which state that only the browser that belong to the honest user is able to make the IdP send out the Assertion. Then it is shown that the Assertion gets in fact delivered to the right SP. Lastly, all the lemmata are combined to show that only the honest browser is able to obtain the Assertion and does not leak it to an attacker, hence the attacker is unable to log in under the identity of the honest browser. The full proof is found in Section B.5.

6.2.2 Proving Session Integrity for Shibboleth

Session Integrity contains multiple statements, which are proven independently. The overall proof starts in the same way as the proof for the Authentication property by starting at the point where the user is logged in. First, it is shown that the user has in fact selected the IdP at which she is now logged in. This is done by simply arguing about the SP which tracks the selected IdP and checks if the issuer of the Assertion matches the IdP that was selected. Next, it is shown that if the user obtains an Assertion from an IdP, she previously selected the identity mentioned in the Assertion and actually owns the identity. Lastly all the previous lemmata are combined to show that in fact all criteria of Session Integrity are fulfilled. The full proof is given in Section B.6.

6.2.3 Proving SLO Consistency For Shibboleth

For proving SLO Consistency, three lemmata are central. The first lemma states that if a positive logout result is presented to the user, the IdP has received an HTTPS Request containing a Logout Response. The second lemma states that if an IdP has received an HTTPS Request containing a Logout Response, it no longer considers the user to be logged

in at the respective Service Provider. The third lemma states that if a Service Provider creates a Logout Response, it no longer considers the user to be logged in. By combining these three lemmata, one is able to prove the SLO Consistency property. The full proof is given in Section B.7.

7 Summary and Outlook

Shibboleth was modeled and analyzed using the WIM model. In the analysis, three attacks on Shibboleth were found which are able to break Authentication and Session Integrity. Fixes are proposed and the security of Shibboleth with the fixes in place regarding Authentication and Session Integrity was formally proven. Furthermore, a third security property for Web SSO systems, namely SLO Consistency, was formalized. This security property makes sure that the result of a logout process in a Web SSO system is presented to the user correctly. Violations of the security property might be as severe as violations of the Authentication property, as the user typically assumes to be logged out of a service, when in fact she is not. It was shown that Shibboleth actually satisfies this property without any changes being required.
Future work might investigate another property related to secure logout, which ensures that the user actually opted into logging out. This property would then also have to consider the different ways in which a logout flow can be initiated, since either the IdP or any SP can start a logout flow. Even though violations of this property are not as severe as violations of the SLO Consistency property, they can still enable an attacker to perform denial of service attacks on the web service. Another topic which might be of interest for future work is to investigate whether all security properties still hold for Shibboleth, if all SAML Bindings are included in the model, even though, as discussed in Section 4.4, they most probably will.

Bibliography

[1] Tim Berners-Lee, Roy Fielding, Henrik Frystyk. *RFC1945: Hypertext Transfer Protocol–HTTP/1.0*. 1996.

[2] Scott Cantor et al. "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2. 0-Errata Composite". In: *Tech. Rep.* (2009).

[3] Donald Eastlake et al. "XML-signature syntax and processing". In: *W3C recommendation* 12 (2002).

[4] Daniel Fett. "An expressive formal model of the web infrastructure". In: (2018).

[5] Seyed Pedram Hosseyni Damabi. "Security analysis of the OpenID financial-grade API". MA thesis. 2018.

[6] John Hughes et al. "Profiles for the oasis security assertion markup language (saml) v2. 0". In: *OASIS standard* (2005).

[7] Michael McIntosh, Paula Austel. "XML signature element wrapping attacks and countermeasures". In: *Proceedings of the 2005 workshop on Secure web services*. 2005, pp. 20–27.

[8] Guido Schmitz. "Privacy-Preserving Web Single Sign-On: Formal Security Analysis and Design". In: (2019).

[9] Oasis Standard et al. *Bindings for the oasis security assertion markup language (SAML) V2. 0*. 2005.

[10] Henry S Thompson et al. "W3C XML schema definition language (XSD) 1.1 part 1: Structures". In: *The World Wide Web Consortium (W3C), W3C Working Draft Dec* 3 (2009).

A The Web Infrastructure Model

A.1 Terms and Messages

The syntax of the WIM is extended for the definition of Shibboleth.
First, a simpler syntax for defining sequences is introduced. Let $X \subseteq \mathcal{T}_{\mathcal{N}}$ be a (possibly infinite) set of terms. Then the set $[X]$ is defined by Equation (A.1):

$$[X] := \{\langle Y \rangle \big| |Y| < \infty \wedge Y \subseteq X\} \tag{A.1}$$

Second, a notation for removing term from sequences is defined. Let $A = \langle a_1, a_2, \ldots, a_m \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ be sequences. Then the sequence $A \setminus^{\langle\rangle} B$ is defined by Equation (A.2)

$$A \setminus^{\langle\rangle} B := \langle \{a_i | a_i \notin^{\langle\rangle} B\} \rangle \tag{A.2}$$

A.2 Web Browsers

The definition of web browsers in [8] is extended to also capture leaks by the browser history. Therefore, the definition of the Web browser state [8, Definition 52] is altered as follows:

Definition 8 (Web browser state extension)**.** The *set of states* $Z_{webbrowser}$ *of a Web browser atomic Dolev-Yao process* consists of terms of the form

$$\begin{aligned}&\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping,\\ &\ sts, DNSaddress, pendingDNS, pendingRequests, isCorrupted, history\rangle\end{aligned} \tag{A.3}$$

with the subterms $windows$, ids, $secrets$, $cookies$, $localStorage$, $sessionStorage$, $keyMapping$, sts, $DNSaddress$, $pendingDNS$, $pendingRequests$, $isCorrupted$ defined as in [8, Definition 52]. The subterm $history \subset^{\langle\rangle}$ URLs contains a list of URLs and reflects the browser history. ◇

Furthermore, the Web browser relation $R_{webbrowser}$ is altered by extending [8, Algorithm A.9] in the following way: The lines from Algorithm A.1 are to be added at line 53 in [8, Algorithm A.9].

Algorithm A.1 Web browser relation: Extension to capture history

1: **let** *docs* $\subset^{\langle\rangle}$ Documents **such that** $\forall i \in \mathbb{N} \forall j \in \mathbb{N} : s.\texttt{windows.i.documents.j} \in$ *docs*
2: **let** $s'.\texttt{history} \subset^{\langle\rangle}$ URLs **such that** $\forall doc \in docs : doc.\texttt{location} \in s'.\texttt{history}$

A.3 Generic HTTPS Server

Daniel Fett proposed a generic template for HTTPS Servers in [4, Appendix A.7]. It will be restated here for completeness

Definition 9 (Base state for an HTTPS server)**.** The states of HTTPS servers that are intstantiations of this generic server must contain at least the following subterms: *pendingDNS* $\in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, *pendingRequests* $\in \mathcal{T}_\mathcal{N}$ (both containing arbitrary terms), *DNSaddress* $\in$ IPs (containing the IP address of a DNS server), *keyMapping* $\in [\text{Doms} \times \mathcal{T}_\mathcal{N}]$ (containing a mapping from domains to public keys), *tlskeys* $\in [\text{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and *corrupt* $\in \mathcal{T}_\mathcal{N}$ (either $\perp$ if the server is not corrupted, or an arbitrary term otherwise). $\diamond$

We note that in concrete instantiations of the generic HTTPS server model, there is no need to extract information from these subterms or to alter these subterms.
Let v_{n0} and v_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. Algorithm A.7 defines the relation of the generic HTTPS server. It works similar to the browser main algorithm and makes use of Algorithms A.3 - A.6 to handle various types of incoming messages. In particular, the function `PROCESS_HTTPS_REQUEST` (Algorithm A.5) is called when an HTTPS Request was received and successfully decrypted. An HTTPS Response is handled by the function `PROCESS_HTTPS_RESPONSE` (Algorithm A.3), trigger messages are handled by `PROCESS_TRIGGER` (Algorithm A.4), and all other messages are handled by `PROCESS_OTHER` (Algorithm A.6). The function `HTTPS_SIMPLE_SEND` (Algorithm A.2) can be used to send HTTPS Requests. In this regard, the generic HTTPS server works just like a (simplified) browser. For example, it first sends a DNS Request to resolve the name into an IP address before sending out an HTTPS Request.

Algorithm A.2 Generic HTTPS Server Model: Sending a DNS message

1: **function** HTTPS_SIMPLE_SEND($reference, message, s', a$)
2: **let** $s'.\texttt{pendingDNS}[v_{\lceil}n0] := \langle reference, message \rangle$
3: **stop** $\langle\langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, message.\texttt{host}, v_{n0}\rangle\rangle\rangle$
4: **end function**

Algorithm A.3 Generic HTTPS Server Model: Default HTTPS Response handler

1: **function** PROCESS_HTTPS_RESPONSE($m, reference, request, key, a, f, s'$)
2: **stop**
3: **end function**

Algorithm A.4 Generic HTTPS Server Model: Default trigger event handler

1: **function** PROCESS_TRIGGER(s')
2: **stop**
3: **end function**

Algorithm A.5 Generic HTTPS Server Model: Default HTTPS Request handler

1: **function** PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2: **stop**
3: **end function**

Algorithm A.6 Generic HTTPS Server Model: Default handler for other messages

1: **function** PROCESS_OTHER(m, a, f, s')
2: **stop**
3: **end function**

Algorithm A.7 Generic HTTPS Server Model: Main relation of a generic HTTPS server

1: **Input:** $\langle a, f, m\rangle, s$
2: **let** $s' := s$
3: **if** $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
4: **let** $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle\rangle$
5: **let** $m' \leftarrow d_V(s')$
6: **let** $a' \leftarrow$ IPs
7: **stop** $\langle\langle a', a, m'\rangle\rangle, s'$
8: **end if**
9: **if** $\exists m_{\text{dec}}, k, k', \mathit{inDomain}$ **such that**
$\langle m_{\text{dec}}, k\rangle \equiv \text{dec}_a(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\texttt{tlskeys}$ **then**
10: **let** $n, \mathit{method}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}$ **such that**
11: $\langle \texttt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
12: **if possible; otherwise stop**
13: **call** PROCESS_HTTPS_REQUEST$(m_{\text{dec}}, k, a, f, s')$
14: **else if** $m \in$ DNSResponses **then**
15: **if** $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin$ IPs
$\vee m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].2.\texttt{domain}$ **then**
16: **stop**
17: **end if**
18: **let** $\langle \mathit{reference}, \mathit{request}\rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
19: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests}$
20: $+^{\langle\rangle}\langle \mathit{reference}, \mathit{request}, v_{n1}, m.\texttt{result}\rangle$
21: **let** $\mathit{message} := \text{enc}_a(\langle \mathit{request}, v_{n1}\rangle, s'.\texttt{keyMapping}[\mathit{request}.\texttt{host}])$
22: **let** $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
23: **stop** $\langle\langle m.\texttt{result}, a, \mathit{message}\rangle\rangle, s'$
24: **else if** $\exists\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
such that $\pi_1(\text{dec}_s(m, \mathit{key})) \equiv \texttt{HTTPResp}$ **then**
25: **let** $m' := \text{dec}_s(m, \mathit{key})$
26: **if** $m'.\texttt{nonce} \not\equiv \mathit{request}.\texttt{nonce}$ **then**
27: **stop**
28: **end if**
29: **remove** $\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle$ **from** $s'.\texttt{pendingRequests}$
30: **call** PROCESS_HTTPS_RESPONSE$(m', \mathit{reference}, \mathit{request}, \mathit{key}, a, f, s')$
31: **stop**
32: **else if** $m \equiv \texttt{TRIGGER}$ **then**
33: **call** PROCESS_TRIGGER(s')
34: **else**
35: **call** PROCESS_OTHER(m, a, f, s')
36: **end if**

B Model of Shibboleth

B.1 Terms and Messages

B.1.1 Message Signing

Some messages in Shibboleth are signed. Let K_{sign} denote the set of all keys used for signing messages.

B.1.2 EntityID

Every IdP and SP is uniquely identified by a special URL called *EntityID*. Let $\mathsf{EntityIDs} \subseteq \mathsf{URLs}$ denote all valid EntityIDs.

B.1.3 Conditions

Conditions under which an Assertion may be used are of the form in (B.1).

$$\langle \texttt{Conditions}, \mathit{audience}, \mathit{one\_time\_use} \rangle \tag{B.1}$$

where

- $\mathit{audience} \in \mathsf{EntityIDs} \cup \{\bot\}$ determines the intended audience.
- $\mathit{one\_time\_use} \in \{\top, \bot\}$ specifies whether the receiving party may retain the Assertion for future use.

Let $\mathsf{Conditions}$ denote the set of all Conditions.

B.1.4 Authentication Statement

An Authentication statement is of the form shown in (B.2).

$$\langle \texttt{AuthnStatement}, \mathit{session\_idx} \rangle \tag{B.2}$$

where $\mathit{session\_idx} \in \mathcal{N} \cup \{\bot\}$ specifies the session id that the subject obtained during authentication at the IdP. Let $\mathsf{AuthnStatements}$ denote the set of all Authentication Statements.

B.1.5 Assertion

An Assertion is a term of the form shown in (B.3).

$$\langle \texttt{Assertion}, id, issuer, subject, conditions, authn\_statement \rangle \tag{B.3}$$

where

- $id \in \mathcal{N}$ uniquely identifies the Assertion.
- $issuer \in \mathsf{EntityIDs}$ describes the issuer of the Assertion.
- $subject \in \mathsf{ID}$ describes the subject of the Assertion.
- $conditions \in \mathsf{Conditions} \cup \{\bot\}$ describes the conditions under which the Assertion may be used.
- $authn\_statement \in \mathsf{AuthnStatements} \cup \{\bot\}$ describes an authentication statement that is made about the subject.

Let $\mathsf{Assertions}$ denote the set of all Assertions. Assertions can be signed. For all Keys $k \in K_{\text{sign}}$ let $\mathsf{SignedAssertions}_k := \{\text{sig}(x, k) | x \in \mathsf{Assertions}\}$ and let $\mathsf{SignedAssertions} := \bigcup_{k \in K_{\text{sign}}} \mathsf{SignedAssertions}_k$.

B.1.6 Authentication Request

General Request

A General Request is a term of the form shown in (B.4):

$$\langle id, destination, issuer \rangle \tag{B.4}$$

where

- $id \in \mathcal{N}$ uniquely identifies the Request.
- $destination \in \mathsf{URLs} \cup \{\bot\}$ denotes the intended destination
- $issuer \in \mathsf{EntityIDs} \cup \{\bot\}$ is the issuer of the Request.

The set of all General Requests is $\mathsf{GeneralRequests}$.

Authentication Request

An Authentication Request is of the form shown in (B.5)

$$\langle \texttt{AuthenticationRequest}, general\_request, acsurl \rangle \tag{B.5}$$

where

- $general\_request \in \mathsf{GeneralRequests}$ is the underlying General Request.
- $acsurl \in \mathsf{URLs} \cup \{\bot\}$ is the URL of the Assertion Consumer Server to which the Response should be delivered.

Let $\mathsf{AuthnRequests}$ denote the set of all Authentication Requests

B.1.7 Response

A SAML Response is a term of the form shown in (B.6)

$$\langle \texttt{Response}, id, inresponseto, destination, issuer, assertion \rangle \tag{B.6}$$

where

- $id \in \mathcal{N}$ uniquely identifies the Response
- $inresponseto \in \mathcal{N} \cup \{\bot\}$ identifies the Request that requested this Response. Furthermore it must hold that $inresponseto \neq \bot$ if and only if this message was generated in response to a Request.
- $destination \in \mathsf{URLs} \cup \{\bot\}$ is the intended destination, called *Assertion Consumer Service*.
- $issuer \in \mathsf{EntityIDs} \cup \{\bot\}$ is the issuer of the Response.
- $assertion \in \mathsf{SignedAssertions}$ is the Assertion that should be delivered.

Let Responses be the set of all Responses.

B.1.8 Artifacts

Artifact

Artifact messages are of the form shown in (B.7):

$$\langle artifact, idp \rangle \tag{B.7}$$

where $artifact \in \mathcal{N}$ and $idp \in \mathsf{IDP}$. Let Artifact be the set of all Artifact messages.

ArtifactResolution

Artifact Resolution messages are of the form shown in (B.8):

$$\langle \texttt{ArtifactResolve}, artifact, issuer \rangle \tag{B.8}$$

where $artifact \in \mathcal{N}$ and $issuer \in \mathsf{SP}$. Let ArtifactResolution be the set of all Artifact Resolution messages.

B.1.9 Logout Request

Logout Requests are of the form denoted by (B.9):

$$\langle \texttt{LogoutRequest}, id, issuer, session\_idx, subject \rangle \tag{B.9}$$

where

- $id \in \mathcal{N}$ uniquely identifies the Response.
- $issuer \in \mathsf{IDP}$ denotes the issuer of the Request.
- $session\_idx$ identifies a session to be terminated.
- $subject \in \mathsf{ID}$ identifies a Subject for which the session should be terminated.

Let $\mathsf{LogoutRequest}$ be the set of all Logout Request messages. Logout Requests can be signed. For all keys $k \in K_{\text{sign}}$ let $\mathsf{SignedLogoutRequest}_k := \{\text{sig}(x,k) | x \in \mathsf{LogoutRequest}\}$ and let $\mathsf{SignedLogoutRequest} := \bigcup_{k \in K_{\text{sign}}} \mathsf{SignedLogoutRequest}_k$

B.1.10 Logout Response

Logout Responses are of the form denoted by (B.10):

$$\langle \texttt{LogoutResponse}, inresponseto, issuer, status \rangle \tag{B.10}$$

where

- $inresponseto \in \mathcal{N}$ identifies the Request that requested this Response.
- $issuer \in \mathsf{SP}$ denotes the issuer of the Response.
- $status \in \{\bot, \top\}$ indicates whether the logout was successful.

Let $\mathsf{LogoutResponse}$ be the set of all Logout Response messages. Logout Responses can be signed. For all keys $k \in K_{\text{sign}}$ let $\mathsf{SignedLogoutResponse}_k := \{\text{sig}(x,k) | x \in \mathsf{LogoutResponse}\}$ and let $\mathsf{SignedLogoutResponse} := \bigcup_{k \in K_{\text{sign}}} \mathsf{SignedLogoutResponse}_k$

B.2 Shibboleth SSO System

Shibboleth is modeled as an SSO Web System as defined in [8, Section 3.1]
Let $\mathcal{SHIB}^p = (\mathcal{W}^p, \mathcal{S}, \text{script}, E_0)$ denote a *Shibboleth SSO Web System*. $\mathcal{W}^p$ is defined as $\mathcal{W}^p = \mathsf{Hon} \dot\cup \mathsf{Web} \dot\cup \mathsf{Net}$ where Hon is here defined by $\mathsf{Hon} := \mathsf{B} \dot\cup \mathsf{SP} \dot\cup \mathsf{IDP} \dot\cup \mathsf{DNS}$ (so $\mathsf{Other} = \emptyset$). Note that this definition deviates slightly from [8, Section 3.1], which defines Relying Parties instead of Service Providers. However, in the following, the two terms can be used interchangeably. For the analysis, one network attacker is considered ($\mathsf{Web} := \emptyset$, $\mathsf{Net} := \{\mathsf{attacker}\}$), DNS servers are considered dishonest (honest DNS servers are irrelevant in the presence of a network attacker).

B.2.1 Entity Identifiers

Entity Identifiers, or simply EntityIDs are special URLs that identify IdPs and SPs uniquely. Let $\mathsf{entityID} : \mathsf{IDP} \cup \mathsf{SP} \to \mathsf{EntityIDs}$ be a bijective mapping that maps IdPs and SPs to their respective EntityIDs.

B.2.2 User Identities

User identities are defined similarly to [8, Definition 1]. The definition is refined here. Users can own multiple identities. They are modeled similarly to an email address. More formally:

Definition 10 (User Identities)**.** A *user identity* (or identity or id for short) *id* is a term of the form $\langle name, eID \rangle$ with $name \in \mathbb{S}$ and $eID \in \mathsf{EntityIDs}$. Let ID be the finite set of identities and y a Dolev-Yao process. Then ID^y is defined to be the set $\{\langle name, \mathsf{entityID}(y) | name \in \mathbb{S} \rangle\}$. ◇

An identity is called *governed* by some DY process i (taking the role of an IdP). Formally this is done by defining a Function $\mathsf{governor} : \mathsf{ID} \to \mathcal{W}$. In the case of Shibboleth, this function is defined as $\mathsf{governor}(\langle name, eID \rangle) := \mathsf{entityID}^{-1}(eID)$. In order to authenticate to an IdP, the user must provide some secret, which represents a password. Formally this is modeled by defining a mapping $\mathsf{secretOfId} : \mathsf{ID} \to \mathsf{Secrets}$ that maps identities to passwords (The set $\mathsf{Secrets}$ is defined later). In the WIM, users are modeled as part of browsers, therefore the assignment of identities to users is modeled by a mapping $\mathsf{ownerOfSecret} : \mathsf{Secrets} \to \mathsf{B}$ which maps secrets to the browser that *owns* it. The mapping $\mathsf{ownerOfId} : \mathsf{ID} \to \mathsf{B}$ is a shortcut, and defined as $\mathsf{ownerOfId}(i) := \mathsf{ownerOfSecret}(\mathsf{secretOfId}(i))$, mapping identities to the browser that (rightfully) owns them.

B.2.3 Addresses and Domain Names

The set IPs contains for $\mathsf{attacker}$, every Service Provider in SP, every Identity Provider in IDP, and every browser in B one address each. Let addr be the assignment of a process to its set of addresses. The set Doms contains one domain for every Service Provider in SP, one for every Identity Provider in IDP and a finite set of domains for $\mathsf{attacker}$. Let dom be the assignment from processes to sets of Doms. If dom or addr returns a set with only one element, this element will be referred to by $\mathsf{dom}(x)$ or addr respectively.
All processes in $\mathcal{W}$ contain all public keys and all private keys of their own domains in their initial state. Let $I^p := \{\mathsf{addr}(p)\}$ for all $p \in \mathsf{Hon}$

B.2.4 Keys and Secrets

The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey} : \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. Atomic processes are given the private keys for their domain in the following form: For an atomic DY process p let

$$tlskeys^p := \langle\{\langle d, \mathsf{tlskey}(d)\rangle | d \in \mathsf{dom}(p)\}\rangle$$

The set $K^{\text{idp}}_{\text{sign}}$ contains the keys that IdPs use for signing messages. Let $\mathsf{signkey}^{\text{idp}} : \mathsf{IDP} \to K^{\text{idp}}_{\text{sign}}$ be an injective mapping that assigns a (different) private key to every Identity Provider.
The set $K^{\text{sp}}_{\text{sign}}$ contains the keys that SPs use for signing messages. Let $\mathsf{signkey}^{\text{sp}} : \mathsf{SP} \to K^{\text{sp}}_{\text{sign}}$ be an injective mapping that assigns a (different) private key to every Service Provider.
The set $\mathsf{Secrets} \subseteq \mathcal{N}$ is the set of passwords (secrets) the browsers share with the Identity Providers.
During a run the attacker can corrupt any number of browsers and can therefore own secrets and identities as well.

B.2.5 Corruption

SPs and IdPs can become corrupted. This works as defined in [8, Section 3.1].

B.2.6 Attackers

Attackers are defined as in [8, Section 3.1]. Its initial state is $s_0^{\text{attacker}} = \langle attdoms, tlskeys, signkeys\rangle$ where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker, *tlskeys* is a sequence of all domains and the corresponding public keys and *signkeys* is a sequence containing all verification keys for all IdPs. All parties use the attacker as a DNS server.

B.2.7 Browsers

Each $b \in \mathsf{B}$ is a web browser as defined in [8, Appendix A.6] with $I^b := \{\mathsf{addr}(b)\}$ being its address.
To define the initial state, first let $ID^b := \mathsf{ownerOfId}^{-1}(b)$ be the set of all identities of b, $ID^{b,d} := \{i | \exists x : i = \langle x, d\rangle \in ID^b\}$ be the set of identities of b for a domain d and $SecretDomains^b := \{d | ID^{b,d} \neq \emptyset\}$ be the set of all domains that b owns identities for.
Then the initial state s_0^b is defined as follows: The key mapping maps every domain to its public (TLS) key, according to the mapping tlskey; The DNS address is $\mathsf{addr}(\mathsf{attacker})$; The list of secrets contains an entry $\langle\langle d, \mathtt{S}\rangle, s\rangle$ for each $d \in SecretDomains^b$ and $s = \mathsf{secretOfId}(i)$ for some $i \in ID^{b,d}$ (s is the same for all i); *ids* is $\langle ID^b\rangle$; *sts* is empty. For scripts, the mapping script maps scripts to their string representation.

B.2.8 Service Providers

A Service Provider $p \in \mathsf{SP}$ is a Web server modeled as an atomic DY process (I^p, Z^p, R^p, s_0^p) with the address $I^p := \{\mathsf{addr}(p)\}$. Its initial state contains its domain, the private key associated with its domain, the DNS server address, and the verification keys for all IdPs. In the following, a formal definition of the DY process will be provided. First, let *idpLookup* denote a dicitionary that maps IdPs to domains:

$$idpLookup := \langle \{\langle i, \mathsf{dom}(i)\rangle | i \in \mathsf{IDP}\} \tag{B.11}$$

Definition 11. A *state* $s \in Z^p$ *of a SP* r is a term of the form defined in (B.12)

$$\begin{aligned}&\langle id, pendingDNS, pendingRequests, DNSaddress, keyMapping,\\ &usednonces, tlskeys, domain, signkey, signkeys, sessions, idps, corrupt\\ &logoutServices, logins, assertions, logoutrequests\rangle\end{aligned} \tag{B.12}$$

where

- $id \in \mathsf{SP}$ is the ID of the SP
- $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ describes the pending DNS Requests
- $pendingRequests \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ describes the pending HTTPS Requests
- $DNSaddress \in \mathsf{IPs}$ describes the address of the SP
- $keyMapping \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$ maps Domains to their respective public keys
- $usednonces \in [\mathcal{N}]$ is a sequence of used nonces
- $tlskeys = tlskeys^p$ maps domains of the Service Provider to private TLS keys
- $domain = \mathsf{dom}(p)$ is the (only) domain of the Service Provider
- $signkey \in \mathcal{N}$ is the signing key of the SP.
- $signkeys \in [\mathsf{IDP} \times \mathcal{N}]$ maps IdPs to their respective signing keys
- $sessions \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary for looking up sessions
- $idps \in [\mathsf{IDP} \times \mathsf{Doms}]$ maps known IdPs to their respective domains
- $corrupt \in \mathcal{T}_{\mathcal{N}}$ determines whether the SP is corrupt
- $logoutServices \in [\mathsf{IDP} \times \mathsf{URLs}]$ maps IdPs to their Logout Response Consumer service
- $logins \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ stores SSO flow state.
- $assertions \in [\mathcal{N} \times \mathsf{Assertions}]$ stores all Assertions that were received
- $logoutrequests \in [\mathsf{LogoutRequest}]$ stores all Logout Requests that were received

Placeholder	Usage
ν_1	ID of AuthnRequest
ν_2	Session-ID of logged-in user

Table B.1: Placeholders and their usage for a Service Provider

Let $logoutSet := \langle\{\langle i, \mathtt{URL}^{\mathsf{dom}(i)}_{\mathtt{/profile/SAML2/Redirect/SLO}}\rangle\}\rangle$ The initial state s_0^p is defined in (B.13)

$$\begin{aligned} s_0^p :=& \langle\langle\rangle, \langle\rangle, \mathsf{addr}(p), keyMapping, \langle\rangle, tlskeys^p, \mathsf{dom}(p), \mathsf{signkey}^{\mathrm{sp}}(p) \\ & \langle\{\langle eID, \mathsf{pub}(\mathsf{signkey}(y))\rangle | y \in \mathsf{IDP}, eID \in \mathsf{entityID}(y)\}\rangle, \langle\rangle, \\ & idpLookup\bot, logoutSet, \langle\rangle, \langle\rangle, \langle\rangle\rangle \end{aligned} \tag{B.13}$$

$\diamond$

The relation R^p is defined in Algorithm B.1. Table B.1 shows a list of all placeholders used in Algorithm B.1.

Algorithm B.1 Relation of a Service Provider p

function PROCESS_HTTPS_REQUEST(m, k, a, f, s)
  let *headers* := $\langle\langle$`Strict-Transport-Security`, $\top\rangle\rangle$
  let s' := s
  let *sp_id* := s'`.id`
  let n, *method*, *path*, *params*, *mheaders*, *body* **such that**
    $\langle$`HTTPReq`, n, *method*, s'`.domain`, *path*, *params*, *mheaders*, *body*$\rangle \equiv m$
    if possible; otherwise stop
  if *path* $\equiv$ `/secure` **then**
    let *scriptstate* := $\langle s'$`.idps`$\rangle$
    let *resp* := $\mathrm{enc_s}(\langle$`HTTPResp`, n, 200, *headers*,
      $\langle$`script_idp_select`, *scriptstate*$\rangle\rangle, k)$
  else if *path* $\equiv$ `/Shibboleth.sso/Login` **then**
    if *mheaders*[`Origin`]`.host` $\equiv s'$`.domain` **then**
      let *idp_id* := *params*[`entityID`]
      let *idp* := s'`.idps`[*idp_id*]
      let *endpoint* $\leftarrow$ {`/profile/SAML2/Redirect/SSO`,
        `/profile/SAML2/POST/SSO`}
      let *idp_url* := $\langle$`URL`, `S`, *idp*, *endpoint*, $\langle\rangle, \bot\rangle$
      let *general* := $\langle \nu_1$, *idp_url*, *sp_id*$\rangle$
      let *saml_request* := $\langle$`AuthenticationRequest`, *general*, $\mathrm{URL}^{s'\texttt{.domain}}_{endpoint}\rangle$
      let *headers*$'$:= *headers* $+^{\langle\rangle}$ $\langle$`Set-Cookie`, $\langle\langle$`LOGIN`, $\nu_2, \top, \top, \top\rangle\rangle\rangle$
      let s'`.logins`[ν_2] := $\langle$*saml_request*, *idp_id*$\rangle$
      if *endpoint* $\equiv$ `/profile/SAML2/Redirect/SSO` **then**
        let *redirect* := $\langle$`URL`, `S`, *idp*, *idp_url*`.path`, $\langle\langle$`SAMLRequest`, *saml_request*$\rangle\rangle, \bot\rangle$
        let *resp* := $\mathrm{enc_s}(\langle$`HTTPResp`, n, 303, *headers*
          $+^{\langle\rangle}\langle$`Location`, *redirect*$\rangle, \langle\rangle), k)$
      else
        let *scriptstate* := $\langle\langle$`destUrl`, *idp_url*$\rangle$, `SAMLRequest`, $\langle$`message`, *saml_request*$\rangle\rangle$
        let *resp* := $\mathrm{enc_s}(\langle$`HTTPResp`, n, 200, *headers*, $\langle$`script_forward`, *scriptstate*$\rangle\rangle)$
      end if
    end if
  else if *path* $\equiv$ `/Shibboleth.sso/SAML2/POST` **then**
    call PROCESS_SAML_RESPONSE(*body*[`SAMLResponse`], *mheaders*, *headers*, s')
  else if *path* $\equiv$ `/Shibboleth.sso/SAML2/Artifact` **then**
    let *artifact*, *idp* **such that** $\langle$*artifact*, *idp*$\rangle \equiv$ *params*[`Artifact`]
    let *endpoint* := $\langle$`URL`, `S`, s'`.idps`[*idp*],
      `/Shibboleth.sso/SAML2/ArtifactResolve`, $\langle\rangle, \bot\rangle$
    let s'`.pendingArtifact` := ν_3
    let *resolve* := sig($\langle$`ArtifactResolve`, *artifact*, *sp_id*$\rangle, s'$`.signkey`)
    let *message* := $\langle$`HTTPReq`, ν_4, `POST`, *idp*, *endpoint*`.path`, $\langle\rangle, \langle\rangle, \langle\langle$`Artifact`, *resolve*$\rangle\rangle\rangle$
    call HTTPS_SIMPLE_SEND(ν_3, *message*, s', *endpoint*)
  else if *path* $\equiv$ `/Shibboleth.sso/SLO/Redirect` **then**
    let *logoutrequest* := *params*[`SAMLRequest`]
    let *id*, *issuer*, *session_idx*, *subject*
      such that $\langle$`LogoutRequest`, *id*, *issuer*, *session_idx*, *subject*$\rangle$
        $\equiv$ extractmsg(*logoutrequest*)
    if checksig(*logoutrequest*, s'`.signkeys`[*issuer*]) **then**
      let s'`.logoutrequests` := s'`.logoutrequests`
        $+^{\langle\rangle}$extractmsg(*logoutrequest*)

```
50:             let terminate := ⟨⟩
51:             let terminateAssertions := ⟨⟩
52:             for all ⟨sid, ⟨_subject, _issuer⟩⟩ ∈^⟨⟩ s'.sessions do
53:                 let idp_session := s'.assertions[sid].authn_statement.session_idx
54:                 if _subject ≡ subject ∧ _issuer ≡ issuer ∧ idp_session ≡ session_idx then
55:                     let terminate[sid] := ⟨_subject, _issuer⟩
56:                     let terminateAssertions[sid] := s'.assertions[sid]
57:                 end if
58:             end for
59:             let s'.sessions := s'.sessions \^⟨⟩ terminate
60:             let s'.assertions := s'.assertions \^⟨⟩ terminateAssertions
61:             if terminate ≡ ⟨⟩ then
62:                 let result := ⊥
63:             else
64:                 let result := ⊤
65:             end if
66:             let logoutresponse := ⟨LogoutResponse, id, sp_id, result⟩
67:             let endpoint := s'.logoutServices[issuer]
68:             let endpoint.parameters[SAMLResponse] ≡ logoutresponse
69:             let headers' := headers +^⟨⟩ ⟨Location, endpoint⟩
70:             let resp := enc_s(⟨HTTPResp, n, 303, headers', ⟨⟩⟩, k)
71:         end if
72:     end if
73:     stop ⟨⟨f, a, resp⟩⟩, s'
74: end function
```

Algorithm B.2 Relation of a Service Provider p: Receiving an Artifact Resolution Response

```
1: function PROCESS_HTTPS_RESPONSE(m, reference, request, key, a, f, s')
2:     let n, method, path, params, mheaders, body such that
3:         ⟨HTTPReq, n, method, s'.domain, path, params, mheaders, body⟩ ≡ m
4:         if possible; otherwise stop
5:     if s'.pendingArtifact ≡ reference then
6:         let response := m.body[ArtifactResponse]
7:         call PROCESS_SAML_RESPONSE(response, mheaders, headers, s')
8:     end if
9: end function
```

Algorithm B.3 Relation of a Service Provider p: Checking a SAML Response

function PROCESS_SAML_RESPONSE($samlresp, mheaders, headers, s'$)
  let $resp\_id, inresponseto, dest, issuer, sassertion$
    such that $\langle \texttt{Response}, resp\_id, inresponseto, dest, issuer, sassertion \rangle$
    $\equiv samlresp$ **if possible; otherwise stop**
  let $request, idp\_id$ **such that**
    $\langle request, idp\_id \rangle \equiv s'.\texttt{logins}[mheaders[\texttt{Cookie}][\texttt{LOGIN}]]$
  let $assertion :=$ extractmsg($sassertion$)
  let $aissuer := assertion.\texttt{issuer}$
  if $request \not\equiv \bot \wedge request.\texttt{id} \equiv inresponseto \wedge aissuer \equiv idp\_id$ **then**
    let $skey := s'.\texttt{signkeys}[issuer]$
    if checksig($sassertion, skey$) $\equiv \top$ **then**
      if $assertion.\texttt{conditions.audience} \equiv s'.\texttt{id}$ **then**
        let $i \in \mathbb{N}$ **such that**
          $s'.\texttt{logoutrequests}.i.\texttt{session\_idx}$
            $\equiv assertion.\texttt{authn\_statement.session\_idx}$
          $\wedge s'.\texttt{logoutrequests}.i.\texttt{issuer} \equiv assertion.\texttt{issuer}$
          $\wedge s'.\texttt{logoutrequests}.i.\texttt{subject} \equiv assertion.\texttt{subject}$
          if possible, else $\bot$
        if $i \equiv \bot$ **then**
          let $sessionid := \nu_2$
          let $s'.\texttt{sessions}[sessionid] := \langle assertion.\texttt{subject}, assertion.\texttt{issuer} \rangle$
          let $s'.\texttt{assertions}[sessionid] := assertion$
          let $headers' := headers +^{\langle\rangle} \langle \texttt{Set-Cookie}, \langle \langle \texttt{SID}, sessionid, \top, \top, \top \rangle \rangle \rangle$
          let $resp := \text{enc}_s(\langle \texttt{HTTPResp}, n, 200, headers', \langle\rangle \rangle, k)$
        end if
      end if
    end if
  end if
  stop $\langle \langle f, a, resp \rangle \rangle, s'$
end function

B.2.9 Identity Providers

An Identity Provider $i \in \mathsf{IDP}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with the address $I^i := \{\mathsf{addr}(i)\}$. Its initial state s_0^i contains a list of domains and (private) TLS keys, a list of users and identities, information about active sessions, a flag that stores whether the IdP is corrupt, a list of used nonces and a private key for signing Assertions. Let

$$userset^i = \langle \{ \langle u, \langle \mathsf{secretOfId}(u) \rangle \rangle | u \in \mathsf{ID}^i \} \rangle \tag{B.14}$$

Furthermore, let $tlskeys^i$ be a dictionary that maps domains to private TLS keys of an IdP i:

$$tlskeys^i := \langle \{ \langle d, \mathsf{tlskey}(d) \rangle | s \in \mathsf{Secrets}^i \} \rangle$$

Definition 12. A *state* $s \in Z^i$ *of an IdP* i is a term of the form described in (B.15)

$$\begin{aligned}&\langle \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{DNSaddress}, \mathit{keyMapping}, \mathit{usednonces},\\ &\mathit{tlskeys}, \mathit{users}, \mathit{sessions}, \mathit{corrupt}, \mathit{signkey}, \mathit{signkeys}, \mathit{domain}, \mathit{acServices},\\ &\mathit{logoutRequests}, \mathit{logoutServices}, \mathit{shib\_sessions}, \mathit{invalid\_sessions}, \mathit{samlResponses}\rangle\end{aligned} \tag{B.15}$$

where

- $\mathit{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ describes the pending DNS Requests
- $\mathit{pendingRequests} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ describes the pending HTTPS Requests
- $\mathit{DNSaddress} \in \mathsf{IPs}$ describes the address of the IdP
- $\mathit{keyMapping} \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$ maps Domains to their respective public keys
- $\mathit{usednonces} \in [\mathcal{N}]$ is a sequence of used nonces
- $\mathit{tlskeys} = \mathit{tlskeys}^i$ maps domains of the IdP to private TLS keys
- $\mathit{users} = \mathit{userset}^i$ is a dictionary as defined in (B.14)
- $\mathit{sessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary for looking up sessions
- $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$ determines whether the IdP is corrupt
- $\mathit{signkey} \in \mathcal{N}$ is the key used to sign Assertions
- $\mathit{signkeys} \in [\mathsf{SP} \times \mathcal{N}]$ maps SPs to their respective signing keys
- $\mathit{domain} \in \mathsf{Doms}$ is the domain owned by the IdP
- $\mathit{acServices} \in [\mathsf{SP} \times [\mathsf{URLs}]]$ is a mapping from Service Providers to their respective Assertion Consumer Services
- $\mathit{logoutRequests} \in [\mathcal{N} \times \mathsf{LogoutRequest}]$ stores pending Logout Requests
- $\mathit{logoutServices} \in [\mathsf{SP} \times \mathsf{URLs}]$ maps SPs to their respective Logout Service
- $\mathit{shib\_sessions} \in [\mathcal{N} \times [\mathsf{Responses}]]$ is a dictionary for looking up authenticated sessions
- $\mathit{invalid\_sessions} \in [\mathcal{N}]$ stores which sessions were invalidated by logging out
- $\mathit{samlResponses} \in [\mathcal{N} \times \mathsf{Responses}]$ stores SAML Responses

Let $\mathit{acsset} := \{\texttt{/Shibboleth.sso/SAML2/POST}\}$. Let further $\mathit{acsGlobal} := \langle\{\mathsf{entityID}(p), \langle\{\texttt{URL}_s^{\mathsf{dom}(p)} | s \in \mathit{acsset}\}\rangle | p \in \mathsf{SP}\}\rangle$.
Let $\mathit{logoutSet} := \langle\{\texttt{URL}^{\mathsf{dom}(p)}_{\texttt{/Shibboleth.sso/SLO/Redirect}} | p \in \mathsf{SP}\}\rangle$. The initial state s_0^i of i is defined in equation (B.16).

$$\begin{aligned}s_0^i := \langle\langle\rangle, \langle\rangle, \mathsf{addr}(i), \mathit{keyMapping}, \langle\rangle, \mathit{tlskeys}^i, \mathit{userset}^i, \langle\rangle, \bot, \mathsf{signkey}(i),\\ \langle\{\langle \mathit{eID}, \mathsf{pub}(\mathsf{signkey}(p))\rangle | p \in \mathsf{SP}, \mathit{eID} \in \mathsf{entityID}(y)\}\rangle, \mathsf{dom}(i),\\ \mathit{acsGlobal}, \langle\rangle, \mathit{logoutSet}, \langle\rangle, \langle\rangle, \langle\rangle\rangle\end{aligned} \tag{B.16}$$

where *keyMapping* maps Domains to their respective public keys. ◇

Placeholder	Usage
ν_1	Session-ID to enable web flow
ν_2	ID for returned Response
ν_3	IdP Session Alias
ν_4	ID for returned Assertion
ν_5	ID for return Response
ν_6	ID for authenticated session
ν_7	Reference for Logout Request

Table B.2: Placeholders and their usage for an Identity Provider

The relation R^i is defined by Algorithm B.4. Table B.2 contains a list of all placeholders used in Algorithm B.4. In Algorithm B.5, the nonce ν_1 is used as a reference to store a SAML Response.

Algorithm B.4 Relation of an Identity Provider i

```
 1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s)
 2:     let s' := s
 3:     let sts := ⟨Strict-Transport-Security, ⊤⟩
 4:     let headers := ⟨sts⟩
 5:     let idp_id := s'.id
 6:     let n, method, path, params, mheaders, body such that
 7:             ⟨HTTPReq, n, method,host, path, params, mheaders, body⟩ ≡ m
 8:             if possible; otherwise stop
 9:     if path ≡ /Shibboleth.sso/SAML2/ArtifactResolve then
10:         let sresolve := body[Artifact]
11:         let resolve := extractmsg(sresolve)
12:         let sp := resolve.issuer
13:         if checksig(sresolve, s'.signkeys[sp]) ≡ ⊤ then
14:             let artifact := resolve.artifact
15:             let response := s'.samlResponses[resolve.artifact]
16:             if response.assertion.conditions.audience ≡ sp then
17:                 let resp := enc_s(⟨HTTPResp, n, 200, headers,
18:                   ⟨⟨ArtifactResponse, response⟩⟩⟩, k)
19:             end if
20:         end if
21:     else if path ∈ {/profile/SAML2/Redirect/SSO,
                        /profile/SAML2/POST/SSO}      then
22:         let request := ⊥
23:         if method ≡ GET ∧ params[SAMLRequest] ≢ ⊥ then
24:             let request := params[SAMLRequest]
25:         else if method ≡ POST ∧ body[SAMLRequest] ≢ ⊥ then
26:             let request := body[SAMLRequest]
27:         end if
28:         if request ≢ ⊥ ∧ request.destination.host ≡ s'.domain then
29:             let sessionid := ν_1
30:             let s'.sessions[sessionid] := ⟨⟨Request, request⟩⟩
31:             let redirect := URL_path^s'.domain
32:             let headers' := headers +^⟨⟩ ⟨Set-Cookie, ⟨⟨JSID, sessionid, ⊤, ⊤, ⊤⟩⟩⟩
33:             let resp := enc_s(⟨HTTPResp, n, 303, headers
34:                   +^⟨⟩⟨Location, redirect⟩, ⟨⟩⟩, k)
35:         else
36:             let jsessionid := mheaders[Cookie][JSID]
37:             let jsession := s'.sessions[jsessionid]
38:             let samlrequest := jsession[Request]
39:             let sp_id := samlrequest.general_request.issuer
40:             let dest := samlrequest.acsurl
41:             let inresponseto := samlrequest.general_request.id
42:             let shib_sessionid := mheaders[Cookie][shib_session]
43:             if shib_sessionid ≢ ⊥ then
44:                 let tmpl := s'.shib_sessions[shib_sessionid].1
45:                 let tmpl.assertion.conditions.audience := sp_id
46:                 let response := ⟨Response, ν_2, dest, idp_id, tmpl.assertion⟩
```

```
47:                let s′.shib_sessions[shib_sessionid] :=
48:                  s′.shib_sessions[shib_sessionid] +^⟨⟩ response
49:                let resp := call GET_HTTP_RESPONSE(s′, response, headers′, k)
50:            else if method ≡ GET ∧ mheaders[Cookie][JSID] ≢ ⊥ then
51:                let resp := enc_s(⟨HTTPResp, n, 200, headers′, ⟨script_idp_login⟩⟩, k)
52:            else if method ≡ POST ∧ mheaders[Origin] ≡ s′.domain then
53:                let user := body[user]
54:                let pass := body[pass]
55:                if ⟨user, pass⟩ ∈^⟨⟩ s′.users then
56:                    let session_idx := ν3
57:                    let assertion := ⟨Assertion, ν4, idp_id, user, ⟨Conditions, sp_id⟩,
58:                      ⟨AuthnStatement, session_idx⟩⟩
59:                    if dest ∈^⟨⟩ s′.acServices[sp_id] then
60:                        let endpoint := dest.path
61:                        let response := ⟨Response, ν5, inresponseto,
62:                          dest, idp_id, sig(assertion, s′.signkey)⟩
63:                        let shib_sessionid := ν6
64:                        let s′.shib_sessions[shib_sessionid] := ⟨response⟩
65:                        let headers′ := headers +^⟨⟩ ⟨Set-Cookie,
66:                          ⟨⟨shib_session, shib_sessionid, ⊤, ⊤, ⊤⟩⟩⟩
67:                        let resp := call GET_HTTP_RESPONSE(s′, response, headers′, k)
68:                    end if
69:                end if
70:            end if
71:        end if
72:    else if path ∈ {/profile/Logout, /profile/PropagateLogout,
                       /profile/SAML2/Redirect/SLO}                    then
73:        let shib_sessionid := mheaders[Cookie][shib_session]
74:        if shib_sessionid ≢ ⊥ then
75:            let responses := s′.shib_sessions[shib_sessionid]
76:            if path ≡ /profile/Logout then
77:                let services := ⟨⟩
78:                let serviceNames := ⟨⟩
79:                for all i ∈ ℕ such that responses.i ≢ ⊥ do
80:                    let service := ⟨URL, S, s′.domain, /profile/PropagateLogout, ⟨key, i⟩⟩
81:                    let services := services +^⟨⟩ service
82:                    let assertion := extractmsg(responses.i.assertion)
83:                    let serviceName := assertion.conditions.audience
84:                    let serviceNames := serviceNames +^⟨⟩ serviceName
85:                end for
86:                let invalid_sessions := invalid_sessions +^⟨⟩ shib_sessionid
87:                let scriptstate := ⟨services, serviceNames⟩
88:                let resp := enc_s(⟨HTTPResp, n, 200, headers, ⟨script_slo, scriptstate⟩⟩, k)
89:            else if path ≡ /profile/PropagateLogout then
90:                let index := params[key]
91:                let assertion := extractmsg(responses.index.assertion)
92:                let sessionidx := assertion.authn_statement.session_idx
93:                let subject := assertion.subject
```

```
94:                 let logoutrequest := ⟨LogoutRequest, ν7, idp_id, sessionidx, subject⟩
95:                 let s′.logoutRequests[ν7] := logoutrequest
96:                 let sp_id := assertion.conditions.audience
97:                 let sp_endpoint := s′.logoutServices[sp_id]
98:                 let srequest := sig(logoutrequest, s′.signkey)
99:                 let sp_endpoint.parameters := ⟨LogoutRequest, srequest⟩
100:                let headers′ := headers +^⟨⟩ ⟨Location, sp_endpoint⟩
101:                let resp := enc_s(⟨HTTPResp, n, 303, headers′, ⟨⟩⟩, k)
102:            else if path ≡ /profile/SAML2/Redirect/SLO then
103:                let signedresponse := params[SAMLResponse]
104:                let inresponseto, issuer, status such that
105:                    ⟨LogoutResponse, inresponseto, issuer, status⟩
106:                    ≡ extractmsg(signedresponse)
107:                if checksig(signedresponse, s′.signkeys[issuer]) then
108:                    let request := s′.logoutRequests[inresponseto]
109:                    if request ≢ ⊥ ∧ status ≡ ⊤ then
110:                        let s′.shib_sessions[shib_sessionid].i := ⊥
111:                    end if
112:                    let resp := enc_s(⟨HTTPResp, n, 200, headers,
113:                        ⟨script_slo_result, ⟨status⟩⟩⟩, k)
114:                end if
115:            end if
116:        end if
117:    end if
118:    stop ⟨⟨f, a, resp⟩⟩, s′
119: end function
```

Algorithm B.5 Relation of an Identity Provider i: Deriving the HTTP Response depending on the Binding used

```
1: function GET_HTTP_RESPONSE(s′, response, headers′, k )
2:     let dest := response.destination
3:     let endpoint := dest.path
4:     if endpoint ≡ /Shibboleth.sso/SAML2/POST then
5:         let scriptstate := ⟨dest, SAMLResponse, response⟩
6:         let resp := enc_s(⟨HTTPResp, n, 200, headers′,
7:                                 ⟨script_forward, scriptstate⟩⟩)
8:     else if endpoint ≡ /Shibboleth.sso/SAML2/Artifact then
9:         let s′.samlResponses[ν1] := response
10:        let redirect := ⟨URL, S, dest.host, endpoint,
11:                                 ⟨⟨Artifact, ⟨ν1, idp_id⟩⟩⟩, ⊥⟩
12:        let resp := enc_s(⟨HTTPResp, n, 303,
13:                                 headers′ +^⟨⟩ ⟨Location, redirect⟩, ⟨⟩⟩, k)
14:    end if
15: end function
```

B.2.10 Scripts

The set $\mathcal{S}$ of $\mathcal{SHIB}^p = (\mathcal{W}^p, \mathcal{S}, \mathsf{script}, E_0)$ consists of the scripts R^{att}, *script_idp_login*, *script_forward* with their string representations defined by script (see Table B.3). In the following, a definition for all honest scripts will be given.

Login Script

This scripts models the login process at the IdP. The user is prompted for a username and a password. The result is sent by submitting a HTML form. This script will now be formally defined.

Definition 13. A *scriptstate s of script_idp_login* is a term of the form

$$\langle\rangle$$

with the *initial scriptstate* $initState_{login}$ *of script_idp_login* being

$$initState_{login} = \langle\rangle$$

◇

The relation *script_idp_login* is defined in Algorithm B.6

Forwarding Script

This script is a simple HTML form with a hidden input for forwarding SAML Responses. In the following, a formal definition will be given.

Definition 14. A *scriptstate s of script_forward* is a term of the form

$$\langle destURL, name, message\rangle$$

$s \in \mathcal{S}$	script(s)
R^{att}	`att_script`
script_idp_login	`script_idp_login`
script_forward	`script_forward`
script_idp_select	`script_idp_select`
script_slo	`script_slo`
script_slo_result	`script_slo_result`

Table B.3: Scripts and their string representation

Algorithm B.6 Relation of *script_idp_login*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sStorage, id, secret\rangle$
let $s' := scriptstate$
let $host, protocol$ **such that**
  $\langle host, protocol\rangle \equiv \text{GETORIGIN}(tree, docnonce)$
  if possible: otherwise
  stop $\langle s', cookies, localStorage, sStorage, \langle\rangle\rangle$
let $endpoint := \text{PARENTWINDOW}(tree, docnonce)$`.documents.1.location.path`
let $url := \langle$`URL`$, protocol, host, endpoint, \langle\rangle, \bot\rangle$
let $body := \langle\langle$`user`$, id\rangle, \langle$`pass`$, secret\rangle\rangle$
let $command := \langle$`FORM`$, url,$ `POST`$, body,$ `_SELF`$\rangle$
stop $\langle s', cookies, localStorage, sStorage, command\rangle$

where $destURL \in$ URLs, $name \in \{$`SAMLRequest`, `SAMLResponse`$\}$, $message \in$ Responses. The *initial scriptstate* $initState_{forward}$ *of script_forward* is the state

$$\langle \bot, \texttt{SAMLRequest}, \bot\rangle$$

◇

The relation *script_forward* is defined in Algorithm B.7

Algorithm B.7 Relation of *script_forward*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sStorage, ids, secret\rangle$
let $s' := scriptstate$
let $url :=$ *scriptstate*`.destURL`
let $data := \langle\langle$*scriptstate*`.name`, *scriptstate*`.message`$\rangle\rangle$
let $command := \langle$`FORM`$, url,$ `POST`$, data,$ `_SELF`$\rangle$
stop $\langle s', cookies, localStorage, sStorage, command\rangle$

IdP selection

This script presents the user with the choice of multiple IdPs to login from. A formal definition will be given in the following.

Definition 15. A *scriptstate s of script_idp_select* is a term of the form

$$\langle idps\rangle$$

where $idps \in [\text{IDP} \times \text{Doms}]$ maps known IdPs to their respective domains. The *initial scriptstate* $initState_{select}$ *of script_idp_select* is the state

$$\langle\langle\rangle\rangle$$

◇

The relation *script_idp_select* is defined in Algorithm B.8

Algorithm B.8 Relation of *script_idp_select*

1: **Input:** $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sStorage, ids, secret\rangle$
2: **let** $s' := scriptstate$
3: **let** $idp \leftarrow \mathsf{IDP}$
4: **let** $url := \langle \texttt{URL}, \texttt{S}, s'\texttt{.idps}[idp], \texttt{/Shibboleth.sso/Login}, \langle \texttt{entityID}, idp\rangle, \bot\rangle$
5: **let** $command := \langle \texttt{HREF}, url, \texttt{\_SELF}, \top\rangle$
6: **stop** $\langle s', cookies, localStorage, sStorage, command\rangle$

Single Logout

This script shows the user all the services where she logged in during the current IdP session.

Definition 16. A *scriptstate s of script_slo* is a term of the form

$$\langle services, serviceNames\rangle$$

where $services \subset^{\langle\rangle} \langle \mathsf{URLs}\rangle$ is a list of all IdP endpoints that the SLO script has to perform Requests to in order to log out the user and $serviceNames \subset^{\langle\rangle} \langle \mathsf{SP}\rangle$ is a list of the corresponding SP entityIDs. The *initial scriptstate* $initState_{slo}$ *of script_slo* is the state

$$\langle\langle\rangle, \langle\rangle\rangle$$

$\diamond$

The relation *script_slo* is defined in Algorithm B.9

Algorithm B.9 Relation of *script_slo*

1: **Input:** $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sStorage, ids, secret\rangle$
2: **let** $s' := scriptstate$
3: **let** $command := \langle\rangle$
4: **if** $s'\texttt{.services} \not\equiv \langle\rangle$ **then**
5: **let** $service := s'\texttt{.services.1}$
6: **let** $serviceName := s'\texttt{.services.1}$
7: **let** $command := \langle \texttt{IFRAME}, service, \texttt{\_SELF}\rangle$
8: **remove** $service$ **from** $s'\texttt{.services}$
9: **remove** $serviceName$ **from** $s'\texttt{.serviceNames}$
10: **end if**
11: **stop** $\langle s', cookies, localStorage, sStorage, command\rangle$

SLO Result

This script outputs the result of the Logout process.

Definition 17. A *scriptstate s of script_slo_result* is a term of the form

$$\langle result \rangle$$

where $result \in \{\bot, \top\}$ is the result of the SLO run. The *initial scriptstate* $initState_{result}$ is the state

$$\langle \bot \rangle$$

◇

Algorithm B.10 Relation of *script_slo_result*

1: **Input:** $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sStorage, ids, secret \rangle$
2: **let** $s' := scriptstate$
3: **stop** $\langle s', cookies, localStorage, sStorage, command \rangle$

B.3 Important Events

In [8, Section 3.2] a set of events that mark important steps in an SSO flow are described. These events will be refined for Shibboleth in the following.

Definition 18 (Start of an SSO Flow)**.** Let $\mathcal{SHIB}^p$ be a Shibboleth Web system. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, b a browser, and p an SP. Write $\mathsf{started}(Q, b, p)$ iff in Q, the browser b is triggered and selects to open a URL, and this URL has the path `/secure` under a secure origin of p. ◇

Definition 19 (Selection of an IdP)**.** Let $\mathcal{SHIB}^p$ be a Shibboleth Web system. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, b a browser and i an IdP. Write $\mathsf{selectedIdP}(Q, b, i)$ iff in Q, the browser b is triggered and selects to run the script `script_idp_select` in some document and - in that script - in Line 3 in Algorithm B.8, i is selected from IDP ◇

Definition 20 (Selection of an Identity)**.** Let $\mathcal{SHIB}^p$ be a Shibboleth Web system. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, b a browser, i an IdP and id an identity. Write $\mathsf{selectedId}(Q, b, i, id)$ iff in Q, the browser b is triggered and selects to run the script `script_idp_login` in some document and - in that script - in Line 9 of Algorithm B.6, id is selected from the browser's identities. ◇

Definition 21 (User is logged in at SP)**.** Let $\mathcal{SHIB}^p$ be a Shibboleth Web system. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, b a browser, p an SP, id an identity and n some term. Write $\mathsf{logggedIn}(Q, b, p, id, n)$ iff in Q, the browser b receives an HTTPS Response from a secure origin (any SP only has one secure origin) where the Cookie `SID` is set to n, while in the state of p, for the subterm `sessions`, it holds that $\texttt{sessions}[n] \equiv id$ ◇

In Section 5.1.1, important events and predicates in an SLO flow are described. These are refined for Shibboleth in the following as well.

Definition 22. Let $\mathcal{SHIB}^p$ be a Shibboleth Web system with a network attacker. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, i an IdP, b a browser, and x some term. Write $\mathsf{secondaryLoggedIn}(Q, b, x, i)$ iff in Q, the browser b has a cookie named `shib_session` stored on the domain of i with a value of y and it holds that

$$s_i\texttt{.shib\_sessions}[y]\texttt{.1.assertion.authn\_statement.session\_idx} \equiv x$$

where s_i is the state of i in Q. ◇

Definition 23. Let $\mathcal{SHIB}^p$ be a Shibboleth Web system with a network attacker. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, i an IdP and b a browser. Write $\mathsf{logoutSummary}(Q, b, i)$ iff in Q, the browser b is triggered and selects to run the script *script_slo* in a document loaded on an origin of i and the subterm `services` of *script_slo*'s scriptstate is empty. ◇

For defining the positiveResult and negativeResult events, the notion of *labeled IFrames* has to be introduced first. This is due to the way Shibboleth presents the logout page.

Definition 24 (Labeled Iframes)**.** Let b be a browser. If in the function RUNSCRIPT ([8, Algorithm A.7]), Line 31 is reached and $s'.\overline{d}\texttt{.script} \equiv \texttt{script\_slo}$, denote by *id* the value of *window* there. Let *scriptstate_old* denote the term *in*`.3` and let *scriptstate_new* denote the term *state*′. Let

$$\mathit{diff} := \mathit{scriptstate\_old}\texttt{.serviceNames} \setminus^{\langle\rangle} \mathit{scriptstate\_new}\texttt{.serviceNames}$$

If $\mathit{diff} \not\equiv \langle\rangle$, write $\mathsf{iframeLabel}(\mathit{id}, \mathit{diff}\texttt{.1})$ ◇

Definition 25. Let $\mathcal{SHIB}^p$ be a Shibboleth Web System with a network attacker. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, i an IdP, b a browser and p a Service Provider. Write $\mathsf{positiveResult}(Q, b, i, p)$ iff in Q, b has a document loaded on an origin of i containing an IFrame with the window reference *id* which contains the script *script_slo_result* and for the scriptstate *scriptstate* of *script_slo_result* it holds that $\mathit{scriptstate}\texttt{.result} \equiv \top$ and it furthermore holds that $\mathsf{iframeLabel}(\mathit{id}, \mathsf{entityID}(p))$. ◇

Definition 26. Let $\mathcal{SHIB}^p$ be a Shibboleth Web System with a network attacker. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, i an IdP, b a browser and p a Service Provider. Write $\mathsf{negativeResult}(Q, b, i, p)$ iff in Q, b has a document loaded on an origin of i containing an IFrame with the window reference *id* and $\mathsf{positiveResult}(Q, b, i, p)$ is false and it holds that $\mathsf{iframeLabel}(\mathit{id}, \mathsf{entityID}(p))$. ◇

Definition 27. Let $\mathcal{SHIB}^p$ be a Shibboleth Web System with a network attacker. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, p a SP, i an IdP and x an arbitrary term. Write $\mathsf{authenticatedWith}(Q, p, i, x)$ iff in Q, there exists a nonce n such that for the state s_p of p in Q, it holds that $s_p.\texttt{assertions}[n].\texttt{authn\_statement}.\texttt{session\_idx} \equiv x$ and $s_p.\texttt{assertions}[n].\texttt{issuer} \equiv \mathsf{entityID}(i)$. ◇

Definition 28. Let $\mathcal{SHIB}^p$ be a Shibboleth Web System with a network attacker. Let ρ be a run of $\mathcal{SHIB}^p$. Let $Q \in \rho$ be a processing step, i an IdP and x an arbitrary term. Let s_i denote the state of i in Q Write $\mathsf{sessionTerminatedIdP}(Q, x, i)$ iff either there does not exist any nonce n such that Equation (B.17) holds

$$s_i.\texttt{shib\_sessions}[n].1.\texttt{assertion}.\texttt{authn\_statement}.\texttt{session\_idx} \equiv x \tag{B.17}$$

or there exists a nonce n such that Equation (B.17) holds and Equation (B.18) holds as well.

$$n \in^{\langle\rangle} s_i.\texttt{invalid\_sessions} \tag{B.18}$$

◇

B.4 Security Properties

B.4.1 Authentication

The definition of the Authentication property is equivalent to the definition given in [8].

Definition 29 (Authentication Property)**.** Let $\mathcal{SHIB}^p$ be a Shibboleth Web system with a network attacker (denoted by $\mathsf{attacker}$). $\mathcal{SHIB}^p$ is called *secure w.r.t. authentication* iff for every run ρ of $\mathcal{SHIB}^p$, every configuration (S, E, N) in ρ, every $p \in \mathsf{SP}$ that is honest in S, every browser $b \in \mathsf{B}$ that is not fully corrupted in S, every identity $id \in \mathsf{ID}$ owned by b with $\mathsf{governor}(id)$ being an honest IdP, every service session $\langle n, id\rangle$ at p, n is not derivable from the attacker's knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$). ◇

B.4.2 Session Integrity

The definition of Session Integrity is slightly weaker than the definition in [8]. Since no trusted third party delivers the scripts for logging in, session integrity cannot be fully guaranteed when the user chooses to log in at a malicious IdP. Furthermore, if the user is already logged in at the IdP, she cannot choose an identity in the flow, so the definition has to account for a shortened SSO flow. The set $\mathsf{SSOSessions}$ is defined as in [8]

Definition 30. Let $\mathcal{SHIB}^p$ be a Shibboleth Web system with a network attacker. $\mathcal{SHIB}^p$ is called *secure w.r.t. session integrity* iff for every run ρ of $\mathcal{SHIB}^p$, every processing step Q_{login} in ρ, for every browser b that is honest in Q_{login}, for every SP $p \in \mathsf{SP}$ that is honest in Q_{login}, and for every $id \in \mathsf{ID}$ the following holds true: If there exists some n such that $\mathsf{logggedIn}(Q_{\text{login}}, b, p, id, n)$, the following statements all hold true:

i) $\exists o \in \mathsf{SSOSessions}(\rho, b, p)$,

ii) $\exists Q_{\text{IdP}} \in o$ such that $\mathsf{selectedIdP}(Q_{\text{IdP}}, b, \mathsf{governor}(id))$,

iii) if $\mathsf{governor}(id)$ is honest in Q_{login} and $\mathsf{secondaryLoggedIn}(Q_{\text{login}}, b, x, i)$ is false, it holds that $\exists Q_{\text{id}} \in o$ such that $\mathsf{selectedId}(Q, b, \mathsf{governor}(id), id)$,

iv) $Q_{\text{login}} \in o$,

v) if $\mathsf{governor}(id)$ is honest in Q_{login}, b owns id.

◇

B.4.3 SLO Consistency

The definition of SLO Consistency is equivalent to the definition given in Section 5.1.2.

Definition 31 (SLO Consistency)**.** Let $\mathcal{SHIB}$ be an SSO Web system. $\mathcal{SHIB}$ is called *secure w.r.t. SLO Consistency* iff for every run ρ of $\mathcal{SHIB}$, every processing step Q_β, every IdP i that is honest in Q_β and every browser b that is honest in Q_β with $\mathsf{logoutSummary}(Q_\beta, b, i)$, the following holds true:
If there exists a processing step $Q_\alpha \in \rho$ and a term x such that $\mathsf{secondaryLoggedIn}(Q_\alpha, b, x, i)$, for all honest Service Providers p, exactly one of the following statements is true

1. There exists no processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$.
2. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{negativeResult}(Q_\beta, b, i, p)$.
3. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{positiveResult}(Q_\beta, b, i, p)$ and not $\mathsf{authenticatedWith}(Q_\beta, p, i, x)$.

Furthermore it holds that $\mathsf{sessionTerminatedIdP}(Q_\beta, x, i)$. ◇

B.5 Proof of Authentication Property

In the following, I will prove that Shibboleth is secure w.r.t. authentication.
I will assume that Shibboleth does not fulfill the autentication property and prove that this leads to a contradiction. Therefore I can make the following assumption: There is a run $\rho = s_0, s_1, \ldots$ of $\mathcal{SHIB}$, a step $s_j = (S_j, E_j, N_j)$ in ρ, a $p \in \mathsf{SP}$ that is honest in S_j, a session token of the form $\langle n, i \rangle$ recorded in p in the state $S_j(p)$ such that $\langle n, i \rangle \in d_{N^{\text{attacker}}}(S_j(\mathsf{attacker}))$ and the browser b owning i is not fully corrupted in S_j and

$\mathsf{governor}(i)$ is an honest IdP (called *idp*) in S_j.
By the definition of SPs, for $\langle n, i\rangle$ there are two ways in which the service token can be created. The first one is by using the HTTP-POST Binding:

Login with HTTP-POST There must exists a corresponding HTTPS Request req_{login} received by p and an HTTPS Response $resp_{\text{login}}$ issued by p. These have to be of the form described in Equations (B.19) and (B.20) where n_{login}, k are some nonces, $d_p \in \mathtt{dom}(p)$ is the (secure) domain of p, *response* $\in$ Responses is a SAML Response.

$$\begin{aligned} req^{\text{POST}}_{\text{login}} = &\text{enc}_{\text{a}}\big(\langle\langle \mathtt{HTTPReq}, n_{\text{login}}, \mathtt{POST}, d_p,\\ &\mathtt{/Shibboleth.sso/SAML2/POST}, \langle\rangle, [\mathtt{Origin} : \langle d_p, \mathtt{S}\rangle],\\ &[\mathtt{SAMLResponse} : \mathit{response}]\rangle, k\rangle, \mathtt{pub}(\mathtt{tlskey}(d_p))\big) \end{aligned} \tag{B.19}$$

$$\begin{aligned} resp^{\text{POST}}_{\text{login}} = \text{enc}_{\text{s}}(&\langle \mathtt{HTTPResp}, n_{\text{login}}, 200,\\ &[\mathtt{Set\text{-}Cookie} : \langle\langle \mathtt{SID}, n, \bot, \top, \top\rangle\rangle], \langle\rangle\rangle, k) \end{aligned} \tag{B.20}$$

Let $\mathit{sassertion} := \mathit{response}\,.\mathtt{assertion}$. Furthermore, there must exist a processing step of the form depicted in Equation (B.21) where $m \leq j$, $a_p \in \mathtt{addr}(p)$ and x is some address:

$$s_{m-1} \xrightarrow[p \to \{(x:a_p:resp^{\text{POST}}_{\text{login}})\}]{(a_p:x:req^{\text{POST}}_{\text{login}}) \to p} s_m \tag{B.21}$$

Login with HTTP-Artifact There must be an HTTPS Request req_{login} received by p and an HTTPS Response $resp_{\text{login}}$ issued by p. There must be an HTTPS Request req_{resolve} issued by p and a corresponding HTTPS Response $resp_{\text{resolve}}$ received by p. These have to be of the form described in Equations (B.22) through (B.25) where $n_{\text{login}}, k_{\text{login}}, n_{\text{resolve}}, k_{\text{resolve}}, k_{\text{sign}}$ are some nonces, $d_p \in \mathsf{dom}(p)$ is the (secure) domain of p, d_i is any domain, *response* $\in$ Responses is a SAML Response, *resolve* $\in$ ArtifactResolution is an Artifact Resolution Request.

$$\begin{aligned} req^{\text{Artifact}}_{\text{login}} = \text{enc}_{\text{a}}(\quad &\langle\langle \mathtt{HTTPReq}, n_{\text{login}}, \mathtt{GET}, d_p,\\ &\mathtt{/Shibboleth.sso/SAML2/Artifact}, [\mathtt{Artifact} : \mathit{artifact}],\\ &[\mathtt{Origin} : \langle d_p, \mathtt{S}\rangle], \langle\rangle\rangle, k_{\text{login}}\rangle, \mathsf{pub}(\mathsf{tlskey}(d_p))) \end{aligned} \tag{B.22}$$

$$\begin{aligned} resp^{\text{Artifact}}_{\text{login}} = \text{enc}_{\text{s}}(\quad &\langle \mathtt{HTTPResp}, n_{\text{login}}, 200,\\ &[\mathtt{Set\text{-}Cookie} : \langle\langle \mathtt{SID}, n, \bot, \top, \top\rangle\rangle], \langle\rangle\rangle, k_{\text{login}}) \end{aligned} \tag{B.23}$$

$$\begin{aligned} req_{\text{resolve}} = \text{enc}_{\text{a}}(\quad &\langle\langle \mathtt{HTTPReq}, n_{\text{resolve}}, \mathtt{POST}, d_i,\\ &\mathtt{/Shibboleth.sso/SAML2/ArtifactResolve}, \langle\rangle, \langle\rangle,\\ &[\mathtt{Artifact} : \text{sig}(\mathit{resolve}, k_{\text{sign}})]\rangle, k_{\text{resolve}}\rangle, \mathsf{pub}(\mathsf{tlskey}(d_i))) \end{aligned} \tag{B.24}$$

$$\begin{aligned} resp_{\text{resolve}} = \text{enc}_{\text{s}}(\quad &\langle \mathtt{HTTPResp}, n_{\text{resolve}}, 200, \langle\rangle,\\ &[\mathtt{ArtifactResponse} : \mathit{sassertion}]\rangle, k_{\text{resolve}}) \end{aligned} \tag{B.25}$$

Let *sassertion* := *response*.assertion. Furthermore there must exists two processing steps depicted in Equations (B.26) and (B.27) where $k < l \leq j$, $a_p \in \mathsf{addr}(p)$ and x, y are some addresses.

$$s_{k-1} \xrightarrow[p \to \{(y:a_p:req_{\text{resolve}})\}]{(a_p:x:req_{\text{login}}^{\text{Artifact}}) \to p} s_k \tag{B.26}$$

$$s_{l-1} \xrightarrow[p \to \{(x:a_p:resp_{\text{login}}^{\text{Artifact}})\}]{(a_p:y:resp_{\text{resolve}}) \to p} s_l \tag{B.27}$$

The Assertion *sassertion* must further fulfil Equations (B.28) and (B.29) because of Algorithm B.3, Line 21.

$$\text{extractmsg}(\mathit{sassertion}).\texttt{subject} = i \tag{B.28}$$

$$\text{extractmsg}(\mathit{sassertion}).\texttt{issuer} = \mathsf{entityID}(\mathit{idp}) \tag{B.29}$$

An honest SP only ever creates a session token in response to a (valid) Assertion, sent to it by some process r. p accepts the (signed) Assertion *sassertion* in s_m, which means that *sassertion* must be signed with a key k_{sign} such that Equation (B.30) holds for *idp*

$$\text{checksig}(\mathit{sassertion}, S_j(p).\texttt{signkeys}[\text{extractmsg}(\mathit{sassertion}).\texttt{issuer}]) \equiv \top \tag{B.30}$$

Together with Equation (B.29) this yields Equation (B.31)

$$\text{checksig}(\mathit{sassertion}, S_j(p).\texttt{signkeys}[\mathsf{entityID}(\mathit{idp})]) \equiv \top \tag{B.31}$$

Also, the subterm signkeys of p's state is never changed, so it holds true that $S_j(p).\texttt{signkeys} = S_0(p).\texttt{signkeys} = \{\langle d, \mathsf{pub}(\mathsf{signkey}(y))\rangle | y \in \mathsf{IDP}, d \in \mathsf{dom}(y)\}$. Together with Equation (B.30) this means that *sassertion* is signed with the (private) key $k_{\text{sign}} := \mathsf{signkey}(\mathit{idp})$.
The signed Assertion *sassertion* must be signed with the private key of *idp*. However, *idp* never leaks his private key, as he is honest, which can be seen from Algorithm B.4, where *idp* only uses its signkey in Lines 62 and 98 where the signkey is not sent. **So the only process being able to generate the signed Assertion** *sassertion* **is** *idp*.
Let's take a closer look at how an IdP generates Assertions: There are two ways in which an IdP emits an Assertion.

Assertion sent by HTTP-POST Assertions can be directly sent to a browser using the HTTP-POST Binding, which can be seen in Algorithm B.5, Line 6. Here, the Assertion is sent within the scriptstate of *script_forward*. Messages of this form are defined in the following.

Definition 32 (Assertion sent by POST)**.** Let $id \in \mathsf{ID}^{idp}$ be an identity governed by *idp*. Let $k \in \mathcal{N}$ be a TLS session key. Then the set $\mathsf{PostedResponses}^{id}$ contains all terms t that fulfil Equations (B.32) and (B.33).

$$\text{extractmsg}(\mathit{resp\_assertion}).\texttt{subject} \equiv id \tag{B.32}$$

$$\text{checksig}(\mathit{resp\_assertion}, k_{\text{sign}}) \equiv \top \tag{B.33}$$

where *resp_assertion* := $\text{dec}_{\text{s}}(t, k).\texttt{body.2.message.assertion}$ ◇

Lemma 1. For all processing steps of the form

$$s_{\alpha-1} \xrightarrow[idp \to \{(x:a_{idp}:resp^i_{\mathrm{POST}})\}]{(a_{idp}:x:req^i_{\mathrm{POST}}) \to idp} s_\alpha \tag{B.34}$$

(for some address x, a_{idp} with $\beta < j$ and $resp^i_{\mathrm{POST}} \in \mathsf{PostedResponses}^i$) it holds that req^i_{POST} was emitted by b.

Proof. There are two ways in which the HTTP Response $resp^i_{\mathrm{POST}}$ can be created, depending on whether req_{POST} contained a cookie `shib_session`.

The Request req^i_{POST} did not contain a cookie `shib_session`
When *idp* sends out $resp^i_{\mathrm{POST}}$, some conditions must be met:

1. The path has to be equivalent to `/profile/SAML2/Redirect/SSO` or `/profile/SAML2/POST/SSO` (Due to Line 21).
2. The method has to be `POST` (Due to Line 52).
3. The request body *body* has to fulfill $body[\texttt{user}] \equiv i$ (Due to Line 58).
4. The request body *body* has to fulfill $\langle body[\texttt{user}], body[\texttt{pass}]\rangle \in^{\langle\rangle} S_{\alpha-1}(idp)\texttt{.users}$ (Due to Line 55).
5. The state $S_{\alpha-1}(idp)$ must contain an Authentication Request *authnrequest* within its subterm `sessions` such that Equations (B.35), (B.36) and (B.37) hold (where $sp\_id \equiv authnrequest\texttt{.general\_request.issuer}$) for some $sid \in \mathcal{N}$

$$\exists sid \in \mathcal{N} : s_{\alpha-1}\texttt{.sessions}[sid] \equiv authnrequest \tag{B.35}$$

$$authnrequest\texttt{.acsurl} \in^{\langle\rangle} s_{\alpha-1}\texttt{.acServices}[sp\_id] \tag{B.36}$$

$$authnrequest\texttt{.acsurl.path} \equiv \texttt{/Shibboleth.sso/SAML2/POST} \tag{B.37}$$

(Due to Algorithm B.4, Line 59 and Algorithm B.5, Line 4).

The subterm `users` of an honest IdP is never changed, so it holds true that $S_{\alpha-1}(idp)\texttt{.users} \equiv S_0(idp)\texttt{.users} \equiv userset^{idp} \equiv \langle\{\langle u, \langle \mathsf{secretOfId}(u)\rangle\rangle | u \in \mathsf{ID}^{idp}\}\rangle$. Together with condition 3 and 4 this means that $body[\texttt{pass}] \equiv \mathsf{secretOfId}(i) =: sec$. This secret is initially only known to b and *idp*. From Algorithm B.4, one can see that an honest IdP never uses *sec* to generate messages, since Algorithm A.7, Lines 5 ff. are never reached.
Honest browsers release secrets only to scripts that are loaded from a specific origin. In this case, according to the initial state given in Section B.2.7, the secret *sec* is only released to scripts from the origin $\langle \mathsf{dom}(idp), \texttt{S}\rangle$ because *idp* is the governor of i. For any such script, with a Lemma from [8, Lemma 2] and the definition of the browser key mapping defined in Section B.2.7, one can see that any script that has access to the secret was sent by *idp*. *idp* can either deliver the script *script_idp_login* or *script_forward*. Neither of these scripts leak *sec*, as will be seen in the following.

script_forward does not access the subterm *secret* or the browser state at all (Algorithm B.7). Therefore it can not leak *sec* at all. *script_idp_login* accesses the subterm *secret* (and therefore *sec*) exactly once, in Algorithm B.6, Line 9. It is then only used as part of an HTTP Request to the browser's own origin. Since this origin is the origin on which the secret is stored, it must be $\langle \mathsf{dom}(idp), \mathtt{S} \rangle$. No data is stored in the scriptstate. So every entity that has access to *secret* never leaks it. Furthermore, as *idp* never creates any HTTPS Requests, b must have created req^i_{POST}.

The Request req_{POST} contained a cookie `shib_session`
Let *shib_session* denote the value of the `shib_session` cookie. *idp* only sends out $resp^i_{\mathrm{POST}}$, if Equation (B.38) holds. Let *sessions* := $S_{\alpha-1}(idp)$`.shib_sessions`

$$\mathrm{extractmsg}(sessions[shib\_session]\texttt{.assertion})\texttt{.subject} \equiv i\texttt{.subject} \tag{B.38}$$

The term *shib_session* is a nonce that is only sent out inside the `shib_session` cookie in a Response of the form of $resp^i_{\mathrm{POST}}$ and only if the corresponding Request did not contain the `shib_session` cookie. As seen previously, only b is able to send the corresponding Request and since the cookie is a secure cookie set on a secure origin of *idp*, it never leaks from b to any party other than *idp*. Therefore b is the only process that is able to send req^i_{POST}.

□

Assertion sent by ArtifactResolve Assertions can be sent out by the Artifact Resolution protocol. This can be seen in Algorithm B.4, Line 18. Here the Assertion is sent directly in the body of the Response as the `ArtifactResponse` parameter. HTTPS Responses of this form are defined in the following.

Definition 33 (ArtifactResolution message)**.** Let $id \in \mathsf{ID}^{idp}$ be an identity governed by *idp* and $k \in \mathcal{N}$ be a private TLS session key. Then the set $\mathsf{ArtifactResolutions}^{id}_{sp}$ contains all terms t that fulfil Equations (B.39) and (B.40).

$$\mathrm{extractmsg}(resp\_assertion)\texttt{.subject} \equiv id \tag{B.39}$$

$$\mathrm{extractmsg}(resp\_assertion)\texttt{.conditions.audience} \equiv sp \tag{B.40}$$

where $resp\_assertion := \mathrm{dec_s}(t,k)$`.body[ArtifactResponse].assertion` ◇

Lemma 2. For all processing steps of the form

$$s_{\beta-1} \xrightarrow[idp \to \{(x:a_{idp}:resp_{\mathrm{resolve}})\}]{(a_{idp}:x:req_{\mathrm{resolve}}) \to idp} s_\beta \tag{B.41}$$

(for some address x, $a_{idp} = \mathsf{addr}(idp)$, $\beta < j$ and $resp_{\mathrm{resolve}} \in \mathsf{ArtifactResolutions}^i_p$) it holds that p emitted req_{resolve}.

Proof. Let *resolve* be the Artifact Resolution Request inside req_{resolve}, as defined by Equation (B.42).

$$resolve := \text{dec}_{\text{a}}(req_{\text{resolve}}, k_{idp})\,.\,\texttt{body}[\texttt{Artifact}] \tag{B.42}$$

idp is honest, so it will only send out $resp_{\text{resolve}}$ if the body of req_{resolve} is signed with a key k_{resolve} (Algorithm B.4, Line 13). From Definition 33, Equation (B.40) one can see that the intended audience of the Assertion is p. Combined with Algorithm B.4, Line 16 this means that extractmsg(*resolve*)$\,.\,\texttt{issuer} \equiv p$ and due to Algorithm B.4, Line 13, the key k_{resolve} must be equivalent to $s_{\beta-1}\,.\,\texttt{signkeys}[\mathsf{entityID}(p)]$. Since *idp* is honest in $s_{\beta-1}$ (by assumption) it holds that

$$S_{\beta-1}(idp)\,.\,\texttt{signkeys} \equiv \{\langle eID, \mathsf{pub}(\mathsf{signkey}(p))\rangle | p \in \mathsf{SP}, eID \in \mathsf{entityID}(y)\}$$

So the key k_{resolve} must be equivalent to the signing key of p. Since p is honest as well in s_β, and never send out its private key anywhere, the only process being able to sign *resolve* is p. □

Definition 34 (Artifact sent by GET)**.** Let $art \in \mathcal{N}$ be a nonce. Let $k \in \mathcal{N}$ be a private TLS session key. Then the set $\mathsf{Artifacts}^{art}$ contains all terms t that fulfil Equation (B.43).

$$\text{extractmsg}(resp\_artifact)\,.\,\texttt{artifact} \equiv art \tag{B.43}$$

where $resp\_artifact := \text{dec}_{\text{s}}(t, k)\,.\,\texttt{body}[\texttt{Artifact}]$ ◇

Lemma 3. For all processing steps of the form

$$s_{\gamma-1} \xrightarrow[idp \to \{(x:a_{idp}:resp_{\text{Artifact}})\}]{(a_{idp}:x:req_{\text{Artifact}}) \to idp} s_\gamma \tag{B.44}$$

(for some address x, a_{idp} with $\gamma < j$ and some $art \in s_\gamma\,.\,\texttt{samlResponses}$ and $resp_{\text{Artifact}} \in \mathsf{Artifacts}^{art}$) where it holds that

$$assertion\,.\,\texttt{subject} \equiv i \tag{B.45}$$

(with $assertion :=$ extractmsg($s_\gamma\,.\,\texttt{samlResponses}[art]\,.\,\texttt{assertion}$)) it holds that req_{Artifact} was emitted by b.

Proof. This proof is very similar to the proof of Lemma 1. In the following, the differences will be outlined:
Condition 5, Equation (B.37) changes to Equation (B.46)

$$authnrequest\,.\,\texttt{acsurl.path} \equiv \texttt{/Shibboleth.sso/SAML2/POST} \tag{B.46}$$

□

Lemma 4. For the Assertion *sassertion* it holds true that

$$\text{extractmsg}(sassertion)\,.\,\texttt{conditions.audience} \equiv \mathsf{entityID}(p) \tag{B.47}$$

Proof. This lemma follows directly from the assumption (p accepts *sassertion*) and the definition of SPs (Algorithm B.1, Line 12). □

Lemma 5. Let $sp\_id \in \mathsf{EntityIDs}$ be an EntityID. Let $k \in \mathcal{N}$ be a TLS session key. Let $assertion \in \mathsf{Assertions}$ be an Assertion such that $assertion\texttt{.conditions.audience} \equiv sp\_id$ holds. Let *scriptstate* be a scriptstate for *script_forward* such that $scriptstate\texttt{.message.assertion} \equiv assertion$. Let *resp* be an HTTPS Response such that $\mathrm{dec_s}(resp, k)\texttt{.body.scriptstate} \equiv scriptstate$. Then it holds that for all processing steps of the form

$$s_{\delta-1} \xrightarrow[idp \to \{(x:a_{idp}:resp)\}]{(a_{idp}:x:req) \to idp} s_\delta \tag{B.48}$$

(for some addresses x, a_{idp} and $\delta \leq j$) $\mathsf{dom}(\mathsf{entityID}^{-1}(sp\_id))$ is equivalent to $scriptstate\texttt{.destURL.domain}$.

Proof. The IdP *idp* is honest in s_δ. The term *scriptstate* is set in Algorithm B.5, Line 5. There, the subterm `destURL` is set to a value *dest* such that $dest \in S_{\delta-1}(idp)\texttt{.acServices}[sp\_id]$. From the definition of the intial state of IdPs (note that the subterm `acServices` never changes for *idp*), it follows that $dest\texttt{.domain} \equiv \mathsf{dom}(\mathsf{entityID}^{-1}(sp\_id))$ □

Lemma 6. Let $sp\_id \in \mathsf{EntityIDs}$ be an EntityID. Let $sp := \mathsf{entityID}^{-1}(sp\_id)$ be the corresponding process. Let $k \in \mathcal{N}$ be a TLS session key. Let $art \in \mathcal{N}$ be a nonce. Then it holds that for all processing steps of the form

$$s_{\epsilon-1} \xrightarrow[idp \to \{(x:a_{idp}:resp)\}]{(a_{idp}:x:req) \to idp} s_\epsilon \tag{B.49}$$

(for some addresses x, a_{idp}, $\epsilon \leq j$ and $resp \in \mathsf{Artifacts}^{art}$) where it holds that

$$assertion\texttt{.conditions.audience} \equiv sp\_id \tag{B.50}$$

(with $assertion := \mathrm{extractmsg}(s_\epsilon\texttt{.samlResponses}[art]\texttt{.assertion})$) it holds that

$$\mathsf{dom}(\mathsf{entityID}^{-1}(sp\_id)) \equiv \mathrm{dec_s}(resp, k)\texttt{.headers}[\texttt{Location}]\texttt{.domain} \tag{B.51}$$

Proof. This proof is analogous to the proof for Lemma 5. □

Lemma 7. The attacker is unable to derive $req^{\mathrm{POST}}_{\mathrm{login}}$ in s_{j-1}.

Proof. In order to be able to derive $req^{\mathrm{POST}}_{\mathrm{login}}$, the attacker must be able to derive *sassertion*. As noted previously, the only process able to generate *sassertion* is *idp*. *idp* sends this Assertion either by HTTP-POST or by ArtifactResolve. For HTTP-POST, one can see from Lemma 1 that only b is able to get the Assertion this way. For ArtifactResolve, one can see from Lemma 2 that only p is able to get the Assertion this way. p never leaks *sassertion*, since p is honest in s_{j-1}. b is not fully corrupted in s_{j-1}. The only script running in b with access to *sassertion* is *script_forward*. This

script issues a POST Request with *sassertion* to *scriptstate*.destURL. By Lemma 4 it holds that extractmsg(*sassertion*).conditions.audience ≡ entityID(p) From Algorithm B.3, Line 12, one can see that extractmsg(*sassertion*).conditions.audience must be equivalent to p. By Lemma 5, *scriptstate*.destURL is therefore equivalent to dom(p).
The script *script_forward* only sends out one HTTPS Request to *scriptstate*.destURL, which is encrypted under the public TLS key of p which is not known to the attacker in s_{j-1}. Therefore there is no way for the attacker to derive $req_{\text{login}}^{\text{POST}}$ □

Corollary 1. The attacker can not derive *sassertion* in s_{j-1}

Lemma 8. The Browser b is honest in s_{k-1}

Proof. Due to the Assumption, the browser can not be fully corrupted in s_{k-1}. This only leaves the possibility of a close-corrupted browser. However, this would lead to a contradiction, as will be seen in the following. Assume that b becomes close-corrupted in s_z with $z \leq k-1$. All session cookies are cleared at this point, which means that in particular the cookie LOGIN, set by p, is cleared. Furthermore, all secrets are cleared. This means that after step s_z, b is unable to log in at *idp*. Due to Lemma 1 and Lemma 3, b must have logged in at *idp*, because *sassertion* must have been generated by *idp*. Therefore, p must have issued an Authentication Request *authn_request* ∈ AuthnRequests in a step s_y such that $y < z$. Due to Algorithm B.1, Lines 21 ff., there is a nonce $n' \in \mathcal{N}$ such that $S_y(p)$.logins[n'] ≡ *authn_request*. This nonce n' never leaks to the attacker, since it is only sent out in a cookie on a secure origin. When p receives $req_{\text{login}}^{\text{Artifact}}$ in s_{k-1}, it only sends out $resp_{\text{login}}^{\text{Artifact}}$ in s_l if n' is stored in the cookie LOGIN, as can be seen from Algorithm B.3, Lines 6 f. However since the browser is close-corrupted in s_{k-1}, there is no way that the cookie LOGIN can be correctly set, which leads to a contradiction. □

Lemma 9. The attacker is unable to derive $req_{\text{login}}^{\text{Artifact}}$ in s_{k-1}.

Proof. The term $req_{\text{login}}^{\text{Artifact}}$ contains an *artifact* ∈ Artifact which the attacker must derive. Upon receiving this nonce, p sends out a message req_{resolve}, which some process *idp*′ must answer with *sassertion*. Due to Corollary 1, the attacker is unable to derive *sassertion* in s_{k-1}. The only process being able to send $resp_{\text{resolve}}$ is *idp*. However, due to Lemma 2, the only process that is able to send req_{resolve} is p. The Artifact *artifact* contains a nonce *art* such that $S_\zeta(idp)$.samlResponses[*art*] ≡ *sassertion* holds for some $\zeta < l-1$. This nonce must have been generated by *idp*, which only sends it out in Algorithm B.5, Line 13. Due to Lemma 3, the only process that is able to send the corresponding HTTPS Request is b. Let $resp^*$ be the HTTPS Response in which this Artifact is contained. By Lemma 4 it holds that extractmsg(*sassertion*).conditions.audience ≡ entityID(p) and therefore, due to Lemma 6 it holds that

$$\text{dec}_{\text{s}}(resp^*, k).\text{headers}[\text{Location}].\text{domain} \equiv \text{dom}(p) \tag{B.52}$$

The Browser b therefore sends *art* to p. Let *req* be the HTTPS Request which b issues to p which contains *art* within an Artifact message *artifact′*. p only sends out *art* in Algorithm $B.1$, Line 41 in a message that will be called *msg* in the following. The message is sent to $idp' := \mathsf{entityID}^{-1}(artifact'.\texttt{idp})$ (The subterm `idps` never changes for p). However, since *artfact′* is created by *idp*, it holds that $artifact'.\texttt{idp} \equiv idp$. The Message *msg* is sent from p to *idp* via HTTPS, which means that the attacker can not derive *art* from *msg* either. Note that due to Lemma 8, the attacker can not derive *art* from the `history` subterm of b by issuing a `CLOSECORRUPT` message to b. Since there is no way for the attacker to derive *art*, he can not derive $req_{\text{login}}^{\text{Artifact}}$ in s_{k-1} □

Combining Lemma 7 and Lemma 9 it is easy to see that the attacker can not derive $resp_{\text{login}}^{\text{POST}}$ or $resp_{\text{login}}^{\text{Artifact}}$ since they contain a nonce unknown to the attacker in s_j. All subsequent requests to p are made via HTTPS, so there is no way that n can leak from the cookie. Therefore the attacker can not derive n which completes the contradiction.
■

B.6 Proof of Session Integrity Property

In the following, I will prove that Shibboleth is secure w.r.t. session integrity.
Let $\rho \in \mathcal{SHIB}$ be any run. By assumption, there exists some n and some $Q_{\text{login}} \in \rho$ such that $\mathsf{logggedIn}(Q_{\text{login}}, b, p, id, n)$ holds. Since the session integrity of Shibboleth heavily relies on cookies, a useful Lemma about integrity of cookies in the presence of the Strict-Transport-Security header will be proven first.

Lemma 10. Let Q_α, Q_β be processing steps with $\alpha < \beta$. Let $p \in \mathsf{SP} \cup \mathsf{IDP}$ be an Identity Provider or Service Provider which is honest in Q_β. Let $b \in \mathsf{B}$ be a browser that is honest in Q_β. If in Q_α, p has sent a HTTPS Response to b which sets a cookie with the name *cname* to the value *cvalue*, it holds that only p is able to set *cname* to a different value in any processing step Q_γ with $\alpha < \gamma \leq \beta$.

Proof. From the definition of R^i or R^p, one can see that every IdP and every SP sends out a $\langle \texttt{Strict-Transport-Security}, \top \rangle$ header in each HTTPS Response. Therefore, if p has set the cookie with the name *cname*, p has also sent out the `Strict-Transport-Security` header to b. b stores the domain of p in the `sts` subterm of its state. Therefore, any subsequent access to the domain of p is made via HTTPS. In particular, since only p is able to create HTTPS Responses for the subsequent Requests of b, only p is able to set *cname* to a different value. □

Similarly to the proof in Section B.5, there are two ways in which this service token can be created by p.

Login with HTTP-POST Let $req_{\text{login}}^{\text{POST}}$, $resp_{\text{login}}^{\text{POST}}$, s_{m-1}, *response* and *sassertion* be defined as in Equations (B.19) through (B.21).

Login with HTTP-Artifact Let $req_{\text{login}}^{\text{Artifact}}$, $resp_{\text{login}}^{\text{Artifact}}$, req_{resolve}, $resp_{\text{resolve}}$, *response*, *sassertion*, s_{k-1}, s_{l-1} be defined as in Equations (B.22) through (B.27).

The Assertion *sassertion* must further fulfil Equations (B.28) and (B.29) because of Algorithm B.3, Line 21.

$$\text{extractmsg}(\mathit{sassertion})\texttt{.subject} = \mathit{id} \tag{B.53}$$

$$\text{extractmsg}(\mathit{sassertion})\texttt{.issuer} = \mathsf{entityID}(\mathit{idp}) \tag{B.54}$$

From Algorithm B.3, Line 9 it further follows that

$$\mathit{response}\texttt{.inresponseto} \equiv \mathit{authn\_request}\texttt{.general\_request.id} \tag{B.55}$$

must hold.
Let $n' \in \mathcal{N}$ be a nonce, $\mathit{idp\_id} \in \mathsf{EntityIDs}$ and $\mathit{authn\_request} \in \mathsf{AuthnRequests}$. When p receives $\mathit{req}_{\text{login}}^{\text{Artifact}}$ in s_{k-1}, it only sends out $\mathit{resp}_{\text{login}}^{\text{Artifact}}$ in s_l if n' is stored in the cookie `LOGIN` and it holds that $S_l(p)\texttt{.logins}[n'] \equiv \langle \mathit{authn\_request}, \mathit{idp\_id} \rangle$, as can be seen from Algorithm B.3, Lines 6 f. When p receives $\mathit{req}_{\text{login}}^{\text{POST}}$ in s_{m-1}, it only sends out $\mathit{resp}_{\text{login}}^{\text{POST}}$, if n' is stored in the cookie `LOGIN` and it holds that $S_m(p)\texttt{.logins}[n'] \equiv \langle \mathit{authn\_request}, \mathit{idp\_id} \rangle$, as can be seen from Algorithm B.3, Lines 6 f. The `LOGIN` cookie is only set in response to a Request to `/Shibboleth.sso/Login` which is sent on the origin of p. The only script that is able to send this Request is *script_idp_select*, which in turn is only delivered to b, if b opens `/secure` on p. This means in turn that there exists a processing step Q'_{started} such that $\mathsf{started}(Q'_{\text{started}}, b, p)$ holds. Let Q_{started} be the latest processing step that fulfils this property. Then it holds that there exists an SSO session $o \in \mathsf{SSOSessions}(\rho, b, p)$ such that $Q_{\text{login}} \in o$ holds.
Let $\mathit{idp} := \mathsf{governor}(\mathit{id})$.

Lemma 11. There exists a processing step Q_α in o such that $\mathsf{selectedIdP}(Q_\alpha, b, \mathit{idp})$ holds.

Proof. The cookie `LOGIN` is set on a secure origin of p with $\texttt{secure} \equiv \top$, so it must have been set by p. The only place that p sets `LOGIN` is at Algorithm B.1, Lines 21 ff. There, a term $\langle \mathit{authn\_request}', \mathit{idp\_id}' \rangle$ is stored in the subterm $\texttt{logins}[n']$. These Lines are reached when p processes an HTTPS Request *req* to `/Shibboleth.sso/Login` which must have been issued by b. The only script that is able to deliver *req* is *script_idp_select*. Let $\mathit{idp}' := \mathsf{entityID}^{-1}(\text{dec}_{\text{a}}(\mathit{req}, \mathsf{tlskey}(\mathsf{dom}(p)))\texttt{.params}[\texttt{entityID}])$. Note that there exists a processing step Q'_α in o such that $\mathsf{selectedIdP}(Q_\alpha, b, \mathit{idp}')$ holds. Furthermore it holds that $\mathsf{entityID}(\mathit{idp}') \equiv \mathit{idp\_id}$. In the following, I will prove that $\mathit{idp} \equiv \mathit{idp}'$ holds.
Lines 21 ff. are the only place where the subterm `logins` of p's state is changed. If there was a later Request to `/Shibboleth.sso/Login`, it would change `logins`, but if the new cookie does not get delivered to b, p will not accept *sassertion* later. However, b only sends out a Request to `/Shibboleth.sso/Login` once, after selecting an IdP in *script_idp_select*. Any Requests from other origins are blocked by p by checking the `Origin` header. This means that $\mathit{authn\_request} \equiv \mathit{authn\_request}'$ and $\mathit{idp\_id} \equiv \mathit{idp}_i d'$ holds. This directly implies that $\mathit{idp}' \equiv \mathit{idp}$. □

Definition 35. Let $\mathit{req\_id} \in \mathcal{N}$ be a nonce. Let $k \in \mathcal{N}$ be a TLS session key. Then the set $\mathsf{PostedResponses}_{\mathit{req\_id}}^{id'}$ contains all terms $t \in \mathsf{PostedResponses}^{id'}$ that fulfil Equation (B.56).

$$\text{dec}_{\text{s}}(t, k)\texttt{.body.2.message.inresponseto} \equiv \mathit{req\_id} \tag{B.56}$$

◇

Lemma 12 (Session Integrity for IdP)**.** For all identities $id' \in \mathsf{ID}$, for every run $\rho' \in \mathcal{SHIB}^p$, it holds that if there exists a processing step as described in Equation (B.57)

$$s_{z-1} \xrightarrow[idp \to \{(x:a_{idp}:resp^*)\}]{(a_{idp}:x:req^*) \to idp} s_z \tag{B.57}$$

(for some addresses x, a_{idp}, $z < k$, $req\_id \in \mathcal{N}$, $resp^* \in \mathsf{PostedResponses}^{id'}_{req\_id}$), and idp is honest and b has sent req^* and $\mathsf{secondaryLoggedIn}(Q_{\text{login}}, b, x, i)$ is false, then all of the following statements are true:

i) $\exists Q_y \in \rho' : \mathsf{selectedId}(Q_y, b, idp, id')$ where $y \leq z$

ii) b has sent an Authentication Request $authn\_request^* \in \mathsf{AuthnRequests}$ to idp in a step Q_x with $x \leq y$ and $authn\_request^*.\texttt{id} \equiv req\_id$

iii) b owns id'

iv) b and idp are the only processes that are able to decrypt $resp^*$

Proof. Note that idp is honest in Q_z (by assumption). Since $\mathsf{secondaryLoggedIn}(Q_{\text{login}}, b, x, i)$ is false, Algorithm B.4, Lines 44 ff. are not reached. Thus, $resp^*$ must have been sent out via lines 52 ff. req^* must have the `Origin` header set to $\mathsf{dom}(idp)$ (Line 52) The parameter *user* must be set to be equivalent to id' for req^* (Line 58). Therefore the Request req^* must have been sent by a script delivered on the origin of idp. The only script that is able to create such a Request is *script_login*. From the definition of *script_login* one can see that there exists a processing step Q_y such that $\mathsf{selectedId}(Q_y, b, idp, id')$ holds, which proves implication i).
idp only sends out $resp^*$ if req^* contains a cookie `JSID` with a value $jsid$ such that $jsid \in S_{z-1}(idp).\texttt{sessions}$ (Lines 36).
Let $authn\_request^{**} := S_{z-1}(idp).\texttt{sessions}[jsid][\texttt{Request}]$. Let $authn\_request^*$ be defined as in ii). In the following, I will prove that $authn\_request^{**} \equiv authn\_request^*$. Note that $authn\_request^{**}.\texttt{id} \equiv req\_id$ due to Line 61. The Cookie `JSID` has the `secure` attribute set to $\top$, so it was set by idp on a secure origin of idp. The `JSID` cookie is only sent in Lines 32 ff. These Lines are only reached, if an Authentication Request is contained in the HTTPS Request. The Authentication Request is stored in the `sessions` subterm of idp's state. Note that the `sessions` subterm is never modified anywhere else, since idp is honest. Therefore, the Request $authn\_request^{**}$ was stored in some processing step Q_x with $x < z$ in the subterm $\texttt{sessions}[jsid]$ and idp has sent out a Response that sets the `JSID` cookie to $jsid$. Since this is the only time that b is able to obtain this cookie with a value of $jsid$ and due to Lemma 10, b must have created the request and $authn\_request^* \equiv authn\_request^{**}$ holds. This completes the proof for implication ii).
Since b only selects an identity that b actually owns, implication iii) holds.
Implication iv) follows directly from [8, Lemma 1]. □

Lemma 13. For all identites $id' \in \mathsf{ID}$, for every run $\rho' \in \mathcal{SHIB}^p$, it holds that if there exists a processing step as described in Equation (B.58)

$$s_{z-1} \xrightarrow[idp \to \{(x:a_{idp}:resp^*)\}]{(a_{idp}:x:req^*) \to idp} s_z \tag{B.58}$$

(for some addresses x, a_{idp}, with $z < k$, some $art \in S_z(idp).\texttt{samlResponses}$ and $resp_{\text{Artifact}} \in \mathsf{Artifacts}^{art}$), where Equations (B.59) and (B.60) hold

$$\text{extractmsg}(S_z(idp).\texttt{samlResponses}[art].\texttt{assertion}) \equiv id' \tag{B.59}$$

$$S_z(idp).\texttt{samlResponses}[art] \equiv req\_id \tag{B.60}$$

and idp is honest in Q_{login} and b has sent req^*, and $\mathsf{secondaryLoggedIn}(Q_{\text{login}}, b, x, i)$ is false, then all of the following statements are true:

i) $\exists Q_y \in \rho' : \mathsf{selectedId}(Q_y, b, idp, id')$ where $y \leq z$

ii) b has sent an Authentication Request $authn\_request^* \in \mathsf{AuthnRequests}$ to idp in a step Q_x with $x \leq y$ and $authn\_request^*.\texttt{id} \equiv req\_id$

iii) b owns id'

iv) b and idp are the only processes that are able to decrypt $resp^*$

Proof. The proof is analogous to the proof for Lemma 12 □

Lemma 14. If idp is honest, there exists a processing step Q_β in o such that $\mathsf{selectedId}(Q_\beta, b, idp, id)$ holds and b owns id.

Proof. Depending on the way in which the service token was created, b either issued a Request $req_{\text{login}}^{\text{POST}}$ or a Request $req_{\text{login}}^{\text{Artifact}}$ to p.

Login with HTTP-POST b issued an HTTPS Request $req_{\text{login}}^{\text{POST}}$ containing a SAML Response $response$. By a similar argument as in the authentication proof, idp is the only process that is able to create $sassertion$. This means that b must have obtained $sassertion$ from idp, which in turn must have sent out $script\_forward$, containing the $response$ in its scriptstate. idp only does so when receiving a Request req^* by b. From Lemma 12, it follows that there exists a processing step Q_β in o such that $\mathsf{selectedId}(Q_\beta, b, idp, id)$. Furthermore, it follows from the same Lemma that b owns id.

Login with HTTP-Artifact b issued an HTTPS Request $req_{\text{login}}^{\text{Artifact}}$ containing a SAML Artifact art. By a similar argument as in the authentication proof (Lemma 9), idp is the only process that is able to create $sassertion$. This means that b must have obtained art from idp, which in turn must have sent out a HTTPS Response $resp^*$ such that $resp^* \in \mathsf{Artifacts}^{art}$. idp only does so when receiving a Request req^* by b. From Lemma 13, it follows that there exists a processing step $Q_\beta$in o such that $\mathsf{selectedId}(Q_\beta, b, idp, id)$ holds. Furthermore, it follows from the same Lemma that b owns id.

□

Combining Lemma 11 and Lemma 14 one can see that the remaining implications hold as well.

■

B.7 Proof of SLO Consistency Property

Every time an authenticated session at an IdP is generated, a corresponding session identifier is generated. This can be seen from Algorithm B.4, Lines 56 ff. This means that for each *shib_session* $\in \mathcal{N}$ there exists exactly one *session_idx* such that Equation (B.61) holds for all $n \geq n_0$ (for some n_0) and any IdP i.

$$\begin{aligned} &\text{extractmsg}(S_n(i)\,.\,\texttt{shib\_sessions}[\textit{shib\_session}]\,.\,\texttt{1}\,.\,\texttt{assertion}) \\ &\qquad .\texttt{authn\_statement}\,.\,\texttt{session\_idx} \equiv \textit{session\_idx} \end{aligned} \tag{B.61}$$

This can be captured by bijective mapping $\mathsf{sessionAlias} : \mathcal{N} \to \mathcal{N}$ which maps the *shib_session* to the corresponding *session_idx*. Next, a useful lemma about tracking of SP sessions at IdPs will be proven, it is given in Lemma 15.

Lemma 15. Let Q_α, Q_γ be processing steps with $\alpha \leq \gamma$. Let $p \in \mathsf{SP}$ be a Service Provider that is honest in Q_γ. Let $i \in \mathsf{IDP}$ be an Identity Provider that is honest in Q_γ. Let $k := \mathsf{signkey}^{\mathrm{sp}}(p)$. Let x be an arbitrary term. Let *shib_session* $\in \mathcal{N}$ be a nonce. Let *responses* $:= S_\gamma(i)\,.\,\texttt{shib\_sessions}[\textit{shib\_session}]$. If it holds that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and there does not exist a processing step Q_β with $\beta < \gamma$ in which i receives a Logout Response $r \in \mathsf{SignedLogoutResponse}_k$, in an HTTPS Request sent to `/profile/SAML2/Redirect/SLO` with a cookie `shib_session` set to *shib_session*, it holds true that there exists an $j \in \mathbb{N}$ such that Equation (B.62) holds.

$$\text{extractmsg}(\textit{responses}\,.\,j\,.\,\texttt{assertion})\,.\,\texttt{conditions}\,.\,\texttt{audience} \equiv p \tag{B.62}$$

Proof. From the assumption it is known that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ holds. Inserting Definition 5, one can see that there exists a nonce n such that Equations (B.63) and (B.64) hold. Let *passertion* $:= S_\alpha(p)\,.\,\texttt{assertions}[n]$

$$\textit{passertion}\,.\,\texttt{authn\_statement}\,.\,\texttt{session\_idx} \equiv x \tag{B.63}$$

$$\textit{passertion}\,.\,\texttt{issuer} \equiv \mathsf{entityID}(i) \tag{B.64}$$

Honest Service Providers only accept valid, signed Assertions. From Algorithm B.3, Line 12 one can see that Equation (B.65) holds.

$$\textit{passertion}\,.\,\texttt{conditions}\,.\,\texttt{audience} \equiv p \tag{B.65}$$

With a similar argumentation as in the proof for the Authentication property, it can be shown that the only process that is able to generate *passertion* is i. This means that there must exist a processing step Q_δ with $\delta \leq \alpha$ where i emitted the Assertion. i only emits a SAML Response (containing an Assertion) within the GET_HTTP_RESPONSE function. Every time this function is called, the SAML Response is also stored within the subterm `shib_sessions` of i's state. Therefore, there must exist a nonce *shib_session*$'$ and a $j' \in \mathbb{N}$ such that Equation (B.66) holds.

$$\text{extractmsg}(S_\delta(i)\,.\,\texttt{shib\_sessions}[\textit{shib\_session}']\,.\,j'\,.\,\texttt{assertion}) \equiv \textit{passertion} \tag{B.66}$$

Together with Equation (B.65), Equation (B.67) follows.
Let $responses' := S_\delta(i).\texttt{shib\_sessions}[shib\_session]$

$$\text{extractmsg}(responses.i.\texttt{assertion}).\texttt{conditions.audience} \equiv p \tag{B.67}$$

This means it is only left to show that Equation (B.68) holds.

$$responses'.j' \overset{!}{\equiv} responses.j \tag{B.68}$$

As can be seen from the definition of $\mathsf{sessionAlias}$, it holds that $shib\_session' \equiv \mathsf{sessionAlias}^{-1}(x) \equiv shib\_session$. From the definition of IdPs it can be seen that elements of the `shib_sessions` subterm of i's state are never removed. Since Q_δ is before Q_γ and the stored Response does not get removed, Equation (B.68) holds indeed. □

Lemma 16. Let Q_α be a processing step. Let $i \in \mathsf{IDP}$ be an IdP that is honest in Q_α. Let $id \in \mathcal{N}$ be a nonce. Then there exists at most one $req \in \mathsf{SignedLogoutRequest}_{\mathsf{signkey}^{\mathrm{idp}}(i)}$ with $\text{extractmsg}(req).\texttt{id} \equiv id$ in Q_α

Proof. To create *req*, a process needs the `signkey` of i. Since i is honest in Q_α and therefore has not leaked its `signkey`, i is the only process that is able to create *req*. Honest IdPs only create Logout Requests in Algorithm B.4, Line 94. There, a fresh nonce is used for the `id` subterm, hence there exists at most one Logout Request with this `id` in Q_α. □

Lemma 16 allows the definition of an *associated Logout Request*.

Definition 36. Let Q_α be a processing step. Let $i \in \mathsf{IDP}$ be an IdP that is honest in Q_α. Let $req \in \mathsf{SignedLogoutRequest}_{\mathsf{signkey}^{\mathrm{idp}}(i)}$ be a (signed) Logout Request. For a Logout Response $resp \in \mathsf{SignedLogoutResponse}$ with $\text{extractmsg}(resp).\texttt{inresponseto} \equiv \text{extractmsg}(req).\texttt{id}$ define the *associated Logout Request* by:

$$\mathsf{associatedLogoutRequest}(Q_\alpha, resp) := req$$

◇

Definition 37. Let $assertion \in \mathsf{Assertions}$. Let $sreq \in \mathsf{SignedLogoutRequest}$.
The Logout Request *sreq* is said to *match assertion*, iff Equations (B.69) through (B.71) hold.

$$req.\texttt{subject} \equiv assertion.\texttt{subject} \tag{B.69}$$

$$req.\texttt{issuer} \equiv assertion.\texttt{issuer} \tag{B.70}$$

$$req.\texttt{session\_idx} \equiv assertion.\texttt{authn\_statement.session\_idx} \tag{B.71}$$

(where $req := \text{extractmsg}(sreq)$). Write $\mathsf{matchesAssertion}(sreq, assertion)$ iff *sreq* matches *assertion* ◇

Lemma 17. Let Q_α, Q_β be processing steps with $\alpha < \beta$. Let $i \in \mathsf{IDP}$ be an IdP that is honest in Q_β. Let $p \in \mathsf{SP}$ be a SP that is honest in Q_β. Let $req \in \mathsf{SignedLogoutRequest}_{\mathsf{signkey}^{\mathrm{idP}}(i)}$ with extractmsg(req)$.\texttt{session\_idx} \equiv x$.
If p receives req in Q_α, there does not exist any $y \in \mathcal{N}$ such that Equation B.72 holds.

$$\mathsf{matchesAssertion}(req, S_\beta(p).\texttt{assertions}[y]) \tag{B.72}$$

Proof. If p receives req in Q_α, it stores req inside the `logoutresponses` subterm of its state. Any subsequent Assertion that matches req is rejected because of Algorithm B.3, Line 19. □

There is a functional dependency between the IdP session alias and the user identity, which is captured by Lemma 18

Lemma 18. Let Q_α be a processing step. Let $i \in \mathsf{IDP}$ be an IdP that is honest in Q_α. Let $req_1, req_2 \in \mathsf{SignedLogoutRequest}_{\mathsf{signkey}^{\mathrm{idP}}(i)}$. If Equation (B.73) holds, then Equation (B.74) holds as well.

$$\text{extractmsg}(req_1).\texttt{session\_idx} \equiv \text{extractmsg}(req_2).\texttt{session\_idx} \tag{B.73}$$

$$\text{extractmsg}(req_1).\texttt{subject} \equiv \text{extractmsg}(req_2).\texttt{subject} \tag{B.74}$$

Proof. As was seen previously, the only process that is able to generate req_1 and req_2 is i. Since they result from the same IdP session, the `subject` is also the same for both Requests, which can be seen from Algorithm B.4, Line 44 ff.: If an existing session is used to authenticate a user, the existing Assertion is reused. Therefore there cannot exist multiple subjects per IdP session. □

Lemma 19. Let Q_α be a processing step. Let $p \in \mathsf{SP}$ be a SP that is honest in Q_α. Let $n \in \mathcal{N}$. Then all of the following statements are true:

1. $n \in S_\alpha(p).\texttt{assertions} \Leftrightarrow n \in S_\alpha(p).\texttt{sessions}$
2. $S_\alpha(p).\texttt{assertions}[n].\texttt{issuer} \equiv S_\alpha(p).\texttt{sessions}[n].\texttt{issuer}$
3. $S_\alpha(p).\texttt{assertions}[n].\texttt{subject} \equiv S_\alpha(p).\texttt{sessions}[n].\texttt{subject}$

Proof. The subterms `assertions` and `sessions` of p's state are only changed in Algorithm B.3, Lines 21 f. and Algorithm B.1, Lines 59 f. In the first location it is easy to see that the lemma holds true after the two lines, if it was true before them (Note that for the inital state the lemma trivially holds true). In the second location, the entries contained in *terminate* and *terminateAssertions* are removed from `sessions` and `assertions` respectively. Since entries are only inserted pairwise into *terminate* and *terminateAssertions* (c.f. Algorithm B.1, Lines 55), statement 1, remains true after removal. □

Lemma 20. Let Q_β be a processing step. Let $i \in \mathsf{IDP}$ be an Identity Provider that is honest in Q_β. Let $p \in \mathsf{SP}$ be a Service Provider that is honest in Q_β. If in Q_β, i received a Logout Response $resp \in \mathsf{SignedLogoutResponse}_{\mathsf{signkey}^{\mathrm{SP}}(p)}$, then there exists a processing step Q_α with $\alpha < \beta$ and a term x such that in Q_α, p received a Logout Request $req := \mathsf{associatedLogoutRequest}(Q_\beta, resp)$ with $\mathrm{extractmsg}(req).\mathtt{session\_idx} \equiv x$ and $\mathrm{extractmsg}(req).\mathtt{issuer} \equiv \mathsf{entityID}(i)$. Furthermore, the predicate $\mathsf{authenticatedWith}(Q_\beta, x, i)$ is false.

Proof. Since p is honest in Q_β and honest SPs never leak their signing key, p is the only process that is able to generate *resp*. For p, *resp* can only be generated in Algorithm B.1, Line 66. For this to happen, p must have received a Logout Request req' with $\mathrm{extractmsg}(req).\mathtt{id} \equiv \mathrm{extractmsg}(resp).\mathtt{inresponseto}$. From Lemma 16, one can see that $req' \equiv req$. Furthermore, Lines 52 ff. were reached too. The term *terminate* contains all subterms *session* of $S_\alpha(p).\mathtt{sessions}$ where Equations (B.75) through (B.77) hold. Let $sid := session.1$ and let $assertion := S_\alpha(p).\mathtt{assertions}[session.1]$

$$session.2.\mathtt{subject} \equiv \mathrm{extractmsg}(req).\mathtt{subject} \tag{B.75}$$

$$session.2.\mathtt{issuer} \equiv \mathsf{entityID}(i) \tag{B.76}$$

$$assertion.\mathtt{authn\_statement}.\mathtt{session\_idx} \equiv x \tag{B.77}$$

Each of these subterms is then removed from $S_\alpha(p).\mathtt{sessions}$. Furthermore each entry with a key of *sid* is removed from $S_\alpha(p).\mathtt{assertions}$.
For the predicate $\mathsf{authenticatedWith}(Q_\beta, x, i)$ to be false, there must not exists a nonce n such that Equations (B.78) and (B.79) hold. Let $assertions := S_\beta(p).\mathtt{assertions}$

$$assertions[n].\mathtt{authn\_statement}.\mathtt{session\_idx} \equiv x \tag{B.78}$$

$$assertions[n].\mathtt{issuer} \equiv \mathsf{entityID}(i) \tag{B.79}$$

If there does not exist a processing step Q_γ with $\gamma \leq \beta$, for which $\mathsf{authenticatedWith}(Q_\gamma, x, i)$ is true, it would also not be true for Q_β. So assume there is such a processing step. Then there are two possibilities:

$\gamma \leq \alpha$: This means that in Q_α there does exist a nonce n for which Equations (B.78) and (B.79) hold. In Q_γ, p receives *req* which means that in this processing step, all subterms contained within *terminate* are removed. Equation (B.76) is fulfilled due to Lemma 19 and (B.77) is trivially fulfilled. Due to Lemma 18 and Lemma 19, Equation (B.75) is fulfilled as well. Therefore, the entries with a key of n are removed from the `sessions` and *assertions* subterms and $\mathsf{authenticatedWith}(Q_\beta, x, i)$ is false.

$\gamma > \alpha$: When p receives *req* in Q_α, *req* is stored inside the `logoutrequests` subterm of p's state. In Algorithm B.3, Lines 13 ff., p checks if the presented Assertion matches *req*. Since p only creates a service session in response to a valid Assertion, there exists a processing step Q_δ for which $\mathsf{authenticatedWith}(Q_\delta, x, i)$ is true and $\mathsf{authenticatedWith}(Q_{\delta-1}, x, i)$ is false. Let $assertion'$ be the Assertion that p received

in this Request. From Algorithm B.3, Lines 21 f., one can see that Equations (B.80) and (B.81) hold. Furthermore, due to Lemma 18, Equation (B.82) holds.

$$\mathit{assertion}'.\texttt{authn\_statement}.\texttt{session\_idx} \equiv x \tag{B.80}$$

$$\mathit{assertion}'.\texttt{issuer} \equiv \mathsf{entityID}(i) \tag{B.81}$$

$$\mathit{assertion}'.\texttt{subject} \equiv \text{extractmsg}(\mathit{req}).\texttt{subject} \tag{B.82}$$

However, this means that the Assertion would be rejected in Lines 13 ff., which contradicts $\mathsf{authenticatedWith}(Q_\delta, x, i)$.

□

Lemma 21. Let Q_α be a processing step. Let $i \in \mathsf{IDP}$ be an IdP that is honest in Q_α. Let $\mathit{sid} \in \mathcal{N}$ be a nonce. Let $\mathit{responses} := S_\alpha(i).\texttt{shib\_sessions}[\mathit{sid}]$. Let $\mathit{assertion}_j :=$ extractmsg($\mathit{responses}.j.\texttt{assertion}$) Let *resp* be a HTTPS Response with $\mathit{resp}.\texttt{body.1} \equiv \texttt{script\_slo}$, which is encrypted with the TLS key k. Let $\mathit{scriptstate} := \text{dec}_\text{s}(\mathit{resp}, k).\texttt{body.2}$ be the scriptstate of the delivered script in Q_α. Let $\mathit{path} :=$ `/profile/PropagateLogout`.
If in Q_α, i has sent out *resp*, Equations (B.83) through (B.85) holds for all $j \in \mathbb{N}$.

$$\mathit{scriptstate}.\texttt{serviceNames}.j \equiv \mathit{assertion}_j.\texttt{conditions.audience} \tag{B.83}$$

$$\mathit{scriptstate}.\texttt{services}.j \in \{\langle \texttt{URL}, \texttt{S}, \mathsf{dom}(i), \mathit{path}, \langle \texttt{key}, j\rangle\rangle, \bot\} \tag{B.84}$$

$$\mathit{scriptstate}.\texttt{serviceNames}.j \equiv \bot \Leftrightarrow \mathit{scriptstate}.\texttt{services}.j \equiv \bot \tag{B.85}$$

Proof. In the following, it will be shown that the three equations hold in Q_α. The script *script_slo* is only sent out in Algorithm B.4, Line 88. There, the *scriptstate* is populated with the *services* and *serviceNames* terms. These terms are in turn populated by the loop that precedes the Lines mentioned before. From Line 80, one can see that Equation (B.84) holds. By observing the definition of $\mathit{assertion}_j$, one can see that $\mathit{assertion}_j$ is equivalent to the value of the *assertion* variable in the j-th iteration of the loop. From this, one can easily see that Equation (B.83) holds. Equation (B.85) holds because there are no $\bot$ terms stored and both sequences, `services` and `serviceNames`, have the same length.
□

Lemma 22. Let Q_β be a processing step. Let $p \in \mathsf{SP}$ be a Service Provider that is honest in Q_β. Let $i \in \mathsf{IDP}$ be an Identity Provider that is honest in Q_β. Let $b \in \mathsf{B}$ be a browser that is honest in Q_β. If $\mathsf{positiveResult}(Q_\beta, b, i, p)$ holds, there must exist a processing step Q_α with $\alpha \leq \beta$ for which the following holds true: In Q_α, i received an HTTP Request containing a Logout Response $\mathit{resp} \in \mathsf{SignedLogoutResponse}_{\mathsf{signkey}^{\text{SP}}(p)}$ for which Equation (B.86) holds.

$$\text{extractmsg}(\mathit{resp}).\texttt{issuer} \equiv \mathsf{entityID}(p) \tag{B.86}$$

Proof. By plugging in Definition 25, one can see that b has loaded *script_slo_result* in an IFrame within a document loaded on a secure origin of i. The only script that is delivered on a secure origin of i and that creates IFrames is *script_slo*. Let *state_summary* denote

the scriptstate of *script_slo* in Q_β. Let *nonce* denote the window nonce of the window in which *script_slo* is loaded. Then, $\mathsf{iframeLabel}(nonce, \mathsf{entityID}(p))$ holds (per Definition 25). From Algorithm B.9 and Definition 24, one can see that there must exist a $j \in \mathbb{N}$ and a processing step Q_γ with $\gamma < \alpha$ such that Equations (B.87) and (B.88) hold. Let s_γ denote the scriptstate of *script_slo* in Q_γ and let $s_{\gamma+1}$ denote the scriptstate of *script_slo* in $Q_{\gamma+1}$.

$$\mathsf{entityID}(p) \in^{\langle\rangle} s_\gamma\,\texttt{.serviceNames} \tag{B.87}$$

$$\mathsf{entityID}(p) \notin^{\langle\rangle} s_{\gamma+1}\,\texttt{.serviceNames} \tag{B.88}$$

Let Q_δ ($\delta \leq \gamma$) denote the processing step in which i sent out *script_slo*. Let s_δ denote the scriptstate of *script_slo* in Q_δ. From Lemma 21 and Equation (B.87), one can see that in Q_δ, there must exist a $k \in \mathbb{N}$ such that Equation (B.89) holds. Let $assertion_k$ be defined as in Lemma 21.

$$s_\delta\,\texttt{.serviceNames.}k \equiv \mathsf{entityID}(p) \equiv assertion_k\,\texttt{.conditions.audience} \tag{B.89}$$

And due to Equations (B.84) and (B.85), Equation (B.90) holds. Let *path* be defined as in Lemma 21.

$$s_\delta\,\texttt{.services.}k \equiv \langle \texttt{URL}, \texttt{S}, \mathsf{dom}(i), path, \langle \texttt{key}, k\rangle\rangle \tag{B.90}$$

Every time, *script_slo* is run, the first elements of the `services` and `serviceNames` subterms are removed and a corresponding IFrame is created. In Q_γ, the IFrame that is labeled with $\mathsf{entityID}(p)$ gets created. Due to Equation (B.90), one can see that it points to the URL $\langle \texttt{URL}, \texttt{S}, \mathsf{dom}(i), path, \langle \texttt{key}, k\rangle\rangle$ b then proceeds to create an HTTPS Request for the IFrame to send it to i. Let Q_δ denote the processing step in which i receives this Request (Note that the Request must reach i, since the IFrame has to load *script_slo_result* in order for $\mathsf{positiveResult}(Q_\beta, b, i, p)$ to be true). When i receives the Request, Algorithm B.4, Lines 90 ff. are reached. An HTTP 303 Redirect Response is sent. Let *dest* denote the value of the `Location` header of the Response.
First, the corresponding Assertion $assertion^\delta$ is looked up, by reading the k-th Assertion from $S_\delta(i)\,\texttt{.shib\_sessions}[sid]$. Since the list of SAML Responses per IdP sessions is only changed by storing $\bot$-terms or appending new elements, Equation (B.91) holds.

$$assertion^\delta \in \{assertion_k, \bot\} \tag{B.91}$$

Next, the EntityID of the Service Provider p' is determined by examining $assertion^\delta$ `.conditions.audience`. From Equation (B.83), one can see that $p' = p$. *dest* is set by looking up the logout service endpoint of p. The service endpoints are stored within the `logoutServices` subterm of i's state. This subterm is never changed, so by using the definition of the initial state of i, one can derive Equations (B.92) through (B.94).

$$dest\,\texttt{.host} \equiv \mathsf{dom}(p) \tag{B.92}$$

$$dest\,\texttt{.path} \equiv \texttt{/Shibboleth.sso/SLO/Redirect} \tag{B.93}$$

$$dest\,\texttt{.protocol} \equiv \texttt{S} \tag{B.94}$$

Furthermore, a Logout Request *slo_req* is stored within the `SAMLRequest` parameter of *dest*. From Line 94, one can see that Equation (B.95) holds.

$$slo\_req\,\texttt{.issuer} \equiv \mathsf{entityID}(i) \tag{B.95}$$

After receiving the HTTP 303 Redirect Response, the Iframe points to a secure origin of p. Therefore, p receives an HTTPS Request in a processing step Q_ζ ($\epsilon \leq \zeta \leq \beta$). This Request is processed in Algorithm B.1, Lines 43 ff. p answers the Request with another HTTPS 303 Redirect Response. Let $dest'$ denote the value of the `Location` header of the Response.
From Lines 67, and Equation (B.95), one can see that Equation (B.96) holds.

$$dest' \equiv S_\zeta(p).\texttt{logoutServices}[\mathsf{entityID}] \tag{B.96}$$

The `logoutServices` subterm is never changed, so from the definition of the initial scriptstate of p, one can derive Equation (B.97) through (B.99).

$$dest'.\texttt{host} \equiv \mathsf{dom}(i) \tag{B.97}$$
$$dest'.\texttt{path} \equiv \texttt{/profile/SAML2/Redirect/SLO} \tag{B.98}$$
$$dest'.\texttt{protocol} \equiv \texttt{S} \tag{B.99}$$

Furthermore, a Logout Response *slo_resp* is stored within the `SAMLResponse` parameter of $dest'$. From Line 66, one can see that Equation (B.100) holds.

$$slo\_resp.\texttt{issuer} \equiv \mathsf{entityID}(p) \tag{B.100}$$

After receiving the HTTP 303 Redirect Response, the Iframe points to a secure origin of i. Therefore, i receives an HTTPS Request in a processing step Q_β for which Equation (B.86) holds. □

Lemma 23. Let Q_α, Q_β be processing steps with $\alpha \leq \beta$. Let i be an IdP that is honest in Q_β. Let x be a term. Let p be a Service Provider that is honest in Q_β. Let b be a browser that is honest in Q_α. If $\mathsf{logoutSummary}(Q_\beta, b, i)$ and $\mathsf{authenticatedWith}(Q_\beta, p, i, x)$ and $\mathsf{secondaryLoggedIn}(Q_\alpha, b, x, i)$ holds, then either $\mathsf{positiveResult}(Q_\beta, b, i, p)$ or $\mathsf{negativeResult}(Q_\beta, b, i, p)$ is true.

Proof. For showing this lemma, one has to show that in Q_β, b has a document loaded on an origin of i which contains an IFrame with a window reference of id and $\mathsf{iframeLabel}(id, \mathsf{entityID}(p))$ holds true. Since $\mathsf{logoutSummary}(Q_\beta, b, i)$ holds, there is a script loaded in b that can potentially create IFrames (namely *script_slo*). Let Q_γ denote the processing step in which i has sent out this script. Let $scriptstate^0$ denote the scriptstate of the script in Q_γ. If there exists a $j \in \mathbb{N}$ such that $scriptstate^0.\texttt{serviceNames}.j \equiv p$, then there exists a processing step in which an IFrame of the form initially mentioned was created. So it is left to prove that there exists such a $j \in \mathbb{N}$.
From Lemma 20, one can see that i has never received a Logout Response signed by p which could terminate the session referenced by x. By Lemma 15, there therefore exists a $j' \in \mathbb{N}$ such that Equation (B.62) holds (with *responses* being defined as in Lemma 15). By observing how $scriptstate^0$ is generated in Algorithm B.4, Lines 79 ff., one can see that $j' = j$. □

In the following, the SLO Consistency property will be proven.
Let ρ be a run of $\mathcal{SHIB}$. Let Q_β be a processing step in ρ. Let i be an IdP that is honest in Q_β. Let b be a browser that is honest in Q_β. Assume that $\mathsf{logoutSummary}(Q_\beta, b, i)$ holds. Assume that there exists a processing step Q'_α and a term x such that $\mathsf{secondaryLoggedIn}(Q_\alpha, b, x, i)$ holds. Then it will first be shown that exactly one of the following statements is true for all Service Providers that are honest in Q_β.

1. There does not exist a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$.
2. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{negativeResult}(Q_\beta, b, i, p)$.
3. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{positiveResult}(Q_\beta, b, i, p)$ and not $\mathsf{authenticatedWith}(Q_\beta, p, i, x)$.

By observing Definitions 3 and 4, one can see that the statements are mutually exclusive, thus it is sufficient to prove that at least one of them is true. This is done by a proof via contradiction. Assume all of the previous statements are false. Therefore, all of the following statements are true:

1. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$.
2. $\mathsf{negativeResult}(Q_\beta, b, i, p)$ is false or there exists no processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$.
3. $\mathsf{positiveResult}(Q_\beta, b, i, p)$ is false or $\mathsf{authenticatedWith}(Q_\beta, p, i, x)$ or there exists no processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$.

This can be achieved in the following two ways:

1. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{negativeResult}(Q_\beta, b, i, p)$ is false and $\mathsf{positiveResult}(Q_\beta, b, i, p)$ is false.
2. There exists a processing step Q_α such that $\mathsf{authenticatedWith}(Q_\alpha, p, i, x)$ and $\mathsf{negativeResult}(Q_\beta, b, i, p)$ is false and $\mathsf{authenticatedWith}(Q_\beta, p, i, x)$ holds.

The first statement directly contradicts Lemma 23. Therefore, assume that statement 2 is true. By combining statement 2 with Lemma 23, one can derive that $\mathsf{positiveResult}(Q_\beta, b, i, p)$ holds. By Lemma 22, there exists a processing step Q_γ ($\gamma \leq \beta$) in which i received an HTTP Request containing a Logout Response such that Equation (B.86) holds. From Lemma 20, one can see that $\mathsf{authenticatedWith}(Q_\beta, p, i, x)$ is false. Now it is only left to show that $\mathsf{sessionTerminatedIdP}(Q_\beta, x, i)$ holds.
Due to $\mathsf{logoutSummary}(Q_\beta, b, i)$, b has loaded the script *script_slo* from a (secure) origin of i. Let Q_δ denote the processing step in which i delivered *script_slo* ($\delta < \beta$). Let *shib_sid_b* := $\mathsf{sessionAlias}^{-1}(x)$. When i delivers *script_slo* in Q_δ, Algorithm B.4 Line 86 is reached. Therefore *shib_sid_b* $\in^{\langle\rangle}$ $S_\delta(i)$`.invalid_sessions` holds. Since elements from the `invalid_sessions` subterm of i's state are never removed, it further holds that *shib_sid_b* $\in^{\langle\rangle}$ $S_\beta(i)$`.invalid_sessions`. This directly implies $\mathsf{sessionTerminatedIdP}(Q_\beta, x, i)$

■

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