The Grant Negotiation and Authorization Protocol: Attacking, Fixing, and Verifying an Emerging Standard

Florian Helmschmidt
University of Stuttgart
Germany
flori@nhelmschmidt.de

Pedram Hosseyni
University of Stuttgart
Germany
pedram.hosseyni@sec.uni-stuttgart.de

Ralf Küsters
University of Stuttgart
Germany
ralf.kuesters@sec.uni-stuttgart.de

Klaas Pruiksma
University of Stuttgart
Germany
klaas.pruiksma@sec.uni-stuttgart.de

Clara Waldmann
University of Stuttgart
Germany
clara.waldmann@sec.uni-stuttgart.de

Tim Würtele
University of Stuttgart
Germany
tim.wuertele@sec.uni-stuttgart.de

ABSTRACT

The Grant Negotiation and Authorization Protocol (GNAP) is an emerging authorization and authentication protocol which aims to consolidate and unify several use-cases of OAuth 2.0 and many of its common extensions while providing a higher degree of security. OAuth 2.0 is an essential cornerstone of the security of authorization and authentication for the Web, IoT, and beyond, and is used, among others, by many global players, like Google, Facebook, and Microsoft. Because of historically grown limitations and issues of OAuth 2.0 and its various extensions, prominent members of the OAuth community decided to create GNAP, a new and completely resigned authorization and authentication protocol. Given GNAP's advantages over OAuth 2.0 and its support within the OAuth community, GNAP is expected to become at least as important as OAuth 2.0.

In this paper, we present the first formal security analysis of GNAP. We build a detailed formal model of GNAP, based on the Web Infrastructure Model (WIM) of Fett, Küsters, and Schmitz. Based on this model, we provide formal statements of the key security properties of GNAP, namely, authorization, authentication, and session integrity for both authorization and authentication. In the process of trying to prove these properties, we have discovered several attacks on GNAP. We present these attacks as well as modifications to the protocol that prevent them. These modifications have been incorporated into the GNAP specification after discussion with the GNAP working group. We give the first formal security guarantees for GNAP, by proving that GNAP, with our modifications applied, satisfies the mentioned security properties.

GNAP was still an early draft when we started our analysis, but is now on track to be adopted as an IETF standard. Hence, our analysis is just in time to help ensure the security of this important emerging standard.

1 INTRODUCTION

Delegated authorization is a common problem on the Web and beyond. With some service providers holding a large amount of data from their users, there are many cases where a user wants to allow some other service to use some (but not all) of this data. For example, a printing service may allow a user to directly print photos from the user's Google account. Authorization protocols provide a way for a user to grant access to data, via an API provided by the host of the data, without the user having to reveal credentials to the service. In the example, the printing service might send a request to Google asking to access the user's data, and Google would then forward this request to the user for authorization. The user could then authorize the printing service to only get read-access to photos stored at his/her Google account.

The OAuth 2.0 framework [33], developed by the IETF OAuth Working Group (OAuth WG), is an omnipresent standard for delegated authorization: A study by Yang et al. [70] found that about 80% of the Alexa Top 500 websites for the US and China use OAuth 2.0. For example, it is used by Google [32], Dropbox [18], Facebook [50], Github [30], and even by financial institutions to provide third-party services access to initiate transactions [44]. OAuth 2.0 is also used for authorizing IoT devices [15, 31].

A closely related goal is authentication. Here a user wants to use a single account for multiple services (single sign-on), see, e.g., the many sites with "login with" buttons. While OAuth 2.0 is primarily an authorization protocol, the OpenID Connect (OICD) [66] extension by the OpenID Foundation adds support for authentication.

A variety of other extensions of the OAuth 2.0 protocol have been developed to improve its functionality in other ways. Some of these extensions improve the security of certain aspects of the protocol, e.g., by adding protection if specific tokens used in the protocol leak [11, 67] or moving certain messages to direct server-to-server communication [46] (instead of communication through the user's browser). Others, e.g. [61], add support for additional features like management of clients (e.g., the printing service in the above example) at service providers or new flows allowing input-constrained devices to be authorized [15].

Motivated by many limitations and shortcomings of OAuth 2.0 and its extensions [55], both in terms of functionality and security, in 2019, members of the OAuth WG – several of them having worked on OAuth 2.0 and its extensions for many years – started to create a new and completely redesigned protocol, the Grant Negotiation and Authorization Protocol (GNAP) [56].

Unlike OAuth 2.0, GNAP includes several ways that a resource owner can interact with an authorization server to authorize resource access. It also includes the ability for a client to renegotiate an ongoing request, and the ability for a client to participate in the protocol without being registered. In addition to the authorization use case, GNAP aims to also provide authentication.

Many details necessary for a typical use case are underspecified in OAuth 2.0, so extensions are needed to create a usable ecosystem. The high number of (historically grown) extensions of OAuth 2.0 is a problem by itself. There are currently 27 standards and nine active drafts by the OAuth WG,[1] 16 standards and more than 20 drafts from the OpenID Foundation,[2] and even more standards related to OAuth 2.0, like User-Managed Access 2.0 Grant (UMA) [47], which originated outside these two main standardization organizations. This makes it difficult for developers to choose the right set of extensions for their use case, in particular, if the use case needs more security than provided by OAuth 2.0. For example, OAuth 2.0 does not define the necessary details of security tokens—these are instead defined in extensions such as [60] and [7]—, and how a third-party service can register at a service provider (e.g., [62]). Also, some of the OAuth 2.0 extensions overlap in functionality, addressing similar problems in different ways. For example, cross-site request forgery (CSRF) protection can be achieved in multiple ways: using the OAuth 2.0 state parameter, using the OIDC nonce value, or using the PKCE extension [67], to name just a few (see also [45, Section 2.1]). Using multiple extensions in the same deployment leads to increasing complexity, and unintended interactions between the extensions can lead to bugs, potentially including security vulnerabilities. Diametrical to having too many options, some of the original OAuth 2.0 flows are not recommended anymore: the OAuth Security Best Current Practice document [45] discourages the use of the implicit flow and the resource owner password credentials flow, two of the four original OAuth 2.0 flows.

The GNAP project attempts to learn from OAuth 2.0 and its extensions in several ways by creating a monolithic protocol that incorporates concepts of several existing OAuth 2.0 extensions. Furthermore, GNAP aims to provide more flexibility and a higher degree of security than OAuth 2.0 and allows for a uniform base for extensions. While GNAP is not designed to be backwards compatible with OAuth 2.0, and will not immediately replace it, its additional expressive power makes it likely to coexist with and gradually take over from OAuth 2.0.

In the past, several attacks on OAuth 2.0 and its extensions, e.g., OpenID Connect, FAPI 1.0 [64, 65], and PKCE [67], have been found [6, 21, 26, 28, 48, 51]. While GNAP is certainly inspired by OAuth 2.0 and tries to cover many features of various OAuth 2.0 extensions, e.g., the Device Authorization Grant [15] and Pushed Authorization Requests [46], altogether, it is a new and completely redesigned authorization and authentication protocol. Thus, the results from previous analyses of OAuth 2.0 and its extensions do not carry over to GNAP.

In this paper, we provide the first formal security analysis of GNAP, uncovering attacks and eventually proving the protocol secure within a very detailed model. We set out to prove four common security properties, which have also been considered in analyses of OAuth 2.0 and related protocols [21, 26, 28]. While the formal versions of these properties depend on the details of the protocol being studied, they have quite simple intuitions. For instance, one major property, security with respect to authorization, states that an attacker cannot access resources of an honest user. Security with respect to authentication states that an attacker cannot log in as another user. We also consider reverse directions of these: session integrity for authorization states that an attacker cannot force another user to access the attacker's resources (consider, for instance, the attacker trying to spread malware), and session integrity for authentication states that an attacker cannot force another user to be logged in as the attacker (and thereby possibly leaking the user's search history etc. to the attacker).

In our analysis, we build a formal model of GNAP and prove that it satisfies the aforementioned security properties. Our model is based on the Web Infrastructure Model (WIM) [23]. The WIM is a very detailed model of the web infrastructure, probably the most comprehensive such model to date. It includes a detailed model of browsers, servers and attacker processes, and communication between processes using HTTP(S) requests and responses. For example, the browser model supports many Web features, including the execution of (honest and malicious) scripts, a complex window and document structure (including iframes), cookies with their various attributes, different redirect methods, various other headers, and in-browser communication using postMessaging. As illustrated, for example in [21, 26, 28], analysis based on the WIM is effective at finding flaws in web-based protocols and standards.

Our model is a pen-and-paper model and we do manual proofs. While there are frameworks for mechanized security analyses, e.g., Tamarin [49] and ProVerif [9], there is currently no tool that directly supports the WIM, and it is actually challenging to build such a tool due to the limitations of current tools and the many features of the WIM, including many complex recursive data structures and computations (see also Section 5). Creating a tool for mechanizing the WIM, while interesting and relevant future work, is hence out of the scope of this work. Previous analyses carried out based on the WIM have been pen-and-paper as well, and as mentioned, have been successful in uncovering attacks on web-based protocols and standards, and in establishing formal security guarantees for them.

Our Contributions. In this paper, we present the first extensive formal security analysis of GNAP. Our analysis has led to the discovery of several attacks. We propose changes to the specification to fix these issues and formally prove that these changes are sufficient to get strong security guarantees for GNAP. We have reported these issues and fixes to the GNAP working group, which have resulted in several changes to the specifications, following our recommendations. Our work has greatly improved on the security of this emerging protocol, which has the potential to become as widely used as OAuth 2.0 and its extensions. More specifically, our contributions can be summarized as follows:

Formal Model of GNAP. We develop a formal model of parties interacting according to GNAP, closely following the specification. As our model is based on WIM, it covers many details of real world implementations of GNAP. To model GNAP in detail, we also extended the WIM with the ability to use codes for the authorization process and by HTTP Message Signatures [3], which is useful for future analysis efforts and of independent interest.

[1] see https://datatracker.ietf.org/wg/oauth/documents/

[2] see https://openid.net/developers/specs/

Formal Security Properties. Based on our formal model, we precisely define what the standard security properties authorization and authentication as well as session integrity for authorization and authentication mean in the context of GNAP.

Attacks and Fixes. While trying to prove that GNAP satisfies these security properties, we found several attacks. Some of these attacks are similar to known attacks on OAuth 2.0 and its extensions, underlying the importance of formal analysis of complex protocols such as GNAP: even very experienced protocol designers are likely to overlook attacks, even if similar problems have occurred in related protocols (which they might even have co-developed).

We have proposed changes to the specification that prevent these attacks, and have provided these suggestions to the GNAP working group. The suggestions that we present in this paper have been accepted and are included in the current version of the GNAP specification [56].

Proof of Security of GNAP. Finally, we prove that GNAP with the proposed fixes is secure w.r.t. standard security properties. This is the first formal proof for GNAP, covering a wide range of attacks.

Structure of this Paper. We first give a brief overview over GNAP (Section 2). We then briefly and informally describe our security properties and attacker model, before we present our attacks in Section 3 that we have found as a result of our first attempt at formally proving the desired security properties of the GNAP protocol. We also discuss fixes. The detailed formal treatment (of the fixed version) of GNAP is then presented in Section 4, including our model of GNAP, formal security properties, and the main security theorem. Related work is discussed in Section 5. We conclude in Section 6. Full details and proofs can be found in the appendix.

2 GRANT NEGOTIATION AND AUTHORIZATION PROTOCOL

The Grant Negotiation and Authorization Protocol (GNAP) [56, 57] specifies how the owner of some resources can give a piece of software access to their resources and how subject information about the owner can be conveyed to the software without the need for the owner to reveal login credentials to that software. Additionally, if the software lacks the rights to access the resources it initially requests, GNAP provides means for this request to be adjusted, via negotiations between the owner of the information and the receiving software.

In this section, we give a detailed description of flows following the GNAP specification. We begin with an overview of the main concepts, which we explain by means of a concrete example of a GNAP flow.

The GNAP specification consists of two documents: the *core* specification and the *resource server* specification. When we started our analysis, the most recent version of the core specification was version 8 [58] and version 2 for the resource server specification [59]. At the time of writing this paper, the current versions of these documents are 15 [56] and 3 [57], respectively, and the core specification is submitted to IESG for publication [69], indicating that it is in its final stages.

Some of the (security-relevant) changes to the core specification introduced since version 8 result from suggestions we have proposed to the working group based on our analysis. We point those out in Section 3. In the following, we describe the originally analyzed version 8 of the core and version 2 of the resource server specifications. Note however that our final formal model (for which we proved security as explained below) incorporates the security-relevant changes introduced in versions 9-15 of the core specification.

Roles. In GNAP, there are five roles that participants can take. Firstly, there is the *resource owner (RO)* that authorizes access to her protected resources. Together with the *end user (EU)*, who wants to access some protected resource, they describe the two roles of non-software participants (for example natural persons). Further, GNAP defines two server roles. The *authorization server (AS)* delegates authorization of the resource owner by issuing access tokens. The protected resources are handled by a *resource server (RS)* that provides operations on these resources when presented with a valid access token. Finally, there is the *client instance (CI)* which is the central piece of software that communicates with ASs to obtain access tokens and with RSs to obtain access to resources. Two settings are differentiated depending on the operation of the client instance. Either an end user is present that operates the client instance (*end-user case*) or the client instance acts on its own (*software-only case*).

Overview of a GNAP Flow. GNAP allows for a variety of concrete protocol flows by supporting several options for some steps in the flow. For example, interaction with the user and authorization server depends on the capabilities of the client. In Figure 1, we give an overview of the general structure of a GNAP flow, by looking at one specific flow for authorization in the end-user case, where we fix some of those choices; of course our formal analysis also covers the other flows and choices (see also Section 4).

The flow is typically initiated by an EU (Step $\boxed{1}$), who wants to authorize a CI to access resources of an RO. The means by which the EU does so are out of scope of the protocol. The CI then contacts a suitable AS (e.g., pre-configured or specified by the EU) to get authorization to access some resource r (Step $\boxed{2}$). When processing this request, the AS might decide that interaction with the RO is needed to approve this access and informs the CI accordingly (Step $\boxed{3}$). There are multiple ways for the AS to contact the RO, so-called *interaction start modes.* For this illustration, we show the case where the AS contacts the RO directly (Step $\boxed{4}$). The RO then authenticates at the AS and authorizes the access to r for the CI (Step $\boxed{5}$). Once the RO has authorized access to r, there are again multiple options for the AS to notify the CI of the finished interaction, so-called *interaction finish methods.* Figure 1 shows the case in which the AS contacts the CI directly (Step $\boxed{6}$). The CI can now continue the request in Step $\boxed{7}$ and request an access token at the AS. The AS generates an access token AT (Step $\boxed{8}$) and sends it to the CI (Step $\boxed{9}$). AT is used by the CI to get access to r from the RS (Step $\boxed{10}$). When receiving AT, the RS has to verify that the access token is valid, for example, by asking the AS about the validity of the token (this method is called *token introspection*, see Steps $\boxed{11}$ and $\boxed{12}$). If AT is valid for r, the RS provides access to r (Step $\boxed{13}$).

The protocol allows participants to take on several of the five roles at the same time: for example, the AS and RS could be the same server, and often the RO and EU are considered to be the same person, in particular, for the authentication use case: If GNAP is

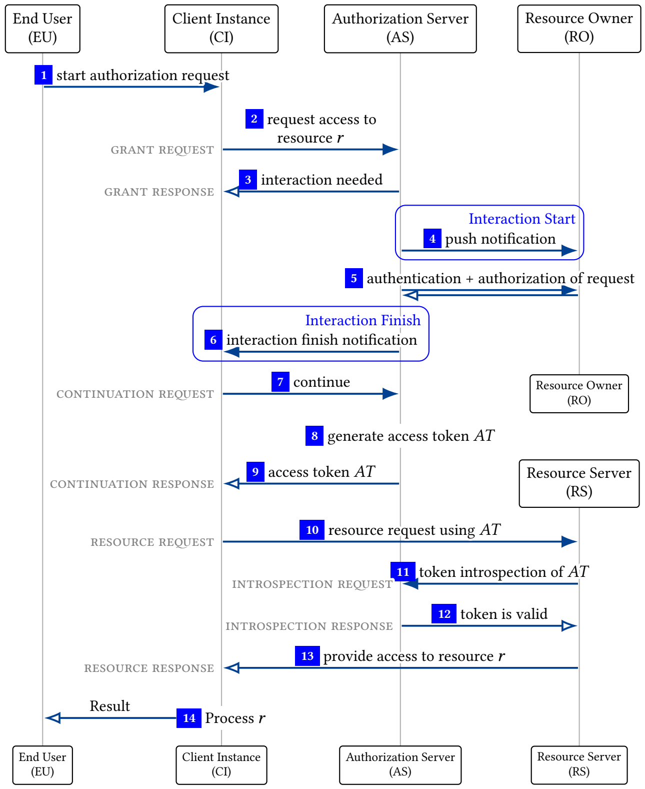


Figure 1: Overview of a GNAP flow

used for authenticating an EU at a CI, then the CI requests subject information on the EU in Step $\boxed{2}$, which the AS returns instead of or in addition to the access token. Once the CI receives this information, the EU is considered to be logged in at the CI.

Detailed Flow. In a bit more detail, a GNAP flow can be grouped into four types of request-response-pairs. A *grant request* from a CI to an AS starts each GNAP flow (Step $\boxed{2}$). In this initial request, the CI specifies the kind of access it requests and gives information to the AS to identify the CI in subsequent requests. The CI may request one or more access tokens to be used to access resources, as well as subject information about the RO. For access tokens, the CI includes details about what kind of token(s) it requests and which rights the token(s) should have. Similarly, for subject information, the CI specifies what kind of information it seeks, e.g., subject identifiers [4], or SAML 2 assertions [12]. Furthermore, the CI has to include the set of supported *interaction start* and may include an *interaction finish method*, i.e., ways in which the CI can facilitate interaction between the RO and AS (descried in more detail below). In addition, the CI has to identify itself to the AS. This identification consists of a *key proof* using the CI's *client instance key*. Each subsequent request from the CI to the AS must also include such a key proof (see "Securing Client Requests").

A grant request is answered by the AS with a *grant response* (Step $\boxed{3}$). If the CI requested access tokens and the AS grants the request, the grant response contains the tokens, optionally together with a reference to the key to which the respective token is bound, if different from the key used by the CI in its grant request. Similarly, the grant response may contain RO subject information if the CI requested such information. If the request cannot immediately be granted (or denied), some interaction with the RO has to happen first (Steps $\boxed{4}$ and $\boxed{5}$). In this case, the grant response contains information on whether the CI can facilitate interaction between the RO and AS, and if so, how (see "Interaction Methods"). The AS may allow the CI to continue the request, e.g., to check whether interaction with the RO has finished, by including a *continuation URI*, as well as a *continuation access token* bound to the key presented by the CI in the grant request.

If the AS allows continuation of the grant, the CI may send a *continuation request* (Step $\boxed{7}$) to the continuation URI specified in the grant response. In a continuation request, the CI includes both the continuation access token that it received in the grant response from the AS and a key proof for the key used by the CI in its initial grant request (see "Securing Client Requests"). The corresponding response from the AS has the same form as a grant response. To clarify that a given message is a response to a continuation request, we sometimes call it a *continuation response*. Continuation requests are used to negotiate ongoing grants, poll whether interaction between RO and AS has finished, and to continue a grant after interaction with RO has finished. There can be a series of continuation requests and responses before the CI and AS reach an agreement.

If the CI received an access token, it can send a *resource request* to the RS (Step $\boxed{10}$) including the token together with a key proof for the key associated with the token (if such a proof is required, see "Presentation of Access Tokens"). When receiving an access token, the RS checks whether it is valid and sufficient for the requested resource, and verifies the key proof (if required for the token). If all checks are successful, the RS returns the resource to the CI in a *resource response* (Step $\boxed{13}$).

To check whether a given access token is valid and sufficient, and to get information in the key to which the token is bound, the RS can either examine the token itself (in case of a structured access token, e.g., [7]), or by token introspection at the issuing AS. In the latter case, the RS sends an *introspection request* with the token to the AS (Step $\boxed{11}$). The AS then checks whether the token is valid and sends an *introspection response* to the RS (Step $\boxed{12}$) including the access rights associated with the token and information about the key and proof method used by the CI (if any). The RS can then check the key proof presented by the CI before providing access to the resource.

Interaction Methods. A central part of many GNAP flows is getting authorization from the RO (Step $\boxed{5}$). To facilitate the necessary interaction between the AS and RO, and to inform CI of completed interaction, GNAP defines several *interaction start* and *interaction finish* methods, marked by the blue boxes in Figure 1, which we describe in the following.

To start interaction with the RO, the AS can contact the RO directly, as shown in Step $\boxed{4}$ of Figure 1. Note that GNAP does not give details on this direct communication between the AS and RO.

If the RO and EU happen to be the same entity (which is often true in practice), the CI can assist the AS in contacting the RO. GNAP defines four additional interaction start methods for this case, the CI has to indicate in its grant request which of those methods it supports, it is up to the AS to select an appropriate one (or resort to out-of-band communication). (1) With the *redirect* method, the AS's grant response contains an URI to which the CI then redirects the RO, e.g., by displaying a QR code, or with a HTTP redirect (see Steps $\boxed{4}$ and $\boxed{5}$ in Figure 2). (2) In the *user code URI* method, the AS includes a short URI and user code in its grant response. The CI then communicates (e.g., by displaying it) this URI and user code to the RO, who visits the URI with a browser and enters the user code. (3) An even simpler variant of (2) is the *user code* method, which works the same, but the URI is static and can thus, e.g., be printed onto a device implementing a CI. The two user code methods are intended for CIs with a limited user interface, e.g., IoT devices. (4) If the CI is able to launch applications, e.g., if CI is a smartphone app, the *application URI* method can be used. For this method, the AS's grant response contains an app URI, which CI uses to launch the associated application.

With all four methods, the URI or user code must uniquely identify the associated grant at the AS.

To notify the CI after interaction between the RO and AS is complete, GNAP specifies two interaction finish methods. In its grant request, the CI indicates which finish method it wants to use (if any), it is up to the AS to decide between using that or no finish method. With the *push* interaction finish method shown in Step $\boxed{6}$ of Figure 1, the CI includes a URI in its grant request. The AS then sends a POST request to that URI to indicate that interaction completed. In the *redirect* finish method, the CI again includes a URI in its grant request, but this time, the AS redirects the RO to that URI. As for the redirect start method, the means by which the AS redirects the RO are not fixed by GNAP.

In both cases, the AS includes a nonce to its grant response and adds an *interaction reference* as a parameter to the finish URIs. The CI then includes the interaction reference in its subsequent continuation request (Step $\boxed{7}$ in Figure 1). In its response, the AS includes an *interaction hash*, covering the nonces by CI and AS, the grant endpoint at which the flow was started, and the interaction reference. This hash is verified by the CI to protect against injection attacks (e.g., CSRF attacks against ROs). Without a finish method, the CI can send continuation requests to the AS, *polling* the current status of the request.

Securing Client Requests. To guarantee that the AS talks to the same CI in a flow, the CI must include a proof of possession of a key in all requests. For the initial request, the CI selects a *client instance key* (which also identifies the CI to the AS) and proof method. The same key and method must then be used for all further requests to the AS in this flow. GNAP supports HTTP Message Signatures [3], Mutual TLS certificates [54], and JSON Web Signatures [42] as proof methods, where both symmetric and asymmetric keys are allowed.

Presentation of Access Tokens. When issuing an access token, the AS has three options to sender-constrain the token: (1) The token can be bound to the *client instance key* of the CI which started the grant. (2) The token can be bound to *a different key*. In this case, the AS includes a hint to the CI in the grant response, so the CI knows which key to use with this token. The means by which such keys are agreed upon between AS and CI are out of scope of GNAP. (3) If requested by the CI, the token can also be a *bearer token*, i.e., not be bound to any key.

With options (1) and (2), the token can only be used at an RS if accompanied by a key proof for the key to which the token is bound. With option (3), the token can be used without a key proof.

3 ATTACKS AND FIXES

In this section, we first briefly and informally describe the security properties GNAP is expected to fulfill, including a short overview of our attacker model. Following this, we present the attacks we found during our formal security analysis of GNAP and the fixes we propose to make GNAP satisfy the security properties. We describe the actual formal security analysis in Section 4.

3.1 Informal Security Properties

As already mentioned in the introduction, GNAP is supposed to fulfill the following properties:

Security w.r.t. Authorization. An attacker should not be able to access resources of an honest RO. This is a natural and one of the main properties every authorization protocol should satisfy.

Session Integrity for Authorization. An honest end user should not unwillingly access resources of the attacker. More specifically, an end user accesses resources of a specific RO only if she explicitly and willingly asked for these resources, given that the responsible AS is honest.

Security w.r.t. Authentication. An attacker should not be able to log in at an honest (i.e., not corrupted) CI under the identity of an honest user, if that identity is governed by an honest AS.

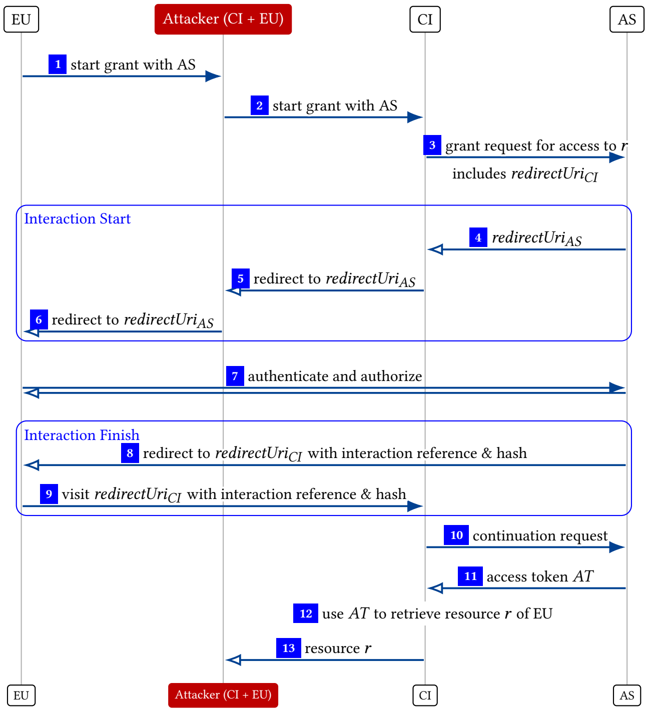
Session Integrity for Authentication. An honest user should only be logged in with a specific identity, if she explicitly and willingly wanted to log in with this identity, given that the responsible AS is honest.

Attacker Model. These security properties should hold in the presence of a network attacker, which can also use Web features. For example, the browser of an honest user may also open websites of the attacker. Such an attacker website can deliver malicious scripts, use postMessage communication within the honest user's browser, or even include websites of honest parties in iframes, and try to exploit these and other Web features to launch attacks; of course, vice versa honest websites can also include malicious websites. Additionally, we assume that the attacker can get hold of access tokens that are sender-constrained. This is an implicit assumption when using token binding mechanisms, as otherwise, there is no need to use token binding. Such a leak of access tokens may happen for a number of reasons, e.g., due to a compromised RS (the token may be valid for multiple RSs), or through unsecured TLS intercepting proxy logs (considered, e.g., in [19]).

We refer to Section 4.4 for more details on the formalized security properties, and to Section 4.1 for more details on the attacker model.

3.2 Client Instance Mix-Up Attack

The client instance mix-up attack enables an attacker to access resources of an honest user, thus breaking the authorization property. This attack has several variants, depending on which interaction start and interaction finish methods are used.



EU
Attacker (CI + EU)
CI
AS
1 start grant with AS
2 start grant with AS
3 grant request for access to r
includes $redirectUri_{CI}$
Interaction Start
4 $redirectUri_{AS}$
5 redirect to $redirectUri_{AS}$
6 redirect to $redirectUri_{AS}$
7 authenticate and authorize
Interaction Finish
8 redirect to $redirectUri_{CI}$ with interaction reference & hash
9 visit $redirectUri_{CI}$ with interaction reference & hash
10 continuation request
11 access token AT
12 use AT to retrieve resource r of EU
13 resource r
EU
Attacker (CI + EU)
CI
AS

Figure 2: Client Instance Mix-Up Attack

Redirect/Redirect Interaction. Figure 2 shows an example flow of the attack using the redirect interaction start method and the redirect interaction finish method.

In the first phase of the flow (up until Step 7), the attacker acts as a man-in-the-middle between an honest EU and an honest CI: an honest EU starts a flow at the attacker's client instance and wants to authorize it to access a resource r managed by an honest AS (Step 1). Instead of starting a grant with the AS, the attacker poses as an EU, starting a flow with the AS at an honest CI to grant CI access to r (Step 2). The honest CI then sends a grant request to the AS (Step 3). Since the redirect interaction *finish* method is used, this grant request includes a URI $redirectUri_{CI}$ to which the EU is redirected by the AS once interaction between AS and EU is completed.

The grant response of AS contains a URI $redirectUri_{AS}$ which is associated with the ongoing grant (Step 4). This second URI is part of the redirect interaction *start* method. CI then instructs its end user (i.e., the attacker), to visit $redirectUri_{AS}$ (Step 5).

The attacker, now again posing as a client instance towards the honest EU, does not visit $redirectUri_{AS}$, but instead instructs the EU to do so (Step 6). The EU complies, authenticates and authorizes the request (Step 7). Note that the EU expected to be asked to authenticate and authorize access to r at that exact AS.

However, from the AS's point of view, $redirectUri_{AS}$ is associated with the grant request sent by the CI in Step 3. Therefore, the AS now invokes the redirect interaction fininsh method and instructs the EU to visit the CI at $redirectUri_{CI}$, adding an interaction reference and interaction hash as URI parameters (Step 8). Once again, the EU complies, thus providing the CI with interaction reference and interaction hash (Step 9). The CI can now request an access token (Step 10). After receiving the access token in Step 11, the CI can access resource r of the honest EU (Step 12). However, from the point of view of the CI, the access token, and thus r, are associated with the session between the CI and the attacker (posing as end user), hence giving the attacker access to r.

Redirect/Push Interaction. Similar problematic flows can occur with other interaction methods as well: If the *redirect* interaction start method is used with the *push* interaction finish method, the attack flow is identical to the one in Figure 2 up to Step 7—except for the grant request, which does not contain a redirect URI. Further, instead of redirecting the EU to the CI (Steps 8 and 9), the AS sends a push notification with the interaction reference and hash directly to the CI. Having received this push notification, CI sends a continuation request and the flow finishes as described above.

User Code/Redirect Interaction. When using the *user code* interaction start method with the *redirect* interaction finish method, the attack starts as before, but instead of including a redirection URI in its grant response (Step 4), AS includes a user code URI and a user code. Similar to the redirection URI shown in Figure 2, these values are passed on to the attacker and from there to the EU. The EU then visits the user code URI, enters the user code, authenticates, and authorizes the grant (similar to Step 7). From there, the attack flow continues as depicted in Figure 2: The EU is redirected to the CI, which sends a continuation request, receives an access token, and ultimately gives the attacker access to the EU's resources.

User Code/Push Interaction. If the *user code* interaction start method is used with the *push* interaction finish method, the attack flow starts as explained in the previous paragraph, up until interaction is finished. From there on, the flow continues with the push interaction finish as described above. This variant is an instance of an illicit consent grant attack for cross-device flows described in [43], conceptually similar attacks have also been described for OAuth 2.0 extensions [15, 20, 63, 71].

Any Start/No Finish Interaction. If no interaction finish method is used, the CI has to poll at the AS with continuation requests until interaction is finished and the AS issues an access token. The interaction start method does not matter in this case, and the attack flows for different start methods are identical to the ones described above, up to and including the EU's authentication and authorization at the AS. While EU authenticates, CI polls the AS with continuation requests until the AS' continuation response contains a token. The remainder of the attack flow is above.

User Code Injection. Another notable variant of this attack works as follows: instead of having a malicious client that an honest EU wants to authorize, the attacker can start a flow at an honest CI with an honest AS, up to the point where the attacker receives a user code. Note that from the point of view of the CI, this user code is associated with a session between the CI and the attacker. The attacker can then try to perform a social engineering attack in which he convinces an honest EU to use the user code at AS and authorize CI. For example, the attacker, claiming to be from the technical support team of a company, can send an email to all employees saying that due of a security breach they need to verify their email accounts at the following QR-code with the user code.

Mitigations. There are conditions under which the described attacks can be prevented: when using the *redirect* start and finish interaction methods with EUs which use a single browser for the whole interaction, the CI can establish a session with the EU's browser, e.g., by setting a session cookie, and verify that the browser initiating the flow is the same browser which is redirected to the CI by the AS. By implementing this fix for flows with browser-EUs using the *redirect* start and finish methods in our fomal model, we were able to prove effectiveness of this fix.

In all other cases, the only reliable way to prevent this class of attacks is for the EU to only authorize CIs which the EU actually wants to authorize (see Step $\boxed{7}$ in Figure 2), which requires the AS to provide enough (reliable) information about the CI, and the EU has to carefully inspect this information. The lack of other fixes is also evident from the fact that the attack variant with user code and push interaction has been described before, but no general protocol-level fix has been proposed so far (see, e.g., [20, Section 5.6.7] and [63]).

In order to prove security of GNAP, we model ASs to present such information and model EUs to carefully compare this information to what they expect for all cases except the one mentioned in the previous paragraph in which a session between CI and EU's browser is sufficient to prevent the attack. We refer to Appendix A.3 for details on the modeling.

We reported these findings, including the fix for the redirect start/redirect finish case, to the GNAP Working Group [34], and helped in developing corresponding security guidance [38], which is now part of the GNAP specification [56, Section 13.22 & 13.23].

3.3 Attack on Authentication

In this attack, the attacker is able to log in at an honest CI under the identiy of an honest end user, if the CI solely relies on subject information provided by the AS to identify a user account, without considering the issuing AS as part of the identifying information. Subject information in GNAP can be either *OpenID Connect ID Tokens*, *SAML 2 assertions*, or *Subject Identifiers for Security Event Tokens*. While not all of these include a form of issuer authentication, we note that this attack also works if subject information contains issuer information, is signed by that issuer, and CIs verify these signatures. Figure 3 shows an example flow.

First, an honest EU uses an identity u, which is governed by an honest AS, to log in at an attacker CI (Steps $\boxed{1}$ – $\boxed{8}$): after EU authenticated, AS sends subject information for u to the attacker CI (Step $\boxed{8}$). Note that this is not yet an attack: after all, the EU authorized release of subject information to the attacker CI.

However, the attacker can now use that subject information to log in at an honest CI using an AS controlled by the attacker (AAS): posing as an end user, the attacker initiates a (login) flow at the honest CI, selecting AAS (Step $\boxed{9}$). The honest CI then starts a grant with AAS, requesting subject information (Step $\boxed{10}$). AAS can now inject the subject information which the attacker received in the first phase of the attack (Step $\boxed{11}$). The honest CI will subsequently consider the attacker to be logged in as u, i.e., under the identity of an honest user, hence breaking the authentication property.

The underlying cause is that the honest CI in the attack identifies its users by subject information only, regardless of the issuer of that information. We note that even without an attacker, this behavior can lead to problems when multiple (honest) ASs happen to use the same subject identifier for different users (e.g., incremental database record numbers). These issues can be fixed by mandating CIs to identify their users by the pair (AS, subject information). We reported the issue to the GNAP editors [13] and they added our fix to the specification [56, Section 3.4].

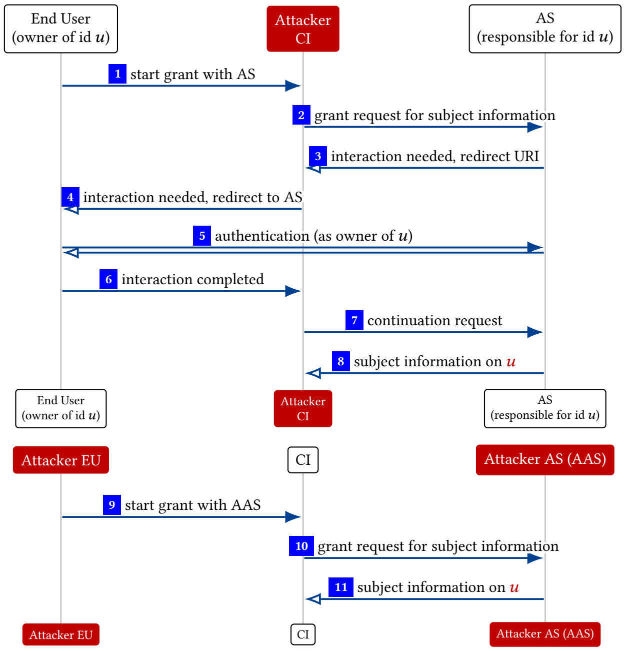


Figure 3: Authentication Attack

3.4 Stolen Token Replay Attack

Recall that our attacker model allows for key-bound access tokens to leak to the attacker (see Section 3.1). In the stolen token replay attack, the attacker can use such a leaked access token that is bound to a key of an honest CI to access resources of an honest RO, if the CI uses the same key and proof method for several ASs (one of which is controlled by the attacker). Note that CIs may support a wide range of ASs, in particular in light of GNAP's *RS-first Method of AS Discovery* [56, Section 9.1], in which CIs can dynamically discover and connect to ASs. Hence, a scenario in which an honest CI interacts with honest as well as malicious ASs is realistic, even without considering otherwise honest ASs being compromised. This attack is similar to the *Cuckoo's Token Attack* described for the OpenID Financial-grade API in [21]. Figure 4 shows an example flow of the attack for GNAP.

We assume that the attacker obtained an access token *AT* (Step $\boxed{2}$) which has been issued by an honest AS (HAS) for an honest CI to access a resource r of an honest RO, and that *AT* is bound to CI's key k (Step $\boxed{1}$). Note that the attacker cannot use *AT* directly to access resource r: when presenting *AT* to the RS, one also has to provide a key proof for key k – which the attacker cannot produce. However, the attacker can start a flow at the CI as an end user, specifying an AS controlled by the attacker (AAS), see Step $\boxed{3}$. We assume that CI uses the same key k and proof method for both HAS

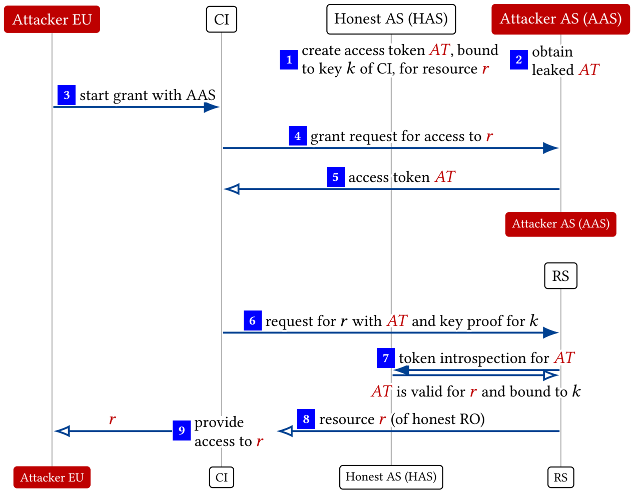

Attacker EU
CI
Honest AS (HAS)
Attacker AS (AAS)
1 create access token AT, bound to key k of CI, for resource r
2 obtain leaked AT
3 start grant with AAS
4 grant request for access to r
5 access token AT
Attacker AS (AAS)
RS
6 request for r with AT and key proof for k
7 token introspection for AT
AT is valid for r and bound to k
8 resource r (of honest RO)
r
9 provide access to r
Attacker EU
CI
Honest AS (HAS)
RS

Figure 4: Stolen Token Replay Attack on GNAP

and AAS (e.g., by using a single client instance key for all ASs). Instead of creating a fresh access token, AAS now responds with the leaked *AT* (Step $\boxed{5}$). Now, CI can use *AT* at the honest RS to access *r* (Step $\boxed{6}$), thereby giving the attacker access to *r* (Step $\boxed{9}$). Hence, this attack breaks the authorization property. This attack is particularly interesting, as in GNAP, an AS can specify which key CI should use to produce key proofs for a token, i.e., even if a CI would use different keys for different ASs, the attacker may still be able to force the CI to use a key associated with an honest AS.

A straightforward fix for this attack is to mandate CIs to use *different* keys for all ASs they talk to: in the attack flow above, CI would then use k_{AAS} when presenting *AT* to the RS (Step $\boxed{6}$), while *AT* would in fact be bound to k_{HAS}, i.e., the RS's token binding check would fail, preventing the attack. This fix was added to the GNAP specification as a security consideration [56, Section 13.30].

3.5 307 Redirect Attack

A central goal of GNAP is security w.r.t. authorization. In particular, GNAP should authorize CIs without revealing RO's credentials at ASs to CIs [58, Section 12.9].

We discovered that such leakage of RO credentials is possible if the redirect finish method is used. If the attacker controls CI to which RO credentials leak, it can then impersonate RO at AS. The attack is similar to an attack on OAuth 2.0 [26] and works as follows. During interaction, RO logs in at AS (see Step $\boxed{5}$ of Figure 1). When using a Web browser, the login typically happens with the credentials being sent in a POST request to AS. After that, AS redirects RO back to CI (redirect finish method, see Section 2).

When we started our analysis, GNAP did not restrict the HTTP redirect method to be used by AS when redirecting RO back to CI. However, if AS uses the 307 redirect status code, the browser forwards the original POST request (sent by RO to AS) to CI, including the request body with RO's credentials; we note that the WIM faithfully models various redirect methods including 307 redirects.

To avoid this, AS should instead use the 303 redirect code, which rewrites the request to a GET request, dropping the request body. We proposed a corresponding specification change to the GNAP editors [37], which is now included in the GNAP specification.

4 FORMAL ANALYSIS

In this section, we present our formal security analysis of GNAP. We first provide a brief overview of the Web Infrastructure Model (WIM), which serves as a basis for our model of GNAP. We then describe our GNAP model in Section 4.2, which follows the latest version of the in-progress GNAP specifications for both the core protocol [56] and for resource server behavior [57]. As already noted, the core protocol has reached the Working Group Last Call, and hence is considered to be stable. Our model includes the fixes presented in Section 3, which, as mentioned, are now also included in the core GNAP specification. We then present formal definitions of the security properties sketched before (Section 4.4), followed by our main theorem, stating that GNAP is secure with respect to these properties. Full details of the security properties and proofs, including formal definitions of clients, authorization servers, and resource servers, are provided in the appendices of this paper.

4.1 WIM

Our model builds on the Web Infrastructure Model (WIM) of Fett, Küsters, and Schmitz, which was introduced in [23], and has since been extended and improved in later work [16, 21, 24–26, 28].Our analysis is based on a consolidated version of the WIM [29]. We here only give a brief description of the WIM, sufficient to follow the paper, but all details can be found in [29]. We also note that in order to model GNAP, we have extended the WIM slightly to model the user-code interaction modes of GNAP, and by adding HTTP Message Signatures [3], including the Signature-Input and Signature HTTP headers (see Appendices A.1 and A.10). These extensions to the WIM are of independent interest, as they can be used for the analysis of other protocols.

The WIM is a formal model of the web infrastructure, designed to be general-purpose and allow for modelling various (web-based) applications. It closely follows published standards and specifications for the web, such as the HTTP/1.1 standard, for example. It provides a general communication model, which models, among other things, HTTP(S) requests and responses, including a variety of headers, such as Origin, Referer, Location, STS, Authorization, and Cookie headers. On top of this communication model, the WIM defines models for several types of processes, including web browsers, web servers, and DNS servers, as well as several forms of attacker processes. For example, the browser model covers the concepts of windows, documents, and iframes. It also has an abstract model of executable JavaScript, which can be sent between processes and executed by browsers, with access to a browser API, e.g., for postMessages, session and local storage, setting and reading headers, XMLHttpRequests, navigating and creating windows/iframes. Users interacting with a browser, e.g., clicking on a link or entering URLs, are modeled as non-deterministic actions of the browser that can be triggered by the attacker. In particular, this means that within our model, the GNAP end-user is subsumed by the browser.

In the WIM, a *web system* is a collection of (atomic) processes, each of which represents a browser, server, or attacker. Each process has a collection of (IP) addresses on which it listens for messages, and processes communicate via events, which consist of a message along with addresses indicating the sender and intended receiver of the message (though attacker processes may tamper with this data, for instance to spoof messages from another process). In every step of execution of the model, an event is selected non-deterministically from the set of waiting events, and delivered to one of the processes that listens to the event's receiver address. This process can then act based on the contents of the event, possibly changing its internal state and/or outputting some number of new events, which are then added to the set of waiting events.

The WIM follows the Dolev-Yao approach [17], meaning that messages are expressed as formal terms over a signature which contains function symbols of various arity, representing, for instance, addresses, strings, nonces, tupling and projection operations, encryption and decryption, and signature creation and checking.

A *Dolev-Yao process* is a process that consists of a set of (IP) addresses the process listens on, a set of terms representing the possible states of the process, an initial state from that set, and a relation that takes an event and a state as input and produces a new state and a sequence of events as output. Note that this is a relation rather than a function because it may be non-deterministic, with the same message and initial state leading to multiple different outputs. This relation models a computation step of the process, and so it must be possible to compute the output of the relation (in a formal sense) from the input event and state.

An *attacker process* in the WIM is a Dolev-Yao process that records all messages it receives and outputs all sequences of events it can possibly derive from its recorded messages. As such, an attacker process can carry out any attack that any Dolev-Yao process could possibly perform, without the details of the attack needing to be specified in the model. There are two types of attackers. A web attacker participates in the network as any other process, and hence only receives messages sent to it directly. In contrast to that, the network attacker can intercept and redirect any messages, i.e., listens to all IP addressses, and hence, sees all messages sent over the network. But it cannot decrypt messages unless it has the corresponding key. This network attacker, despite being a single process, subsumes all other forms of attacker, as it can emulate any more limited attacker or family of limited attacker processes. Attacker processes may also corrupt other processes, both browsers and servers, turning them into web attacker processes (retaining whatever state they had at the point of corruption, including secrets like passwords or keys).

Scripts in the model represent JavaScript running in a browser, and are defined similarly to processes, but rather than being able to directly interact with the network by sending events, they only interact with the browser they are run in. Similar to an attacker process, an attacker script can perform any action that any script could perform within a browser.

Now, a bit more formally, a *web system* $\mathcal{W}$ is then defined as a set of Dolev-Yao processes, along with a set of scripts (those that honest servers would send to browsers), plus the attacker script. In a run of a web system, its state at any given time is a *configuration* of the form (S, E, N) where S maps each process of the system to its current state, E is the set of waiting events, and N is a set of nonces which have not yet been used (and so are unknown to all processes). By convention, we will write s_p^0 for the initial state of process p. A web system induces a set of *runs*, or sequences of configurations, where each configuration is derived by delivering one of the waiting events of the previous configuration to a process, which then performs a computation step, potentially consuming some nonces from N, updating its own state (changing S), and/or adding some number of events to E.

4.2 Modeling GNAP

Since the WIM already formalizes the web infrastructure, to model applications we only need to specify application-specific processes, including scripts they use. Our model of GNAP therefore provides definitions of processes representing ASs, RSs, and CIs. Browsers are already specified in the core WIM – we only need to make some small modifications in order to model the user code interaction mode of GNAP and HTTP Message Signature. End-users are modelled by non-deterministic behaviour of browsers.

A *GNAP web system*, denoted by $\mathcal{GWS}$, is a web system containing some arbitrary (but finite) number of processes including ASs, RSs, CIs, and browsers along with a network attacker process. Other than the attacker, all processes are initially honest (that is, they follow their given algorithms to take steps), but can become corrupted by the attacker at any time.

CIs and ASs are modeled in a straightforward way according to the specification of GNAP [56]. Our model includes a client script that initiates the protocol, and an AS script for authenticating the resource owner. For RSs we follow GNAP's resource server specification [57] for the pieces that are specified, and attempt to make minimal assumptions about RS behaviour otherwise in order to cover a wide variety of use cases.

As mentioned, the formal model presented in this section reflects the security considerations presented in the GNAP specification to avoid known attacks, particularly the fixes described in Section 3, which are necessary for our security properties to be provable.

In a real-world GNAP environment, participants may have diverse setups. For example, each CI has its own set of supported interaction methods. To account for all possible combination of setups, we include these setups in two ways. Some parts are specified in the initial configuration of each process, e.g., the private keys of a process. Other parts are modeled by non-deterministic choices during the execution of processes. For example, for each new grant request, a CI chooses non-deterministically which interaction start and finish method to offer to the AS from the set of all available methods. We refer to Appendix A for our full formal model.

4.3 Modeling Considerations

There are several places in our model where we deviate from the GNAP specification. In some places we use a simplification and consider only a safe over-approximation. In others, we omit some pieces of the specification that are not security-relevant or under-specified. Additionally, our model covers relevant security considerations recommended in the specification and all of our fixes to the attacks as described in Section 3. We will briefly discuss the key constraints and choices for our model (a full list can be found in Appendix A.3).

Simplifications. GNAP does not specify any particular resource access model, and so we use a simple model in which an access token issued for a given resource owner grants access to all of that owner's resources. This allows our model to avoid details of token management such as requests for multiple tokens, or requests to extend the rights of a token, while still being a safe over-approximation of real behavior. Since we use this simple resource access model, the resource owner does not need to be informed during interaction with the AS what access permissions it is being asked to authorize, and so our model does not need to include an explicit authorization step. As another safe over-approximation, we do not model token expiry or revocation — once a token is issued, it is valid forever.

Omissions due to under-specification. While GNAP allows for both structured access tokens or opaque access tokens used in combination with token introspection, there is no specification for the format of structured access tokens, and so we have only modeled the case of opaque tokens together with token introspection.

Further, we only consider the case where the end user attempting to access a resource is the same as (or at least in direct out-of-band contact with) the resource owner. Otherwise, we have to model the case where an AS must get authorization from some third-party RO in order to complete a grant request. However, the details of such an interaction are out of scope of GNAP.

Security Considerations. We restrict the possible flows in our model by requiring the use of an interaction finish method. Allowing polling by the client instance can lead to AS mix-up attacks. To avoid this, we follow the recommendation of the corresponding security consideration [56, Section 13.22] and do not include polling in our model.

4.4 Definitions and Security Properties

In this section, we formalize the security properties GNAP is supposed to satisfy, i.e., security with respect to authorization and authentication, and session integrity for both authorization and authentication.

Security with respect to authorization. As mentioned before, this property states that an attacker should not be able to access the resources of an honest resource owner. There are several immediate problems with this intuition.

Most obviously, if the resource owner stores its resources on an RS that is corrupted by the attacker, there is no hope of security, as the resource server can simply give the attacker access to any resources it stores. Likewise, the resource owner and the authorization server that is managing the resource need to be honest.

A few other simple problems can lead to security failures as well. If a resource owner authorizes a corrupt client to access its resources, then the attacker naturally can learn those resources. Likewise, if the AS validly issues a bearer token (see Section 2) to a client, and that client then attempts to use the token at a corrupt RS, any security associated with that token is lost. Finally, and more subtly, if a client shares a symmetric key with an honest AS and a corrupted RS, that RS can take over a session between the client and AS, by impersonating the client to the AS, and can thereby gain access to resources that it should not be able to.

A few definitions will be useful in stating our security properties. For simplicity of presentation, we elide some of the technical details of the model. More precise definitions can be found in Appendices B, E.1, E.2, F.1 and F.2.

Definition 4.1. We identify resource owners by *identities*. An identity u consists of a name and a domain for some process. We think of this domain as indicating what process manages an account, and the name as some identifier of that account at the manager.

We also define a map governor, which maps an identity to the AS responsible for managing access to resources belonging to that identity.

Finally, we define a map ownerOfID, which takes an identity to the browser controlling the identity (i.e., the browser used by the end user to whom the identity corresponds). This can be formalized by requiring that each identity u has an associated secret credential, which is initially known only to the governor of u and some browser. That browser is then $\mathsf{ownerOfID}(u)$.

These definitions allow us to talk about identifiers for resource owners, as well as what processes we expect to be responsible for handling access to a given resource owner's resources — namely, in order for a client to access the resources belonging to an identity u, it should request permission from the governor of u, which will then request authorization from the owner of the identity. It will also be useful to be able to easily refer to the situation where the owner of u attempts to authenticate at the governor of u, authorizing such a grant request (Step $\boxed{5}$ of Figure 1).

Definition 4.2. If, during step i of a run ρ, a browser b attempts to authenticate/log-in using identity u at a process as, in order to authorize a client c, we write $\mathsf{tryLogin}^{i}_{\rho}(b, c, u, as)$.

We can now define the authorization property for GNAP.

Definition 4.3 (Security with Respect to Authorization). Let $\mathcal{GWS}$ be a GNAP web system, ρ a run of $\mathcal{GWS}$, and (S^j, E^j, N^j) a configuration in ρ. Suppose that u is an identity (of some resource owner), and RS rs stores a resource r_u on behalf of u. We say that $\mathcal{GWS}$ is *secure with respect to authorization* iff the following implication holds: Given conditions (1)–(5), the attacker cannot derive r_u from its knowledge in S^j, where conditions (1)–(5) are defined as follows: (1) rs is honest in S^j; (2) $\mathsf{governor}(u)$ is honest in S^j; (3) $\mathsf{ownerOfID}(u)$ is honest in S^j; (4) for every client instance c which is honest in S^j, c does not share a symmetric key with $\mathsf{governor}(u)$ and with a corrupted RS rs'; (5) if for some $i \leq j$, $\mathsf{tryLogin}^{i}_{\rho}(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u))$, then c is honest in S^j and, if this login succeeds and grants c a bearer token, then c does not send this token to a corrupted RS rs'.

Note that each of the five premises in the implication above corresponds to one of the situations described before where security would necessarily fail. This property then states that other than the simple, relatively apparent failures of security when some parties in the protocol are dishonest, there is no way for an attacker to derive a resource belonging to an honest user.

Security with respect to authentication. This property is similar in many respects to security with respect to authorization. Informally, we want to ensure that an attacker is not able to log in to an honest client as an honest user. To make this more precise, we need to define what it means for a user to be logged in at a client, and what identifies who they are logged in as.

Formally, we identify each login, or service session, by a nonce called the service session ID, which the client sets as a cookie to the browser that is logged in.

The client generates a service session ID when it receives authentication information (in the form of a subject identifier) from an AS, indicating that the browser that the client is interacting with is the owner of some identity u. This subject ID may be opaque to the client, but the pair of the subject ID and the AS used can uniquely identify an account on the client, and we use this pair to identify what account is logged in.

Definition 4.4. We write $\mathsf{loggedIn}^{i}_{\rho}(b, c, u, as)$ if in step i of a run ρ c sends to b a service session ID cookie, say with value *ssid*, which c associates with the account $\langle n, as\rangle$ for some n, and *as* associates n to the identity u.

With this definition, we can formalize the authentication property for GNAP:

Definition 4.5 (Security with Respect to Authentication). Let $\mathcal{GWS}$ be a GNAP web system, ρ a run of $\mathcal{GWS}$, and (S, E, N) a configuration in ρ. Suppose that u is an identity with $\mathsf{loggedIn}^{i}_{\rho}(b, c, u, as)$, and that *ssid* is the service session ID corresponding to this login. We say that $\mathcal{GWS}$ is *secure with respect to authentication* iff the following implication holds: If c, *as*, and $\mathsf{ownerOfID}(u)$ are honest in S, then the attacker cannot derive *ssid* from its knowledge in S.

As with authorization, there are several cases where a corrupt party can easily violate the intuitive security property: a corrupt client can simply allow the attacker to log in as any user, a corrupt end user can log in and then send its service session ID to the attacker, and a corrupt AS can authenticate logins as any account it manages, skipping any checks. Again, other than these obvious (and essentially unavoidable) failures of the intuitive version of authentication, if this property is provable, no attacks are possible.

Session integrity. In addition to these properties capturing that the attacker should not be able to access honest user's resources or accounts, session integrity (for both authorization and authentication) captures the intuitive idea that an honest user should not be forced to access the attacker's resources or accounts. Informally, if an honest user accesses a resource or is logged in with some account, then the user authorized that resource access or authenticated for that login. This idea is captured by the two definitions below:

Definition 4.6. Let $\mathcal{GWS}$ be a GNAP web system, ρ a run of $\mathcal{GWS}$, and (S, E, N) a configuration in ρ. Suppose that u is an identity (of some resource owner), and an RS *rs* stores a resource r_u on behalf of u, and that a client c sends r_u to a browser b. We say that $\mathcal{GWS}$ has *session integrity for authorization* iff the following implication holds: If c, *rs*, $\mathsf{governor}(u)$, and $\mathsf{ownerOfID}(u)$ are honest in S, then at some prior point, c began a GNAP flow at some *as* on behalf of b. Moreover, if *as* is honest, then there is some earlier step i in ρ such that $\mathsf{tryLogin}^{i}_{\rho}(b, c, u, \mathsf{governor}(u))$.

Definition 4.7. Let $\mathcal{GWS}$ be a GNAP web system, ρ a run of $\mathcal{GWS}$, and (S, E, N) a configuration in ρ. Suppose that u is an identity with $\mathsf{loggedIn}^{i}_{\rho}(b, c, u, as)$. We say that $\mathcal{GWS}$ has *session integrity for authentication* iff the following implication holds: If c, *as*, and b are honest in S, then there is some earlier step i in ρ such that $\mathsf{tryLogin}(b, c, u, as)$.

Note that in the case of session integrity for authorization, we need to mention two potentially different authorization servers. This is because, in principle, nothing prevents a GNAP client from being directed from one AS to another over the course of a GNAP flow, and in particular, if the first AS a client interacts with is an attacker, this attacker has a great deal of influence on what steps the client continues to take. Without this constraint that the initial AS in the flow is honest, some strange flows are possible, in which, for instance, an attacker may be able to force the end user to access resources which the attacker owns, but which are controlled by an honest AS. We discussed this problematic flow with the GNAP editors [14]. In their opinion, considering only cases where the initial AS is honest, is not a restriction. However, they will add a security consideration to the resource server specification explaining the problematic flow.

4.5 Results

Our key result, captured by the following theorem, is that the current GNAP specification including all our fixes satisfies the security properties that we set out in Section 4.4.

Theorem 4.8. *Every GNAP web system $\mathcal{GWS}$ is secure with respect to authorization and authentication, and also has session integrity with respect to authorization and authentication.*

This means that even in the presence of a strong attacker which controls the network, observing all messages and determining when and to whom they are delivered, and which can corrupt other parties on the network (see Section 3.1 for details), GNAP ensures that users are protected from a wide variety of attacks. In particular, attackers cannot access a user's resources or a user's account unless some failure which is outside the scope of the protocol occurs (e.g. the user willingly gives their credentials to the attacker, or the attacker takes over the server on which the user's resources or account data is stored). Similarly, a user cannot be forced to access an attacker's resources or account through the GNAP protocol.

We highlight that our analysis accounts for many Web features that might possibly cause attacks, e.g., in-browser communication using postMessages, cookie management, HTTP redirect behavior, various headers, or the window and document structure of browsers, including iframes. The 307 redirect attack is an example of where seemingly irrelevant details matter. Furthermore, the web system that our analysis considers can have an arbitrary number of CIs, ASs, and RSs, each of them can be corrupted at any time by the attacker, with parallel flows between these. Also, honest participants may communicate with corrupted ones, e.g., we show the properties for CIs, even if these CIs use dishonest ASs, and for honest ASs which may support dishonest CIs.

Proof Sketch. The proof of this result is quite long, being broken down into just under 30 lemmas, and relies on the technical details of the model. However, we can give a high-level overview of the main steps of the proof. First, we establish some basic properties about leakage of information, proving that various secret values are only known to the parties who are supposed to know them (e.g. private keys are only known to their owners, see Lemmas D.1 and D.2). Based on this, we then establish that key proofs (in several different forms) can be used to authenticate the sender of a message as the owner of a particular encryption key, and to validate that the message has not been tampered with (see Lemma D.8). This is a critical property, as key proofs are used in GNAP both to ensure that the AS is talking to the same client throughout a GNAP flow (i.e. the flow cannot be hijacked by the attacker) and to allow a client to prove to an RS that it is entitled to use a token bound to its private key.

Building on the ability to verify message integrity and provenance with key proofs, we establish several lemmas about the integrity of communication between a client and an AS, for instance showing that if the AS accepts a continuation request, it was sent by the same client that started the flow at the AS (see Lemma D.10). Similarly, we establish results showing that communication between a client and an end-user browser maintain continuity — that the client can identify if a different end-user browser attempts to take over a session (see Lemma D.14).

We can then combine these results about the continuity of flows with details of the implementation of the various parties to prove that access tokens and subject information are handled correctly. For instance, to show security w.r.t. authorization, we prove that resources are only sent from an RS to a client if that client presents a valid access token for the corresponding resource, along with proof that it is entitled to use the token (see Lemmas E.10 and E.11). Similarly, we show that an AS will only issue a token for a resource to a client if the owner of the resource authorizes this token to be issued (see Lemma E.12). Security w.r.t. authorization then follows by combining these — e.g. a client only receives a resource if it provided a valid access token to the RS, and the client could only have such a valid access token if it were issued by an AS, which only occurs if the client is authorized by the resource owner, thus ensuring that only authorized clients receive resources. The full proof for the authorization property and the other properties can be found in the appendix.

5 RELATED WORK

To the best of our knowledge, our work is the first formal security analysis of GNAP. Axeland and Oueidat [2] informally analyze GNAP by testing the protocol against five attack classes known for OAuth 2.0 and conclude that most of those do not apply to GNAP, except for an AS mix-up attack (first discovered in [26]). They also only consider the redirect interaction modes.

There are several formal security analyses of OAuth 2.0 and related protocols: Bansal et al. [5] use ProVerif [8] to analyze OAuth 2.0, considering some features of the Web infrastructure like cookies and origins. However, they focus on finding attacks and do not aim to provide security guarantees. Pai et al. [52] formally analyze OAuth 2.0 using the Alloy Analyzer [40] in a very limited model that does not incorporate generic web features, showing that their approach can be used to find known weaknesses. Arnaboldi and Tschofenig [1] use Tamarin [49] to analyze an abstract model of ACE-OAuth [68] (a flow specifically designed for IoT devices) without considering generic web features. Hofmeier [39] models OpenID Connect in Tamarin with a rudimentary browser model and analyzes different grant types in isolation, discovering two previously known attacks. However, the models created for these analyses, as well as the models of the underlying web infrastructure, are very limited. For example, the analysis in [52] does not cover Web features, and [5] does not consider advanced Web features like documents, iframes, and postMessages.

To date, the WIM is the most detailed and expressive model of Web infrastructure, and as already mentioned, has successfully been used to analyze a variety of protocols, including several in the OAuth family ([21, 26, 28]), as well as Mozilla Browser ID [23, 24] and the W3C Web Payment APIs [16]. All of the analyses using the WIM, including ours, rely on manual proofs. To the best of our knowledge, there is no mechanized model of the WIM with a level of detail comparable to the WIM yet. Building such a model is a challenging future task.

We refer the reader to [10] for an overview of formal methods in web security.

6 CONCLUSION

In this paper, we performed the first formal security analysis of GNAP. To this end, we built a detailed formal model of GNAP based on the WIM, which includes several start and finish interaction methods, both the software-only and end-user cases, arbitrary numbers of continuation requests, grant negotiation between CI and AS, token introspection, key-bound and bearer access tokens, subject information grants, different key proof methods, all while accounting for details of the Web infrastructure like different HTTP redirection codes, JavaScript running in browsers, important HTTP headers, and so on.

We formalized central security properties that are expected from and have previously been applied to authorization and authentication protocols. We tried to prove these properties, but discovered attacks that break them. These attacks were reported to the IETF's GNAP Working Group along with proposed fixes, which are now part of the specifications.

We incorporated these fixes into our formal model and proved that the fixed model fulfills the security properties. As our model accounts for many Web features, our security proof excludes large classes of attacks, even in the presence of a network attacker that can use all web features, e.g., provide malicious scripts, manipulate headers, etc., and can get hold of (sender-constrained) access tokens, and even if an unlimited number of users, CIs, ASs, and RSs, all of which the adversary can corrupt at any point, operate with an unlimited number of parallel sessions.

Considering GNAP's progress towards an IETF standard with the core specification being in the Working Group Last Call at the time of this writing, our analysis is just in time to support the standardization of an important protocol in terms of its security. Our work also shows that formal analysis in a meaningful and rich model is necessary for complex protocols, like GNAP, as even very experienced protocol designers easily overlook not only new attacks but also attacks previously found on related protocols.

Acknowledgements: This research was supported by the DFG through grant KU 1434/12-1.

REFERENCES

[1] Luca Arnaboldi and Hannes Tschofenig. 2019. A formal model for delegated authorization of IoT devices using ACE-OAuth. https://homepages.inf.ed.ac.uk/larnibol/img/publications/Paper-03.pdf 4th OAuth Security Workshop.

[2] Åke Axeland and Omar Oueidat. 2021. *Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol.* Master's thesis. Chalmers University of Technology, University of Gothenburg. https://odr.chalmers.se/items/7d36a5d4-c295-4270-886b-d5ed1154a8e8

[3] Annabelle Backman, Justin Richer, and Manu Sporny. 2022. *HTTP Message Signatures*. Internet-Draft draft-ietf-httpbis-message-signatures-15. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-httpbis-message-signatures/15/ Work in Progress.

[4] Annabelle Backman, Marius Scurtescu, and Prachi Jain. 2022. *Subject Identifiers for Security Event Tokens*. Internet-Draft draft-ietf-secevent-subject-identifiers-14. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-secevent-subject-identifiers/14/ Work in Progress.

[5] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2014. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *Journal of Computer Security* 22, 4 (2014), 601–657.

[6] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. 2012. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *25th IEEE Computer Security Foundations Symposium, CSF 2012*, Stephen Chong (Ed.). IEEE Computer Society, 247–262.

[7] Vittorio Bertocci. 2021. JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens. RFC 9068. https://doi.org/10.17487/RFC9068

[8] B. Blanchet. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*. IEEE Computer Society, 82–96.

[9] Bruno Blanchet. 2013. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science)*, Vol. 8604. Springer, 54–87.

[10] Michele Bugliesi, Stefano Calzavara, and Riccardo Focardi. 2017. Formal methods for web security. *J. Log. Algebraic Methods Program.* 87 (2017), 110–126. https://doi.org/10.1016/j.jlamp.2016.08.006

[11] Brian Campbell, John Bradley, Nat Sakimura, and Torsten Lodderstedt. 2020. OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. RFC 8705. https://doi.org/10.17487/RFC8705

[12] Scott Cantor, John Kemp, Rob Philpott, Eve Maler, and Eric Goodman (ed.). 2015. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite. https://www.oasis-open.org/committees/download.php/56776/sstc-saml-core-errata-2.0-wd-07.pdf

[13] cwaldm. 2022. Issue 461 - Missing Details for Verificaton of Identity Information. GitHub Issue. https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/461

[14] cwaldm. 2022. Issue 56 - RS validation of access token. GitHub Issue. https://github.com/ietf-wg-gnap/gnap-resource-servers/issues/56

[15] William Denniss, John Bradley, Michael Jones, and Hannes Tschofenig. 2019. OAuth 2.0 Device Authorization Grant. RFC 8628. https://doi.org/10.17487/RFC8628

[16] Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, Nils Wenzler, and Tim Würtele. 2022. A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification. In *43rd IEEE Symposium on Security and Privacy (S&P 2022)*, Vol. 1. IEEE Computer Society, 134–153. https://publ.sec.uni-stuttgart.de/dohosseynikuestersschmitzwenzlerwuertele-sp-2022.pdf

[17] Danny Dolev and Andrew C. Yao. 1983. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208.

[18] Dropbox Platform Team. 2020. OAuth Guide. https://developers.dropbox.com/oauth-guide.

[19] Daniel Fett. 2022. *FAPI 2.0 Attacker Model, Draft 02*. OpenID Foundation. https://openid.net/specs/fapi-2_0-attacker-model-02.html

[20] Daniel Fett. 2022. *FAPI 2.0 Security Profile, Draft 02*. OpenID Foundation. https://openid.net/specs/fapi-2_0-security-02.html

[21] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. 2019. An Extensive Formal Security Analysis of the OpenID Financial-Grade API. In *40th IEEE Symposium on Security and Privacy (S&P 2019)*. IEEE Computer Society, Los Alamitos, CA, USA, 1054–1072. https://doi.org/10.1109/SP.2019.00067

[22] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. 2019. *An Extensive Formal Security Analysis of the OpenID Financial-grade API*. Technical Report arXiv:1901.11520. arXiv. Available at http://arxiv.org/abs/1901.11520.

[23] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2014. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *35th IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE Computer Society, 673–688.

[24] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2015. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 9326. Springer, 43–65.

[25] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2015. SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. ACM, 1358–1369.

[26] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. ACM, 1204–1215.

[27] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. *A Comprehensive Formal Security Analysis of OAuth 2.0*. Technical Report arXiv:1601.01229. arXiv. http://arxiv.org/abs/1601.01229

[28] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2017. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. In *IEEE 30th Computer Security Foundations Symposium (CSF 2017)*. IEEE Computer Society.

[29] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2022. The Web Infrastructure Model (WIM). https://www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf

[30] GitHub Docs. 2023. Authorizing OAuth Apps. https://docs.github.com/en/developers/apps/building-oauth-apps/authorizing-oauth-apps.

[31] Google. 2022. OAuth 2.0 for TV and Limited-Input Device Applications. https://developers.google.com/identity/protocols/oauth2/limited-input-device.

[32] Google. 2022. Using OAuth 2.0 to Access Google APIs. https://developers.google.com/identity/protocols/oauth2.

[33] Dick Hardt (ed.). 2012. The OAuth 2.0 Authorization Framework. RFC 6749. https://doi.org/10.17487/RFC6749

[34] Florian Helmschmidt. 2021. Issue 364 - End user/client instance mix-up attack. GitHub Issue. https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/364

[35] Florian Helmschmidt. 2021. Issue 47 - How does token introspection handle symmetric keys? GitHub Issue. https://github.com/ietf-wg-gnap/gnap-resource-servers/issues/47

[36] Florian Helmschmidt. 2021. Issue 48 - Security Consideration: Derivation of Stolen Tokens. GitHub Issue. https://github.com/ietf-wg-gnap/gnap-resource-servers/issues/48

[37] Florian Helmschmidt. 2021. Pull Request 351 - Add redirection status code security considerations. GitHub Pull Request. https://github.com/ietf-wg-gnap/gnap-core-protocol/pull/351

[38] Florian Helmschmidt. 2022. Issue 390 - Clarified user presence on interaction finish methods. GitHub Commit. https://github.com/ietf-wg-gnap/gnap-core-protocol/pull/390/commits/b028a1e363e90ad1c2711bd8244ea76e3957f935

[39] Xenia Hofmeier. 2019. Formal Analysis of Web Single-Sign On Protocols using Tamarin. Bachelor's thesis, Swiss Federal Institute of Technology in Zurich, Switzerland.

[40] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. 2000. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf (Eds.). ACM, 730–733. https://doi.org/10.1145/337180.337616

[41] Michael Jones. 2015. JSON Web Key (JWK). RFC 7517. https://doi.org/10.17487/RFC7517

[42] Michael Jones, John Bradley, and Nat Sakimura. [n.d.]. RFC7515 – JSON Web Signature (JWS). IETF. May 2015. https://tools.ietf.org/html/rfc7515.

[43] Pieter Kasselman, Daniel Fett, and Filip Skokan. 2022. *Cross-Device Flows: Security Best Current Practice*. Internet-Draft draft-ietf-oauth-cross-device-security-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-oauth-cross-device-security/00/ Work in Progress.

[44] Open Banking Limited. 2022. Open Banking UK. https://www.openbanking.org.uk/.

[45] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. 2022. *OAuth 2.0 Security Best Current Practice*. Internet-Draft draft-ietf-oauth-security-topics-21. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-21 Work in Progress.

[46] Torsten Lodderstedt, Brian Campbell, Nat Sakimura, Dave Tonge, and Filip Skokan. 2021. OAuth 2.0 Pushed Authorization Requests. RFC 9126. https://doi.org/10.17487/RFC9126

[47] Maciej Machulak and Justin Richer. 2018. User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization. https://docs.kantarainitiative.org/uma/wg/rec-oauth-uma-grant-2.0.html.

[48] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. 2017. SoK: Single Sign-On Security – An Evaluation of OpenID Connect. In *IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*.

[49] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference (CAV 2013) (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 696–701.

[50] Meta for Developers Documents. 2023. Facebook Login for the Web with the JavaScript SDK. https://developers.facebook.com/docs/facebook-login/web/.

[51] Vladislav Mladenov, Christian Mainka, Julian Krautwald, Florian Feldmann, and Jörg Schwenk. 2016. On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect. *CoRR* abs/1508.04324v2 (2016). http://arxiv.org/abs/1508.04324v2

[52] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, and Sanjay Singh. 2011. Formal Verification of OAuth 2.0 Using Alloy Framework. In *CSNT '11 Proceedings of the 2011 International Conference on Communication Systems and Network Technologies*. Proceedings of the International Conference on Communication Systems and Network Technologies, 655–659.

[53] PedramHD. 2022. Issue 481 - Unguessability of User Codes. GitHub Pull Request. https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/481

[54] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446

[55] Justin Richer. 2018. Moving On from OAuth 2: A Proposal. https://justinsecurity.medium.com/moving-on-from-oauth-2-629a00133ade.

[56] Justin Richer and Fabien Imbault. 2023. *Grant Negotiation and Authorization Protocol.* Internet-Draft draft-ietf-gnap-core-protocol-15. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-gnap-core-protocol/15/ Work in Progress.

[57] Justin Richer and Fabien Imbault. 2023. *Grant Negotiation and Authorization Protocol Resource Server Connections.* Internet-Draft draft-ietf-gnap-resource-servers-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-gnap-resource-servers/03/ Work in Progress.

[58] Justin Richer, Aaron Parecki, and Fabien Imbault. 2021. *Grant Negotiation and Authorization Protocol.* Internet-Draft draft-ietf-gnap-core-protocol-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-gnap-core-protocol/08/ Work in Progress.

[59] Justin Richer, Aaron Parecki, and Fabien Imbault. 2022. *Grant Negotiation and Authorization Protocol Resource Server Connections.* Internet-Draft draft-ietf-gnap-resource-servers-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-gnap-resource-servers/02/ Work in Progress.

[60] Justin Richer (ed.). 2015. OAuth 2.0 Token Introspection. RFC 7662. https://doi.org/10.17487/RFC7662

[61] Justin Richer (ed.), Michael Jones, John Bradley, and Maciej Machulak. 2015. OAuth 2.0 Dynamic Client Registration Management Protocol. RFC 7592. https://doi.org/10.17487/RFC7592

[62] Justin Richer (ed.), Michael Jones, John Bradley, Maciej Machulak, and Phil Hunt. 2015. OAuth 2.0 Dynamic Client Registration Protocol. RFC 7591. https://doi.org/10.17487/RFC7591

[63] Nat Sakimura. 2022. Issue 543 - Browser swap attack explained on 2022-09-28. Bitbucket Issue. https://bitbucket.org/openid/fapi/issues/543

[64] N. Sakimura, J. Bradley, and E. Jay. 2021. Financial-grade API Security Profile 1.0 - Part 1: Baseline. https://openid.net/specs/openid-financial-api-part-1-1_0.html. OpenID Foundation.

[65] N. Sakimura, J. Bradley, and E. Jay. 2021. Financial-grade API Security Profile 1.0 - Part 2: Advanced. https://openid.net/specs/openid-financial-api-part-2-1_0.html. OpenID Foundation.

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

[67] Nat Sakimura (ed.), John Bradley, and Naveen Agarwal. 2015. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636. https://doi.org/10.17487/RFC7636

[68] Ludwig Seitz, Göran Selander, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. 2022. Authentication and Authorization for Constrained Environments Using the OAuth 2.0 Framework (ACE-OAuth). RFC 9200. https://doi.org/10.17487/RFC9200

[69] Yaron Sheffer. 2023. GNAP core protocol - Publication has been requested. GNAP Mailing List. https://mailarchive.ietf.org/arch/msg/txauth/4q2W9cTho2chk4IunHKXztcorO0/.

[70] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. 2016. Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016-05). ACM. https://doi.org/10.1145/2897845.2897874

[71] Kristina Yasuda. 2021. Issue 1269 - Add Security Considerations for Cross-device SIOP. Bitbucket Issue. https://bitbucket.org/openid/connect/issues/1269

A FORMAL MODEL OF GNAP

This appendix contains our formal model of GNAP, which is used to prove the security properties defined in Appendices E.2 and F.2. Our model is based on the consolidated version of the WIM as described in [29]. Here we only describe the details needed for our analysis and refer the reader to [29] for the underlying model. Appendix A.1 will explain what adjustments were made to the WIM for modeling GNAP. Appendix A.2 provides an outline of the model, while Appendix A.3 provides decisions and notes as well as limitations regarding our modeling. The assignment of addresses and domain names to processes is explained in Appendix A.4. Appendix A.5 explains which nonces are used in the model. Appendix A.6 describes how identities of resource owners are modeled. Corruption of processes is discussed in Appendix A.7. Network attackers and browsers are then covered in Appendix A.8 and Appendix A.9, respectively. Appendix A.10 defines helper functions that are used in the modeling of the servers. Modeled as servers are client instances, authorization servers, and resource servers which are defined in Appendix A.11, Appendix A.12 and Appendix A.13, respectively.

A.1 Adjustments to the Web Infrastructure Model

A.1.1 Headers. GNAP uses several headers for signing and authorizing requests. Since only certain headers with certain values are defined in the WIM, we have to redefine the `Authorization` header and add three more headers for modeling GNAP.

In our modeling of GNAP, the `Authorization` header is a term of the form

$$\langle \texttt{Authorization}, \langle \mathit{scheme}, n \rangle \rangle$$

with $n \in \mathcal{N}$ and $\mathit{scheme} \in \{\texttt{GNAP}, \texttt{Bearer}\}$. The nonce n models an access token and *scheme* is the used HTTP Authentication scheme. This header is used in continuation requests from client instances to ASs and in resource requests from client instances to RSs. For continuation requests, the scheme is always `GNAP`, while for resource requests it can be either `GNAP` or `Bearer`, depending on whether the specified access token is a bound access token or a bearer token.

In addition, we use the following headers for signing requests:

- $\langle \texttt{Digest}, t \rangle$ with $t \in \mathcal{T}_{\mathcal{N}}$. In this header, t is the hash value of the body of the request m, with which this header is sent, i.e., $\mathsf{hash}(m.\mathsf{body})$. This header is used together with the `Signature-Input` header and the `Signature` header to simulate HTTP Message Signature key proofs.
- $\langle \texttt{Signature-Input}, t \rangle$ with $t \in \mathcal{T}_{\mathcal{N}}$. In this header, t is a sequence that denotes the input to the signature algorithm, i.e., everything that is signed by the signature. This includes the value of the `Digest` header, meaning that the body of the request is also signed.
- $\langle \texttt{Signature}, t \rangle$ with $t \in \mathcal{T}_{\mathcal{N}}$. In this header, t is the signature value resulting from the execution of the signature algorithm. If a symmetric key is used by the signer, this value can also be a MAC.

A.1.2 Browser Model. To use the user code interaction start mode we have adapted the browser model of the WIM [29]. These adaptations simulate that an end user receives a user code, remembers it and then enters it as part of the login process on the user code interaction page of the AS. For this we have made the following changes.

To Definition 32 of the WIM, we add two more types of references for requests. $\langle \texttt{START}, \mathit{nonce} \rangle$ is used when a browser prompts a client instance to send a grant request to an AS, where *nonce* is a window reference. $\langle \texttt{UCL}, \mathit{nonce} \rangle$ is used when a browser wants to perform a login using a received user code, where *nonce* is also a window reference.

To the definition of the set of states $Z_{\text{webbrowser}}$ of a web browser atomic DY process in Definition 33 of the WIM, we add the following new subterms:

- $\mathit{pendingInteractions} \in \left[\mathcal{N} \times \mathsf{URLs}\right]$ is used to store received user codes and the corresponding URLs of the user code interaction pages

- *usedCIs* $\in \left[\mathcal{N} \times \mathsf{Doms}\right]$ is used to store the domains of the client instances through which a user code was obtained

In the upcoming code sections, old code taken from the WIM is shown in this color, while new or changed code is shown in black.

To RUNSCRIPT (Algorithm A.1) we have added a new command ⟨STARTGRANT, *url*, *as*⟩. This is used by the index page of the client instances to send a request to the client instance at the URL *url*, which then sends a grant request to the AS at the domain *as* (if the client instance is configured to use this AS). Thereby the new START reference type is used.

Algorithm A.1 Web Browser Model: Execute a script.

```
function RUNSCRIPT($\overline{w}, \overline{d}, s'$)
    ⋮
    switch command do
        case ⟨STARTGRANT, url, as⟩
            let reference := ⟨START, $s'.\overline{w}$.nonce⟩
            let req := ⟨HTTPReq, $v_4$, POST, url.host, url.path,
                ↪ url.parameters, ⟨⟩, as⟩
            let $s'$ := CANCELNAV(reference, $s'$)
            call HTTP_SEND(reference, req, url, docorigin,
                ↪ referrer, referrerPolicy, $s'$)
        case ⟨HREF, url, hrefwindow, noreferrer⟩
    ⋮
```

In PROCESSRESPONSE (Algorithm A.2) we change the reference type of a START type request to REQ (Line 28) when handling redirects, since the redirect interaction start mode is used when receiving a redirect, but we only need the START type for user code interactions. Since the interaction start mode used is not known to the browser when sending the request, we always use the START type for the request and then change the type if the redirect interaction start mode is used.

If the user code interaction start mode is used, the browser stores the user code together with the URL of the user code interaction page in *pendingInteractions* (Line 37), and the user code together with the domain of the client instance used for the request in *usedCIs* (Line 38).

The stored user codes are then used in the main algorithm of the browser (Algorithm A.3). To simulate a login, we have added the `login` case to the possible actions a browser can perform when triggered. When `login` is chosen in Line 8 and there is an entry in *pendingInteractions*, the browser sends a GET request to the user code interaction page from the entry. In the request, the browser includes the user code as a parameter. The UCL reference type is used for this request. The AS uses the user code to identify the domain of the client instance that sent the corresponding grant request to the AS and transmits this domain to the browser in the response. This is done to prevent the client instance mix-up attack (see Section 3.2). For details on how we handle this attack in our model, see Appendix A.3.

In the main algorithm, we also set the two new subterms *pendingInteractions* and *usedCIs* to ⟨⟩ if the browser is closed (Lines 60f.).

In PROCESSRESPONSE (Algorithm A.2), when a response is received to a request with reference type UCL, the user code used is obtained from the parameters in *requestUrl*. The user code can then be used to obtain the domain of the client instance used by the browser from *usedCIs*. The browser takes the domain of the client instance that sent the grant request to the AS from the body of the response. If the two domains match, the script selected by the AS receives the used user code as input via *scriptinputs*. Otherwise, *scriptinputs* remains empty (Lines 46ff.). In Lines 50f. the used entries from *pendingInteractions* and *usedCIs* are removed so that they are not used again.

A.1.3 Definition of stored in *and* initially stored in*.* We additionally introduce the following formulation that is used to describe the states of processes:

Definition A.1. We say that a term t is *stored in* an atomic DY process p in a state S if t is a subterm of $S(p)$. If $S = s_0$, we say that t is *initially stored in* p.

In addition to *stored in* and *initially stored in*, we will also use the formulation *appears only as a public key* from Definition 50 in [29] to describe states.

A.2 Outline

We model GNAP as a web system in the WIM as defined in [29].

We call a web system $\mathcal{GWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ a *GNAP web system* if it is of the form described in this and the following sections.

Similar to the work of Fett et al. [27], the system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker process (in Net), a finite set B of web browsers, a finite set CI of web servers for client instances, a finite set AS of web servers for authorization servers, and a finite set RS of web servers for resource servers with $\mathsf{Hon} = \mathsf{B} \cup \mathsf{CI} \cup \mathsf{AS} \cup \mathsf{RS}$. More details on the processes in $\mathcal{W}$ are provided below. We do not model DNS servers, as they are subsumed by the network attacker. Table 1 shows the set of scripts Scripts and their respective string representations that are defined by the mapping script. The set E^0 contains only trigger events.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
script_ci_index	`script_ci_index`
script_as_login	`script_as_login`

Table 1: List of scripts in $\mathcal{S}$ and their respective string representations.

A.3 Modeling Remarks and Limitations

This section contains comments on our overall modeling of GNAP and its limitations. See the descriptions of the algorithms in Appendix A.11, Appendix A.12, and Appendix A.13 for role-specific modeling remarks.

Preventing Client Instance Mix-Up Attacks. As described in Section 3.2, most variants of the client instance mix-up attack can only be prevented if the AS provides EU with enough reliable information on the CI which is about the be authorized and the EU

Algorithm A.2 Web Browser Model: Process an HTTP response.

```
function PROCESSRESPONSE($response, reference, request, requestUrl, key, f, s'$)
    ⋮
            let $referrerPolicy$ := $response$.headers[ReferrerPolicy]
            if $\pi_1(reference) \equiv$ START then
                let $reference := \langle$REQ$, \pi_2(reference)\rangle$        ▷ Redirect interaction start mode is used
            call HTTP_SEND($reference, req, url, origin, referrer, referrerPolicy, s'$)
        else stop $\langle\rangle, s'$
    switch $\pi_1(reference)$ do
        case START
            if userCode $\notin response \vee$ userCodeUrl $\notin response$ then
                stop $\langle\rangle, s'$        ▷ The response must contain a user code and the URI of the user code interaction page
            let $userCode := response$[userCode]
            let $userCodeUrl := response$[userCodeUrl]
            let $domainCI := requestUrl$.host
            let $s'$.pendingInteractions := $s'$.pendingInteractions $+^{\langle\rangle} \langle userCode, userCodeUrl\rangle$
            let $s'$.usedCIs := $s'$.usedCIs $+^{\langle\rangle} \langle userCode, domainCI\rangle$
        case UCL
            let $\overline{w} \leftarrow$ Subwindows($s'$) such that $s'.\overline{w}$.nonce $\equiv \pi_2(reference)$ if possible; otherwise stop
            if $response$.body $\not\sim \langle *, *\rangle$ then stop $\langle\rangle, s'$
            let $script := \pi_1(response$.body$)$
            let $domainCI := \pi_2(response$.body$)$
            let $userCode := requestUrl$.parameters[user-code]
            let $domainUsedCI := s'$.usedCIs[$userCode$]
            if $domainCI \equiv domainUsedCI$ then
                let $scriptinputs$ := [userCode:$userCode$]
            else
                let $scriptinputs := \langle\rangle$
            let $s'$.pendingInteractions := $s'$.pendingInteractions $- userCode$
            let $s'$.usedCIs := $s'$.usedCIs $- userCode$
            let $d := \langle v_7, requestUrl, response$.headers$, referrer, script, \langle\rangle, scriptinputs, \langle\rangle, \top\rangle$
            if $s'.\overline{w}$.documents $\equiv \langle\rangle$ then
                let $s'.\overline{w}$.documents $:= \langle d\rangle$
            else
                let $\overline{i} \leftarrow \mathbb{N}$ such that $s'.\overline{w}$.documents.$\overline{i}$.active $\equiv \top$
                let $s'.\overline{w}$.documents.$\overline{i}$.active $:= \bot$
                remove $s'.\overline{w}$.documents.$(\overline{i}+1)$ and all following documents from $s'.\overline{w}$.documents
                let $s'.\overline{w}$.documents := $s'.\overline{w}$.documents $+^{\langle\rangle} d$
            stop $\langle\rangle, s'$
        case REQ
    ⋮
```

carefully checks this information. We model this as part of our browser and AS models: The browser used by EU stores the CI it talks to, and in the authorization step, AS includes the CI it associates with the ongoing request. If the two mismatch, the browser (i.e., the end user we model as part of the browser) aborts the flow.

Modeling of MTLS. GNAP uses MTLS as one of its key proofing methods. MTLS has already been modeled within the WIM by Fett et al. [21], which is why we adopt the modeling provided there.

Unique URLs. In GNAP, unique URLs are used in various places to associate a request with a particular flow. The examples in the protocol use random values within the path. However, since GNAP does not prohibit using the query string for this purpose, we use a parameter whose value contains a nonce to uniquely associate a URL.

Empty HTTP Responses and Error Messages. Using the push interaction finish mode results in two empty HTTP responses that do not convey any information except that the associated request was successful. The model does not include these responses because an attacker cannot extract any information from them. We do not model error messages.

Algorithm A.3 Web Browser Model: Main algorithm.

```
Input: $\langle a, f, m\rangle, s$
    ⋮
if $m \equiv$ TRIGGER then
    let $switch \leftarrow$ {script, login, urlbar, reload, forward, back}
    if $switch \equiv$ script then
        ⋮
        call RUNSCRIPT($\overline{w}, \overline{d}, s'$)
    else if $switch \equiv$ login then                    ▷ Perform login using user code
        if $s'$.pendingInteractions $\equiv \langle\rangle$ then stop    ▷ No user code has been received yet or all received user codes have been used
        let $newwindow \leftarrow \{\top, \bot\}$
        if $newwindow \equiv \top$ then                   ▷ Create a new window
            let $windownonce := \nu_1$
            let $w' := \langle windownonce, \langle\rangle, \bot\rangle$
            let $s'$.windows := $s'$.windows $+^{\langle\rangle}$ $w'$
        else                                              ▷ Use existing top-level window
            let $windownonce := s'.\overline{tlw}.nonce$
        let $\langle userCode, userCodeUrl\rangle \leftarrow s'$.pendingInteractions
        let $req := \langle$HTTPReq, $\nu_2$, GET, $userCodeUrl$.host, $userCodeUrl$.path, [user-code:$userCode$], $\langle\rangle, \langle\rangle\rangle$
        call HTTP_SEND($\langle$UCL, $windownonce\rangle$, $req$, $url$, $\bot, \bot, \bot$, $s'$)
    else if $switch \equiv$ urlbar then
        ⋮
else if $m \equiv$ CLOSECORRUPT then
    let $s'$.pendingInteractions := $\langle\rangle$
    let $s'$.usedCIs := $\langle\rangle$
    let $s'$.secrets := $\langle\rangle$
    ⋮
```

Cross Domain Referrer Header Leakage. If the redirect interaction start mode is used, the RO may click on a link on the AS's interaction page after being redirected to the AS. This can leak information found in the URL to an attacker through the HTTP `Referer` header. To prevent this, GNAP recommends redirecting the RO to an internal interstitial page without any identifying or sensitive information in the URL before the actual redirect is performed. This way, after the second redirect, no part of the original interaction URL will be found in the `Referer` header (cf. [56]). For simplicity, we prevent such a leak not by using interstitial pages, but by using the HTTP `Referrer-Policy` header with the `origin` directive, which also prevents the described problem.

Key References and Instance Identifiers. In GNAP both *key references* and *instance identifiers* are used. A key reference refers to a specific key, as the name implies, while an instance identifier can also have additional information associated. This information can be displayed to the RO when authorizing a request, for example. Since modeling this information in the WIM would not be meaningful, we do not model it so that an instance identifier encompasses the same information as a key reference. Therefore, we do not explicitly model key references, but instead sometimes use instance identifiers as key references.

Keys Used for Access Tokens. As recommended by GNAP in response to the stolen token replay attack described in Section 3.4, we use each client instance key with only one AS. Therefore, for MTLS, we do not use the TLS keys that a server following the WIM's generic HTTPS server model has, but keys specifically intended for key proofing methods. This models the use of self-signed certificates for MTLS as allowed by GNAP. It may not be possible to use the TLS keys for MTLS, since the number of domains of a client instance may be smaller than the number of ASs the client instance is configured to use. For binding access tokens to keys different from the client instance's key, the GNAP specification does not specify how the corresponding (private) keys can be distributed between AS and CI. Thus we only allow this option for pre-registered client instances, and assume that the AS and the CI have a shared set of keys for this purpose. Again, to prevent a variant of the stolen token replay attack, we require this sets of keys to be pairwise disjoint for different ASs.

Modeled Interaction Start Modes. Of the interaction start modes existing at the time of this work, we only model the redirect mode and the user code mode.

Resource Access Rights. For simplicity, we do not model a particular resource access model (of which GNAP also does not prescribe a particular one). Therefore, the grant requests in our model do not describe which resources and rights should be associated with a requested access token. Instead, an access token is always associated with the RO for which the access token is issued, and an access token can always be used to access all resources of the associated RO. Introspection responses thus only specify which RO's resources can be accessed with an access token. If the RO is an end user, it is identified by its identity (see Appendix A.6). A client instance is identified by its instance identifier at the respective AS.

Since all resources of the RO can always be accessed with an access token, we do not model the possibilities of access token splitting and requesting multiple access tokens using a single grant request. This also means that a client instance cannot extend or restrict the requested rights or resources through a continuation request. However, in our model, a continuation request can be used to request different values than before, for example, a bearer token instead of a key-bound access token or the request is extended to include a subject identifier of the RO.

Authorization During Interaction with the RO. Since we do not model a particular resource access model, the RO does not need to be informed during the interaction with the AS which resources the client instance wants to access. Therefore, in our model, there is no explicit authorization of grant requests by the RO. Instead, a login by the RO at the AS also means that the grant request of the client instance is authorized.

Since there is no possibility to extend the requested rights or resources through a continuation request, in our model the interaction with the resource owner is always only required for the first grant request of a flow, all subsequent continuation requests are automatically accepted by the AS.

Not Using an Interaction Finish Mode. Using active polling instead of an interaction finish mode is insecure, since in this case, an AS mix-up attack is possible. Therefore, GNAP recommends using an interaction finish mode whenever possible, which is why we have not included the possibility of polling in our model.

Token Management. In our model, once issued, access tokens are valid forever since in a secure protocol an attacker should never succeed in using an access token not issued to him. Therefore, we do not model the token management functions offered by GNAP, such as rotating and revoking access tokens. We therefore also do not model the durable flag, since our modeling behaves as if the durable flag is always set.

Relation between End User and RO. In our modeling, the end user always corresponds to the RO. Thus, we do not model a scenario where an AS determines that a particular RO is needed to authorize a grant request and contacts it, for example, via asynchronous authorization. This is also because a concrete implementation of asynchronous authorization is out of scope for GNAP. Consequently, there is no scenario in our modeling where multiple ROs have to approve a grant request.

Note that this does not mean that an RO is always an end user since in the case of software-only authorization an RO can also be a client instance.

Access Token Formats. GNAP allows both the use of nonces as access tokens in combination with token introspection and the use of structured access tokens, both of which have their advantages and disadvantages. We chose to use nonces as access tokens in our modeling because it allows us to model token introspection as well, while GNAP does not specify a particular format for structured access tokens, which would make it more difficult to model them.

Downstream Tokens. We did not include the possibility of deriving downstream tokens in our model because, at the time of this work, it was not yet fully specified and still subject to the issue [36].

Transfer of Subject Identifiers from CI to AS. GNAP allows a client instance to transfer a subject identifier of its current end user within a grant request to the AS if the client instance knows such a subject identifier (e.g., from a previous grant response of the same AS). The AS can use this subject identifier, for example, to reject a login of the end user during the interaction if the end user logs in with a different subject identifier than the one submitted by the client instance in the grant request. In our model, if a client instance receives a request to start a grant request from a browser and this request includes a session ID for which the client instance has already received a subject ID from the used AS in the past, the client instance will send the subject ID to the AS in the grant request and the AS will reject the login if the end user logs in with a different subject ID. We decided to reject the login in this case because an end user cannot usually log in to an AS with different identities within a single session at the client instance. Instead, we modeled a logout endpoint that allows an end user to logout from the client instance so that a new session ID is assigned to the browser, preventing the client instance from associating an old subject identifier with a new request from that browser and thus allowing the end user to log in to the AS under a different identity.

User Codes. In our model we choose user codes to be fresh nonces. The GNAP specification recommends user codes to be no more than eight characters long and requires them to consist only of easily typeable characters. Since user codes are supposed to be unguessable [53], choosing them as nonces is a safe over-approximation.

AS Proxy Setting. GNAP allows an AS proxy setting, where the AS the client talks to delegates access control to another AS. The initial AS acts itself as a client instance towards the other AS, gets an access token from the second AS and passes that token on to the client instance. The client instance can use this token as usual at an RS. The RS will do token introspection at the issuing AS (not the initial AS).

This immediately works for bearer tokens and tokens bound to a new key chosen by the issuing AS. In the latter case, the initial AS just passes this new key along with the token to the client instance. If, however, the token is bound to the key of the proxy AS, the client instance can not provide a key proof for this key when using the token. To still allow this setting, the initial AS and the issuing AS both have to know the key of the client instance and when the initial AS starts a flow at the issuing AS it has to tell the other AS which client instance the token is for. This additional communication between the two ASs is outside the scope of GNAP.

In our model, we allow this setting by considering the two ASs to be the same protocol participant. This guarantees that both ASs know the client instance's key without having to specify the inter AS communication.

With this we potentially "overlook" flows where only one of the two ASs in the original setting is corrupted. If the issuing AS is corrupt, no additional attacks are possible in the proxy setting that are not already possible in the usual case, where there is only one

AS. (I.e., the honest initial AS cannot prevent attacks that would not be possible otherwise.) We discussed the case where only the initial AS is corrupt (but the issuing AS is honest) with the GNAP editors. The result was that they only consider flows where the initial AS is honest, and hence do not expect any security guarantees in that case. Thus, our way of modeling the proxy scenario does not overlook new attacks.

A.4 Addresses and Domain Names

We will now define the atomic Dolev-Yao processes in $\mathcal{GWS}$ and their addresses, domain names, keys and secrets in more detail.

Similar to [26], the set IPs contains for the network attacker in Net, every client instance in CI, every authorization server in AS, every resource server in RS, and every browser in B a finite set of addresses each. The set Doms contains a finite set of domains for every client instance in CI, every authorization server in AS, every resource server in RS, and the network attacker in Net. Browsers (in B) do not have a domain.

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

A.5 Keys and Secrets

Also similar to [26], the set $\mathcal{N}$ of nonces is partitioned into six sets, the infinite sequence N and finite sets K_{TLS}, K_{KP}, KeyIDs, Passwords, and ProtectedResources. We thus have

$$\mathcal{N} = \underbrace{N}_{\text{infinite sequence}} \dot\cup \underbrace{K_{\text{TLS}}}_{\text{finite}} \dot\cup \underbrace{K_{\text{KP}}}_{\text{finite}} \dot\cup \underbrace{\mathsf{KeyIDs}}_{\text{finite}} \dot\cup \underbrace{\mathsf{Passwords}}_{\text{finite}} \dot\cup \underbrace{\mathsf{ProtectedResources}}_{\text{finite}} .$$

These sets are used as follows:

- The set N contains the nonces that are available for each DY process in $\mathcal{W}$ (it can be used to create a run of $\mathcal{W}$).
- The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define:
  $\mathit{tlskeys}^p = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(p)\}\rangle$.
- The set K_{KP} contains the keys that will be used for the key proofing methods.
- The set KeyIDs contains identifiers that will be used by the client instances, the ASs, and the RSs to identify keys used by the client instances and the RSs to sign their requests.
- The set Passwords is the set of passwords (secrets) the browsers share with the ASs. These are the passwords the ROs use to log in at the ASs (if the RO is not a client instance).
- The set ProtectedResources contains a secret for each combination of AS, RO, and RS. These are thought of as protected resources that only the RO should be able to access. An RO can thus access exactly one resource at a given RS using a given AS. This resource subsumes all possible resources for which the RO may request access using GNAP since, as mentioned above, we do not model a specific resource access model. Note that in our model, an RO can be not only an end user but also a client instance accessing resources using software-only authorization.

A.6 Identities and Passwords

As in [26], we use identities to model an RO logging in at an AS. Identities consist, similar to email addresses, of a username and a domain part. They are defined as follows:

Definition A.2. An *identity* i is a term of the form $\langle \mathit{name}, \mathit{domain}\rangle$ with $\mathit{name} \in \mathbb{S}$ and $\mathit{domain} \in \mathsf{Doms}$. We set ID to be the set of all identities and refer to the set $\{\langle \mathit{name}, \mathit{domain}\rangle \in \mathsf{ID} \mid \mathit{domain} \in \mathsf{dom}(y)\}$ by ID^y.

We say that an ID is *governed* by the DY process to which the domain of the ID belongs. Formally, we define the mapping $\mathsf{governor}\colon \mathsf{ID} \to \mathcal{W}$, $\langle \mathit{name}, \mathit{domain}\rangle \mapsto \mathsf{dom}^{-1}(\mathit{domain})$.

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

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

Identities will also be used when an AS returns subject identifiers requested by a client instance. In this case, the AS returns the identity of the RO that logged in to the AS, which subsumes the different types of subject identifiers specified in [4] .

A.7 Corruption

Similar to [26], client instances, ASs, and RSs can become corrupted: If they receive the message `CORRUPT`, they start collecting all incoming messages in their state and (upon triggering) send out messages that are non-deterministically chosen from the set of all messages that are derivable from their state and collected input messages, just like the attacker process. We say that an AS, a client instance, or an RS is *honest* if the according part of their state ($s.\mathtt{corrupt}$) is $\perp$, and that they are corrupted otherwise.

A.8 Network Attackers

As mentioned, the network attacker na is modeled to be a network attacker as specified in [29]. As in [26], we allow it to listen to/spoof all available IP addresses, and hence, define $I^{na} = \mathsf{IPs}$. The initial state is $s_0^{na} = \langle \mathit{attdoms}, \mathit{tlskeys}, \mathit{keyproofkeys}\rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker na, *tlskeys* is a sequence of all domains and the corresponding public keys, and *keyproofkeys* is a sequence containing the public keys of all private keys in K_{KP} (i.e., all keys used for signatures or MTLS, but not the keys used for MACs, see Appendix A.11).

A.9 Browsers

Each $b \in \mathsf{B}$ is a web browser as defined in [29], with $I^b := \mathsf{addr}(b)$ being its addresses.

We define the initial state similar to [28]. We denote the set of all IDs of b by $\mathsf{ID}_b := \mathsf{ownerOfID}^{-1}(b)$. The set of passwords that a browser b gives to an origin o is defined as follows: If the origin belongs to an AS, then the user's passwords of this AS are contained in the set. To define this mapping in the initial state, we first define for some process p

$$\mathsf{Secrets}^{b,p} = \Big\{ s \;\Big|\; b = \mathsf{ownerOfSecret}(s) \land \\ \big(\exists\, i\colon s = \mathsf{secretOfID}(i) \land i \in \mathsf{ID}^p\big) \Big\}.$$

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

A.10 Helper Functions

In our modeling, the following key proof related helper functions are used, which can be used by all servers.

A.10.1 SIGN_AND_SEND. This algorithm inserts the headers used for key proofs into an HTTP request and then sends this request via the HTTPS_SIMPLE_SEND algorithm of the generic HTTPS server model from [29]. The algorithm simulates the HTTP Message Signature (`httpsig`) key proofing method [3]. We do not model the JWS-based methods (`jwsd` and `jws`) that are currently also supported by GNAP, because they differ primarily syntactically at the abstraction level of the WIM and the few semantic differences do not affect the security properties. ν_{n3} and ν_{n4} denote placeholders for nonces that are not used elsewhere by any of the processes that use this algorithm.

The input parameters are used as follows: *HTTPMethod* is the HTTP method that will be used to send the request. *url* is the URL to which the request will be sent. If *keyProof* $\equiv$ `sign`, *key* is used to sign the message. If *keyProof* $\not\equiv$ `sign` (e.g. `mac`), *key* is used to create a MAC for the message. *keyID* models the `kid` property of JSON Web Keys [41]. *body* is the body of the HTTP request. Through *authHeader* an `Authorization` header can be included in the request. If no `Authorization` header is to be used, *authHeader* must be $\bot$. *reference*, s', and a are required as input parameters for HTTPS_SIMPLE_SEND.

A.10.2 VALIDATE_KEY_PROOF. This algorithm can be used to validate key proofs. If the key proof is invalid, the algorithm stops, otherwise it returns.

The input parameters are used as follows: m is the HTTP request for which the key proof should be validated. *method* is the key proofing method. If *method* $\equiv$ `sign`, this algorithm simulates the validation of an HTTP Message Signature key proof using the public key *key*. If *method* $\equiv$ `mac`, this algorithm simulates the validation of an HTTP Message Signature key proof with a symmetric key *key*. In these two cases, *keyID* models the `kid` property of JSON Web Keys [41]. If *method* $\equiv$ `mtls`, this algorithm finishes an MTLS key proof by verifying the MTLS nonce sent by the requester[3] and checking that the key used matches *key*. *keyID* is ignored in this case. When validating signatures or MACs, the state s' is used to store nonces that are used as replay protection. This is because the security considerations of GNAP recommend using some form of replay protection for signatures/MACs [56].

[3] See Fett et al. [21] for an explanation of how MTLS is modeled within the WIM.

Algorithm A.4 Helper Functions: Signing and sending requests.

```
function SIGN_AND_SEND($HTTPMethod$, $url$, $keyID$, $key$, $keyProof$, $authHeader$, $body$, $reference$, $s'$, $a$)
    let $sigInput$ := [method:$HTTPMethod$, targetURI:$url$]
    let $sigParams$ := [covered:$\langle$method, targetURI$\rangle$, keyID:$keyID$, nonce:$\nu_{n3}$]
    if $body \not\equiv \langle\rangle$ then        ▷ If the message contains a body, its digest must be signed
        let $sigInput$[contentDigest] := hash($body$)
        let $sigParams$[covered] := $sigParams$[covered] $+^{\langle\rangle}$ contentDigest
    if $authHeader \not\equiv \bot$ then        ▷ If present, the AuthZ header must be covered by the signature
        let $sigInput$ := $sigInput$ $+^{\langle\rangle}$ $authHeader$
        let $sigParams$[covered] := $sigParams$[covered] $+^{\langle\rangle}$ authorization
    let $sigInput$[sigParams] := $sigParams$
    if $keyProof \equiv$ sign then
        let $signature$ := sig($sigInput$, $key$)
    else
        let $signature$ := mac($sigInput$, $key$)
    let $headers$ := [Signature-Input:$sigParams$, Signature:$signature$]
    if $body \not\equiv \langle\rangle$ then
        let $headers$[Digest] := hash($body$)
    if $authHeader \not\equiv \bot$ then
        let $headers$ := $headers$ $+^{\langle\rangle}$ $authHeader$
    let $req$ := $\langle$HTTPReq, $\nu_{n4}$, $HTTPMethod$, $url$.host, $url$.path, $\langle\rangle$, $headers$, $body\rangle$
    call HTTPS_SIMPLE_SEND($reference$, $req$, $s'$, $a$)
```

Algorithm A.5 Helper Functions: Validating key proofs.

```
function VALIDATE_KEY_PROOF($method, m, keyID, key, s'$)
    if $method \equiv \texttt{sign} \vee method \equiv \texttt{mac}$ then        ▷ HTTP Message Signature
        if $m.\text{body} \not\equiv \langle\rangle$ then
            let $digest := m.\text{headers}[\texttt{Digest}]$
            if $digest \not\equiv \text{hash}(m.\text{body})$ then stop
        let $sigParams := m.\text{headers}[\texttt{Signature-Input}]$
        let $signature := m.\text{headers}[\texttt{Signature}]$
        if $keyID \not\equiv sigParams[\texttt{keyID}]$ then stop
        let $covered := sigParams[\texttt{covered}]$
        if $\texttt{method} \not\in^{\langle\rangle} covered \vee \texttt{targetURI} \not\in^{\langle\rangle} covered \vee (m.\text{body} \not\equiv \langle\rangle \wedge \texttt{contentDigest} \not\in^{\langle\rangle} covered)$
          $\hookrightarrow$ $\vee (\texttt{Authorization} \in m.\text{headers} \wedge \texttt{authorization} \not\in^{\langle\rangle} covered)$ then stop ▷ Not all required parts covered by signature
        if $\text{nonce} \not\in sigParams \vee sigParams[\texttt{nonce}] \in^{\langle\rangle} s'.\texttt{sigNonces}$ then stop        ▷ Replay protection
        let $controlURL := \langle \text{URL}, \text{S}, m.\text{host}, m.\text{path}, m.\text{parameters}, \langle\rangle\rangle$
        let $controlInput := [\texttt{method}{:}m.\text{method}, \texttt{targetURI}{:}controlURL]$
        if $m.\text{body} \not\equiv \langle\rangle$ then
            let $controlInput[\texttt{contentDigest}] := \text{hash}(m.\text{body})$
        if $\texttt{Authorization} \in m.\text{headers}$ then
            let $controlInput := controlInput +^{\langle\rangle} \langle \texttt{Authorization}, m.\text{headers}[\texttt{Authorization}]\rangle$
        let $controlInput[\texttt{sigParams}] := sigParams$
        if $controlInput \not\equiv \text{extractmsg}(signature)$ then stop        ▷ The signature was not created for the correct input
        if $method \equiv \texttt{sign}$ then
            if $\text{checksig}(signature, key) \not\equiv \top$ then stop        ▷ Invalid signature
        else        ▷ $method \equiv \texttt{mac}$
            if $\text{checkmac}(signature, key) \not\equiv \top$ then stop        ▷ Invalid MAC
        let $s'.\texttt{sigNonces} := s'.\texttt{sigNonces} +^{\langle\rangle} sigParams[\texttt{nonce}]$
    else if $method \equiv \texttt{mtls}$ then
        let $mtlsNonce := m.\text{body}[\texttt{mtlsNonce}]$
        let $mtlsInfo$ such that $mtlsInfo \in^{\langle\rangle} s'.\texttt{mtlsRequests} \wedge mtlsInfo.1 \equiv mtlsNonce$ if possible; otherwise stop
        let $s'.\texttt{mtlsRequests} := s'.\texttt{mtlsRequests} - mtlsInfo.1$
        if $mtlsInfo.2 \not\equiv key$ then stop        ▷ Key used for MTLS does not match the key of the sender
    else stop        ▷ Unsupported method
    return $s'$
```

A.11 Client Instances

A client instance $c \in \mathsf{CI}$ is a web server modeled as an atomic DY process (I^c, Z^c, R^c, s_0^c) with the addresses $I^c := \mathsf{addr}(c)$.

In the client instance state, *key records* are used to store information about the keys required by the client instance for key proofs.

Definition A.3. A *key record* is a term of one of the following forms

- $\langle \mathtt{sign}, \mathit{keyID}, \mathit{key}, \mathit{instanceID} \rangle$
- $\langle \mathtt{mac}, \mathit{keyID}, \mathit{key}, \mathit{instanceID}, \mathit{rs} \rangle$
- $\langle \mathtt{mtls}, \mathit{key}, \mathit{instanceID} \rangle$

with $\mathit{keyID} \in \mathsf{KeyIDs}$, $\mathit{key} \in K_{\mathrm{KP}}$, $\mathit{instanceID} \in \mathbb{S} \cup \{\bot\}$, and $\mathit{rs} \in \mathsf{Doms}$.

For a key record r we use $r.\mathtt{method}$ as notation for $r.1$. If $\mathit{instanceID} \not\equiv \bot$, $\mathit{instanceID}$ is the instance identifier with which the client instance is registered with an AS using the key key and (if applicable) the key ID keyID. Otherwise, this key is used without an existing registration, i.e. the key is not known to the AS in advance. If $r.\mathtt{method} \equiv \mathtt{mac}$, $\mathit{instanceID}$ must not be $\bot$. This is because key is a symmetric key in this case and GNAP requires that symmetric keys can be dereferenced by the AS. Thus, to use a symmetric key, a client instance must already be registered with the AS. If symmetric keys are used, rs is the domain of the RS with which this key is additionally shared (besides the AS). We only allow client instances to use symmetric keys in combination with a specific AS and a specific RS, since using them with multiple RSs carries a high risk, since any of the RSs used could impersonate the client instance at the AS or other RSs.

Next, we define the set Z^c of states of c and the initial state s_0^c of c.

Definition A.4. A *state* $s \in Z^c$ *of client instance* c is a term of the form $\langle \mathit{DNSaddress}, \mathit{pendingDNS}, \mathit{corrupt}, \mathit{pendingRequests}, \mathit{keyMapping}, \mathit{tlskeys}, \mathit{keyRecords}, \mathit{authServers}, \mathit{resourceServers}, \mathit{sessions}, \mathit{grants}, \mathit{receivedValues}, \mathit{browserRequests} \rangle$ with $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{pendingRequests} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$, $\mathit{keyRecords} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{tokenKeys} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{authServers} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{resourceServers} \in \mathcal{T}_{\mathcal{N}}$,
$\mathit{sessions} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{grants} \in \left[\mathcal{N} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$, $\mathit{receivedValues} \in \left[\mathcal{N} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$, and $\mathit{browserRequests} \in \left[\mathcal{N} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$,

An *initial state* s_0^c *of* c is a state of c with $s_0^c.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^c.\mathtt{corrupt} \equiv \bot$, $s_0^c.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^c.\mathtt{keyMapping}$ being the same as the keymapping for browsers, $s_0^c.\mathtt{tlskeys} \equiv \mathit{tlskeys}^c$, $s_0^c.\mathtt{sessions} \equiv \langle\rangle$, $s_0^c.\mathtt{grants} \equiv \langle\rangle$, $s_0^c.\mathtt{receivedValues} \equiv \langle\rangle$, and $s_0^c.\mathtt{browserRequests} \equiv \langle\rangle$.

sessions will contain a dictionary that maps from session identifiers to information about that session. The session identifiers are nonces that are stored in the browser via the `Set-Cookie` header so that a particular browser session can be recognized in a further request by the same browser. *sessions* is used to store the subject identifiers received from the different ASs for the different browser instances that started grants using c. A subject identifier stored for a particular browser instance can be included in a new grant request to the same AS within the user field of the grant request. *sessions* is also used to store at which AS the browser instance is currently logged in with which identity. A corresponding service session ID is also stored in *sessions*.

grants will store various information about ongoing grants. The different grants are distinguished by a nonce called *grantID*, which acts as a key for the outer dictionary.

receivedValues will store the access tokens and subject identifiers received by c. The key for the outer dictionary is the *grantID* of the grant process in which the values were received. The values of the outer dictionary are dictionaries in which the access tokens and subject identifiers are stored under the keys `accessToken` and `subjectID`. An access token contains both the actual value of the access token in the form of a nonce and information needed to use the access token, such as which key the access token bound to, if any. Received access tokens can be used by c at any time when a trigger message is received.

browserRequests stores requests from browsers in order to be able to answer them at a later time. The key for the outer dictionary is the *grantID* of the grant process in which the requests were sent. The strings `startRequest` and `finishRequest` are used as keys for the inner dictionaries. The values under `startRequest` contain requests sent by browsers to start a grant request and the values under `finishRequest` contain requests sent after the RO has finished its interaction with the AS.

$s_0^c.\mathtt{authServers}$ is a non-empty sequence of domains representing the authorization servers c is configured to use. For all domains $d \in^{\langle\rangle} s_0^c.\mathtt{authServers}$ there must be an AS $\mathit{as} \in \mathsf{AS}$ with $d \in \mathsf{dom}(\mathit{as})$.

$s_0^c.\mathtt{resourceServers}$ is a non-empty sequence of domains representing the resource servers c is configured to use. For all domains $d \in^{\langle\rangle} s_0^c.\mathtt{resourceServers}$ there must be an RS $\mathit{rs} \in \mathsf{RS}$ with $d \in \mathsf{dom}(\mathit{rs})$.

$s_0^c.\mathtt{keyRecords}$ is a non-empty dictionary mapping domains of ASs to sequences of key records. For all domains $d \in s_0^c.\mathtt{keyRecords}$ it must hold that $d \in^{\langle\rangle} s_0^c.\mathtt{authServers}$. The sequence $s_0^c.\mathtt{keyRecords}[d]$ then contains the key records c will use when interacting with the AS $\mathit{as} = \mathsf{dom}^{-1}(d)$ when using the domain d. The values of $s_0^c.\mathtt{keyRecords}$ must be non-empty, so there must be at least one key record for each domain. $s_0^c.\mathtt{keyRecords}$ must contain a value for each $d \in^{\langle\rangle} s_0^c.\mathtt{authServers}$. We collect all key records in $s_0^c.\mathtt{keyRecords}$ in $R := \left\langle \bigcup_{d \in s_0^c.\mathtt{keyRecords}} \bigcup_{r \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[d]} r \right\rangle$. For any two distinct key records $r, r' \in^{\langle\rangle} R$ it must hold that $r.\mathtt{key} \not\equiv r'.\mathtt{key}$. If $r.\mathtt{method} \in \{\mathtt{sign}, \mathtt{mac}\} \wedge r'.\mathtt{method} \in \{\mathtt{sign}, \mathtt{mac}\}$, it must hold that $r.\mathtt{keyID} \not\equiv r'.\mathtt{keyID}$. For all key records $r \in^{\langle\rangle} R$ with $r.\mathtt{method} \equiv \mathtt{mac}$ it must hold that $r.\mathtt{rs} \in^{\langle\rangle} s_0^c.\mathtt{resourceServers}$. Given a domain $d \in s_0^c.\mathtt{keyRecords}$, it must hold for any two distinct key records $r, r' \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[d]$ that $r.\mathtt{instanceID} \not\equiv r'.\mathtt{instanceID}$. For two distinct domains $d, d' \in s_0^c.\mathtt{keyRecords}$, which both belong to the same AS as (i.e., $d \in \mathsf{dom}(\mathit{as}) \wedge d' \in \mathsf{dom}(\mathit{as})$), it must hold for all key records $r \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[d]$ and all key records $r' \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[d']$ that $r.\mathtt{instanceID} \not\equiv r'.\mathtt{instanceID}$. For all processes $p \neq c$ it must hold that the key

$r.\texttt{key}$ of each key record $r \in^{\langle\rangle} R$ with $r.\texttt{method} \in \{\texttt{sign}, \texttt{mtls}\}$ appears only as a public key in s_0^p. If $r.\texttt{method} \equiv \texttt{mac}$, $r.\texttt{key}$ must only be initially stored in c, $\mathsf{dom}^{-1}(r.\texttt{rs})$, and the AS *as* that has the domain d in $\mathsf{dom}(as)$ for which $r \in^{\langle\rangle} s_0^c.\texttt{keyRecords}[d]$.

$s_0^c.\texttt{tokenKeys}$ is a non-empty dictionary mapping domains of ASs to sequences of key records. This dictionary stores keys that can be used by an AS to bind an access token to a key different than the client instance's key. For all domains $d \in s_0^c.\texttt{tokenKeys}$ it must hold that $d \in^{\langle\rangle} s_0^c.\texttt{authServers}$. We further require that those keys are all distinct, in particular for different ASs. Formally we have, for all domains d and d' in $s_0^c.\texttt{tokenKeys}$ and key records $k \in^{\langle\rangle} s_0^c.\texttt{tokenKeys}[d]$ and $k' \in^{\langle\rangle} s_0^c.\texttt{tokenKeys}[d']$ with $k \not\equiv k'$ that $k.\texttt{key} \not\equiv k'.\texttt{key}$. The keys stored in this field should only be known to the CI c, i.e., for every domain $d \in s_0^c.\texttt{tokenKeys}$, every key record $r \in^{\langle\rangle} s_0^c.\texttt{tokenKeys}[d]$, and every processes $p \neq c$ it must hold that the key $r.\texttt{key}$ appears only as a public key in s_0^p. We only allow `sign` and `mtls` for the methods of the key records in $s_0^c.\texttt{tokenKeys}$. We do not allow the use of MACs here, since the symmetric key would then have to be transmitted to the resource servers for MAC validation during token introspection. Since symmetric keys should not be transmitted via token introspection, it is unclear at the time of this work how this could be implemented. We started a discussion [35] to clarify this with the editors of the GNAP specification, which is still ongoing at the time this work was completed.

We now specify the relation R^c: This relation is based on the generic HTTPS server model defined in [29]. Hence, we only need to specify algorithms that differ from or do not exist in the generic server model. These algorithms are defined in Algorithms A.6–A.11. Note that in several places throughout these algorithms we use placeholders to generate "fresh" nonces. Table 2 shows a list of all placeholders used.

	Usage
ν_1	new grant ID
ν_2	nonce to generate a unique interaction finish URL
ν_3	nonce for the calculation of the interaction finish hash
ν_4	new session identifier for the browser
ν_5	new HTTP request nonce
ν_6	new HTTP request nonce
ν_7	new grant ID
ν_8	new HTTP request nonce
ν_9	new HTTP request nonce
ν_{10}	new HTTP request nonce
ν_{11}	new service session identifier

Table 2: List of placeholders used in the client instance algorithms.

The script that is used by the client instance is described in Algorithm A.12. In this script, to extract the current URL of a document, the function $\mathsf{GETURL}(\mathit{tree}, \mathit{docnonce})$ is used which is also defined in [29].

The following algorithms are used for modeling the client instances:

- Algorithm A.6 processes requests to the client instance. A browser can obtain the index page of c by sending a GET request m to c with $m.\texttt{path} \equiv \texttt{/}$. The index page contains a script that sends a request to the `/startGrantRequest` path, which causes c to send a grant request to an AS chosen by the browser if c is configured to use this AS. Furthermore, the algorithm accepts requests sent as part of the interaction finish modes. These originate from a browser when using the redirect interaction finish mode (`/finish`) or from an AS when using the push interaction finish mode (`/push`). c can also accept a request for data from a browser (`/getData`). This is used for the push interaction finish mode. While in the redirect interaction finish mode the browser receives the resource and/or the service session ID as a response to the redirect by the AS, this is not possible when using the push interaction finish mode, since here the AS informs the client instance about the completion of the interaction and not the browser. Therefore, in the push interaction finish mode, we let the AS redirect the browser to the `/getData` endpoint to return the data received from the AS in response to this redirect. For this, we use the nonce from the interaction finish URL so that c can uniquely associate the request with the associated grant. A request to `/logout` allows a browser instance to log out from the client instance. For this, c creates a new session identifier and returns it as a cookie. If data was stored for an old session identifier, it will be deleted.
- Algorithm A.7 processes responses to the client instance. These can be grant responses from ASs or resource responses from RSs. Additionally, responses from ASs or RSs can be processed for the purpose of modeling MTLS. If a grant response contains a subject identifier, this is stored in $s'.\texttt{sessions}$ under the corresponding session ID and the domain of the AS used, so that it can be specified in the user field in further grant requests in the same browser session and to the same AS. If a grant response contains an access token, it is stored in $s'.\texttt{receivedValues}$ so that it can be used later in requests to resource servers. We do not use received access tokens directly, because continuation requests can also be sent to the AS in response to a grant response, so we would possibly have to emit both a resource request to an RS and a continuation request to an AS in one processing step. This would unnecessarily complicate the algorithms used for sending these requests, such as $\mathsf{SIGN\_AND\_SEND}$. Moreover, this modeling is closer to reality, since an access token can be used at any time (as long as it has not been revoked).
- Algorithm A.8 non-deterministically does one of two things. Either it is used to enable a client instance to send a grant request to an AS without the presence of an end user (software-only authorization), or the client instance uses a received access token to request a resource. In the first case, the client instance sends a grant request to a non-deterministically chosen AS with which it is registered. Since no interaction can take place without an end user, neither an `interaction` entry nor a `finish` entry is transmitted in the *grantRequest* dictionaries for these grant requests. In our modeling, we only allow client instances that are already registered with the AS to use software-only authorization, since otherwise

client instances unknown to the AS could also request access to resources protected by the AS. Thus, arbitrary client instances could access resources, which makes these resources irrelevant for a security analysis. In the second case, the client instance non-deterministically chooses one of the received access tokens and one of the RSs from the *resourceServers* subterm. Then it sends a resource request to the chosen resource server using the chosen access token. If the access token is bound to a symmetric key, the RS is not chosen non-deterministically, but the RS specified in the key record of that key is used. If only a subject identifier and no access token was requested, the associated service session identifier is returned to the browser directly, since a resource does not have to be requested from an RS first.

- Algorithm A.9 is used to non-deterministically select the information requested by the client instance from an AS. A client instance can request an access token and/or a subject identifier. A subject identifier can only be requested when an end user is present, i.e., not when a grant request has been triggered by a trigger message to the client instance and thus software-only authorization is used. If an access token is requested, it can be either a bearer access token or a key-bound access token. We do not allow a client instance to request neither an access token nor a subject identifier, since in this case the client instance would have no reason to send the grant request.
- Algorithm A.10 comes into play after the interaction with the RO is completed by one of the interaction finish modes. First, the transmitted hash value is checked. If successful, a continuation request is sent to the AS including the interaction reference. The client instance may decide to change the values requested from the AS. If this is the case, the continuation request is an HTTP PATCH request and Algorithm A.9 is called again, otherwise it is an HTTP POST request. The key proofing method used corresponds to that of the corresponding grant request.
- Algorithm A.11 is used to answer the browser's request after the interaction is finished. If an access token was requested with the grant request, a resource received from an RS is returned in the body of the response. If a subject identifier was requested with the grant request, this subject identifier is used as the subject identifier under which the browser instance is currently logged in and the associated service session ID is returned within the `Set-Cookie` header.

Algorithm A.6 Relation of a Client Instance R^c: Processing HTTPS requests.

```
function PROCESS_HTTPS_REQUEST($m, k, a, f, s'$)
        ▷ **Process an incoming HTTPS request.** $m$ is the incoming message, $k$ is the encryption key for the response, $a$ is the receiver, $f$ the sender of the message. $s'$ is the current state of the atomic DY process $c$.
    if $m$.path ≡ / ∧ $m$.method ≡ GET then                                          ▷ Serve index page
        let $m' := \mathsf{enc_s}(\langle$HTTPResp, $m$.nonce, 200, ⟨⟩, ⟨script_ci_index, ⟨⟩⟩⟩, $k)$      ▷ Send *script_ci_index* in HTTP response.
        stop ⟨⟨$f, a, m'$⟩⟩, $s'$
    else if $m$.path ≡ /startGrantRequest ∧ $m$.method ≡ POST then                  ▷ Start a new grant request
        let $domainAS$ := $m$.body                                   ▷ Domain of the AS to send the grant request to
        if $domainAS \not\in^{\langle\rangle} s'$.authServers then stop          ▷ $c$ is not configured to use this AS
        let $endpoint$ := ⟨URL, S, $domainAS$, /requestGrant, ⟨⟩⟩                   ▷ Endpoint for the grant request
        let $grantID := v_1$                                                  ▷ Identifier for this grant request
        let $inquiredValues$ := GENERATE_INQUIRED_VALUES(⊤)
        let $finishMode$ ← {redirect, push}                      ▷ Non-det. select the used interaction finish mode
        let $finishURLnonce := v_2$
        let $CIfinishNonce := v_3$
        if $finishMode$ ≡ redirect then
            let $finishURL$ := ⟨URL, S, $m$.host, /finish, [request:$finishURLnonce$]⟩
        else
            let $finishURL$ := ⟨URL, S, $m$.host, /push, [request:$finishURLnonce$]⟩
        let $finish$ := [finishMode:$finishMode$, finishURL:$finishURL$, nonce:$CIfinishNonce$]
        let $grantRequest$ := [inquiredValues:$inquiredValues$, finish:$finish$, interaction:⊤]      ▷ Interaction with RO possible
        let $s'$.browserRequests[$grantID$][startRequest] := ⟨$k, a, f, m$.nonce⟩
        if ⟨__Host, sessionID⟩ ∈ $m$.headers[Cookie] then                         ▷ The browser sent a session ID
            let $sessionID$ := $m$.headers[Cookie][⟨__Host, sessionID⟩]
            if $domainAS \in s'$.sessions[$sessionID$] then        ▷ Check if a subject identifier is stored for this session ID and this AS
                let $grantRequest$[user] := $s'$.sessions[$sessionID$][$domainAS$]      ▷ Include previously received subject identifier
        else
            let $sessionID := v_4$
        let $keyRecord$ ← $s'$.keyRecords[$domainAS$]                         ▷ Non-det. select key record for this AS
        let $s'$.grants[$grantID$] := [AS:$domainAS$, sessionID:$sessionID$, keyRecord:$keyRecord$,
          ↪ finishURLnonce:$finishURLnonce$, CIfinishNonce:$CIfinishNonce$, requested:$inquiredValues$]
        if $keyRecord$.instanceID ≢ ⊥ then                                      ▷ key is registered at the AS used
            let $grantRequest$[instanceID] := $keyRecord$.instanceID
            let $s'$.grants[$grantID$][request] := $grantRequest$
            if $keyRecord$.method ≢ mtls then
                let $reference$ := [responseTo:grantResponse, grantID:$grantID$, sentTo:$domainAS$]
                call SIGN_AND_SEND(POST, $endpoint$, $keyRecord$.keyID, $keyRecord$.key, $keyRecord$.method,
                  ↪ ⊥, $grantRequest$, $reference$, $s'$, $a$)
            else
                let $body$ := [instanceID:$keyRecord$.instanceID]
                let $message$ := ⟨HTTPReq, $v_5$, POST, $domainAS$, /MTLS-prepare, ⟨⟩, ⟨⟩, $body$⟩
                let $reference$ := [responseTo:MTLS_GR, grantID:$grantID$]
                call HTTPS_SIMPLE_SEND($reference$, $message$, $s'$, $a$)
```

This algorithm is continued on the next page.

Continuation of Algorithm A.6 (Client PROCESS_HTTPS_REQUEST)

```
    else                                                          ▷ $c$ is not registered with the AS used
        let $key$ := $keyRecord$.key
        if $keyRecord$.method $\equiv$ sign then
            let $keyID$ := $keyRecord$.keyID
            let $grantRequest$[client] := [keyID:$keyID$, key:pub($key$), method:sign]
            let $s'$.grants[$grantID$][request] := $grantRequest$
            let $reference$ := [responseTo:grantResponse, grantID:$grantID$, sentTo:$domainAS$]
            call SIGN_AND_SEND(POST, $endpoint$, $keyID$, $key$, sign, $\perp$, $grantRequest$, $reference$, $s'$, $a$)
        else                                                      ▷ MTLS is used as key proofing method
            let $grantRequest$[client] := [key:pub($key$), method:mtls]
            let $s'$.grants[$grantID$][request] := $grantRequest$
            let $body$ := [publicKey:pub($key$)]
            let $message$ := $\langle$HTTPReq, $v_5$, POST, $domainAS$, /MTLS-prepare, $\langle\rangle$, $\langle\rangle$, $body\rangle$
            let $reference$ := [responseTo:MTLS_GR, grantID:$grantID$]
            call HTTPS_SIMPLE_SEND($reference$, $message$, $s'$, $a$)
else if $m$.path $\equiv$ /finish $\wedge$ $m$.method $\equiv$ GET then             ▷ Redirect interaction finish mode
    let $finishURLnonce$ := $m$.parameters[request]
    let $grantID$ such that $s'$.grants[$grantID$][finishURLnonce] $\equiv$ $finishURLnonce$ if possible; otherwise stop
    if $s'$.grants[$grantID$][request][finish][finishMode] $\not\equiv$ redirect then  stop   ▷ Wrong interaction finish mode was used
    if $m$.headers[Cookie][$\langle$__Host, sessionID$\rangle$]$\not\equiv$ $s'$.grants[$grantID$][sessionID] then
        stop                          ▷ Browsers session identifier does not match the one from the grant request
    let $s'$.browserRequests[$grantID$][finishRequest] := $\langle k, a, f, m$.nonce$\rangle$
    let $interactRef$ := $m$.parameters[interactRef]
    let $hash$ := $m$.parameters[hash]
    call SEND_CONTINUATION_REQUEST($grantID$, $interactRef$, $hash$, $s'$, $a$)
else if $m$.path $\equiv$ /push $\wedge$ $m$.method $\equiv$ POST then                 ▷ Push interaction finish mode
    let $finishURLnonce$ := $m$.parameters[request]
    let $grantID$ such that $s'$.grants[$grantID$][finishURLnonce] $\equiv$ $finishURLnonce$ if possible; otherwise stop
    if $s'$.grants[$grantID$][request][finish][finishMode] $\not\equiv$ push then  stop      ▷ Wrong interaction finish mode was used
    let $interactRef$ := $m$.body[interactRef]
    let $hash$ := $m$.body[hash]
    call SEND_CONTINUATION_REQUEST($grantID$, $interactRef$, $hash$, $s'$, $a$)
else if $m$.path $\equiv$ /getData $\wedge$ $m$.method $\equiv$ GET then
    let $finishURLnonce$ := $m$.parameters[request]
    let $grantID$ such that $s'$.grants[$grantID$][finishURLnonce] $\equiv$ $finishURLnonce$ if possible; otherwise stop
    if $s'$.grants[$grantID$][request][finish][finishMode] $\not\equiv$ push then
        stop                    ▷ This endpoint is only used when the push interaction finish mode is used
    if $m$.headers[Cookie][$\langle$__Host, sessionID$\rangle$] $\not\equiv$ $s'$.grants[$grantID$][sessionID] then
        stop                          ▷ Browsers session identifier does not match the one from the grant request
    let $s'$.browserRequests[$grantID$][finishRequest] := $\langle k, a, f, m$.nonce$\rangle$
    if domainFirstRS $\in$ $s'$.grants[$grantID$] $\wedge$ $s'$.grants[$grantID$][domainFirstRS] $\in$ $s'$.grants[$grantID$][resources] then
        ▷ Resource has already been received from the RS
        call SEND_RESPONSE_TO_BROWSER($grantID$, $s'$)
    else stop $\langle\rangle$, $s'$
else if $m$.path $\equiv$ /logout $\wedge$ $m$.method $\equiv$ POST then
        ▷ Set a new session ID so that the browser instance can log in with a different identity at already used ASs
    if $\langle$__Host, sessionID$\rangle$ $\in$ $m$.headers[Cookie] then                        ▷ The browser sent a session ID
        let $oldSessionID$ := $m$.headers[Cookie][$\langle$__Host, sessionID$\rangle$]
        if $oldSessionID$ $\in$ $s'$.sessions then
            let $s'$.sessions := $s'$.sessions $-$ $oldSessionID$                          ▷ Delete data of the old session
    let $headers$ := [Set-Cookie:$\langle\langle\langle$__Host, sessionID$\rangle$, $\langle v_4, \top, \top, \top\rangle\rangle\rangle$]
    let $m'$ := $\mathsf{enc_s}$(HTTPResp, $m$.nonce, 200, $headers$, $\langle$script_ci_index, $\langle\rangle\rangle$, $k$)
    stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
else stop                                                                             ▷ Unsupported operation
```

Algorithm A.7 Relation of a Client Instance R^c: Processing HTTPS responses.

```
function PROCESS_HTTPS_RESPONSE($m$, $reference$, $request$, $a$, $f$, $s'$)
    let $grantID := reference[\texttt{grantID}]$
    let $grantRequest := s'.\texttt{grants}[grantID]$
    let $domainAS := grantRequest[\texttt{AS}]$
    let $sessionID := grantRequest[\texttt{sessionID}]$
    let $keyRecord := grantRequest[\texttt{keyRecord}]$
    if $reference[\texttt{responseTo}] \equiv \texttt{grantResponse}$ then
        let $grantResponse := m.\texttt{body}$
        if $\texttt{instanceID} \in grantResponse$ then                                  ▷ The AS has registered $c$
            if $keyRecord.\texttt{instanceID} \equiv \perp$ then                           ▷ $c$ was not already registered
                let $\bar{i} \leftarrow \mathbb{N}$ such that $s'.\texttt{keyRecords}[domainAS].\bar{i} \equiv keyRecord$
                let $s'.\texttt{keyRecords}[domainAS].\bar{i}.\texttt{instanceID} := grantResponse[\texttt{instanceID}]$
        if $\texttt{subjectID} \in grantResponse$ then
            if $\texttt{subjectID} \notin^{\langle\rangle} grantRequest[\texttt{requested}]$ then stop     ▷ AS returned subject identifier that was not requested
            let $subjectID := grantResponse[\texttt{subjectID}]$
            let $domainAS' := reference[\texttt{sentTo}]$          ▷ Store subject identifier of this end user at the AS we sent the request to
            let $s'.\texttt{sessions}[sessionID][domainAS'] := subjectID$
            let $s'.\texttt{receivedValues}[grantID][\texttt{subjectID}] := \langle subjectID, domainAS' \rangle$
        if $\texttt{accessToken} \in grantResponse$ then
            let $AT := [grantResponse][\texttt{accessToken}]$
            if $\texttt{accessToken} \notin^{\langle\rangle} grantRequest[\texttt{requested}] \vee \texttt{bearerToken} \notin^{\langle\rangle} grantRequest[\texttt{requested}]$ then
                stop                                                              ▷ No (bearer) access token was requested
            if $\texttt{accessToken} \in^{\langle\rangle} grantRequest[\texttt{requested}] \wedge AT[\texttt{flags}] \equiv \texttt{bearer}$ then
                stop                                              ▷ Access token was requested, but bearer token was issued
            if $\texttt{bearerToken} \in^{\langle\rangle} grantRequest[\texttt{requested}] \wedge AT[\texttt{flags}] \not\equiv \texttt{bearer}$ then
                stop                                              ▷ Bearer token was requested, but access token was issued
            if $\texttt{key} \in^{\langle\rangle} AT \wedge \nexists kr \in^{\langle\rangle} s'.\texttt{tokenKeys}[domainAS] : \mathsf{pub}(kr.\texttt{key}) \equiv AT[\texttt{key}][\texttt{pubKey}]$ then
                stop                                              ▷ The key sent by AS for the token does not belong to AS
            let $s'.\texttt{receivedValues}[grantID][\texttt{accessToken}] := grantResponse[\texttt{accessToken}]$
        if $\texttt{interact} \in grantResponse$ then                                          ▷ Interaction is required
            if $\texttt{finishedInteraction} \in grantRequest$ then stop                 ▷ Interaction has already been completed
            if $\texttt{interaction} \notin grantRequest[\texttt{request}]$ then stop ▷ Interaction is required, but $c$ did not indicate support for interaction
            if $\texttt{continue} \notin grantResponse$ then stop          ▷ $c$ needs to be allowed to continue once the interaction is finished
            let $s'.\texttt{grants}[grantID][\texttt{continueAT}] := grantResponse[\texttt{continue}][\texttt{accessToken}]$
            let $s'.\texttt{grants}[grantID][\texttt{continueURL}] := grantResponse[\texttt{continue}][\texttt{url}]$
            let $s'.\texttt{grants}[grantID][\texttt{ASfinishNonce}] := grantResponse[\texttt{interact}][\texttt{finish}]$
            let $\langle key, receiver, sender, nonce \rangle := s'.\texttt{browserRequests}[grantID][\texttt{startRequest}]$
            let $cookies := \langle\langle\langle \texttt{\_\_Host}, \texttt{sessionID} \rangle, \langle sessionID, \top, \top, \top \rangle\rangle\rangle$
            let $startMode \leftarrow \{\texttt{redirect}, \texttt{userCode}\}$
            if $startMode \equiv \texttt{redirect}$ then
                let $redirectURL := grantResponse[\texttt{interact}][\texttt{redirect}]$
                let $s'.\texttt{grants}[grantID][\texttt{redirectNonce}] := redirectURL.\texttt{parameters}[\texttt{request}]$
                let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, nonce, \texttt{303}, [\texttt{Location}{:}redirectURL, \texttt{Set-Cookie}{:}cookies], \langle\rangle \rangle, key)$
                stop $\langle\langle sender, receiver, m' \rangle\rangle, s'$
            else                                                                  ▷ user code interaction start mode
                let $url := \langle \texttt{URL}, \texttt{S}, domainAS, \texttt{/interactUC}, \langle\rangle \rangle$
                let $userCode := grantResponse[\texttt{interact}][\texttt{userCode}]$
                let $s'.\texttt{grants}[grantID][\texttt{userCode}] := userCode$
                let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, nonce, \texttt{200}, [\texttt{Set-Cookie}{:}cookies], [\texttt{userCodeUrl}{:}url, \texttt{userCode}{:}userCode] \rangle, key)$
                stop $\langle\langle sender, receiver, m' \rangle\rangle, s'$
    This algorithm is continued on the next page.
```

Continuation of Algorithm A.7 (Client PROCESS_HTTPS_RESPONSE)

```
        else if continue $\in$ grantResponse then          ▷ $c$ can continue and no interaction is required
            let continue $\leftarrow \{\top, \bot\}$          ▷ Non-det. decide whether to continue
            if continue $\equiv \top$ then          ▷ Request values again using a PATCH request
                if interaction $\in$ grantRequest[request] then
                    let inquiredValues := GENERATE_INQUIRED_VALUES($\top$)
                else
                    let inquiredValues := GENERATE_INQUIRED_VALUES($\bot$)
                let $s'$.grants[grantID][requested] := inquiredValues
                let continueAT := grantResponse[continue][accessToken]
                let continueURL := grantResponse[continue][url]
                let authHeader := $\langle$Authorization, $\langle$GNAP, continueAT$\rangle\rangle$
                let body := [inquiredValues:inquiredValues]
                if keyRecord.method $\not\equiv$ mtls then
                    let ref := [responseTo:grantResponse, grantID:grantID, sentTo:continueURL.host]
                    call SIGN_AND_SEND(PATCH, continueURL, keyRecord.keyID, keyRecord.key,
                      ↪ keyRecord.method, authHeader, body, ref, $s'$, $a$)
                else          ▷ MTLS is used as key proofing method
                    let $s'$.grants[grantID][patchRequest] := $\langle$authHeader, body, continueURL$\rangle$
                    if keyRecord.instanceID $\not\equiv \bot$ then
                        let body := [instanceID:keyRecord.instanceID]
                    else
                        let body := [publicKey:pub(keyRecord.key)]
                    let message := $\langle$HTTPReq, $v_6$, POST, continueURL.host, /MTLS-prepare, $\langle\rangle$, $\langle\rangle$, body$\rangle$
                    let ref := [responseTo:MTLS_PR, grantID:grantID]
                    call HTTPS_SIMPLE_SEND(ref, message, $s'$, $a$)
        else stop          ▷ AS rejected request without possibility to continue or grant response is invalid
    else if reference[responseTo] $\equiv$ resourceResponse then
        let domainRS := reference[domainRS]
        let $s'$.grants[grantID][resources][domainRS] := $m$.body
        if finishRequest $\in s'$.browserRequests[grantID] $\wedge$ domainRS $\equiv s'$.grants[grantID][domainFirstRS] then
            ▷ Browser awaits response and the resource of the first used RS was obtained
            call SEND_RESPONSE_TO_BROWSER(grantID, $s'$)
        else stop $\langle\rangle, s'$
    else if reference[responseTo] $\equiv$ MTLS_GR then          ▷ A new grant request is to be sent
        let $m_{\text{dec}}$ := $\text{dec}_{\text{a}}$($m$.body, keyRecord.key)
        let mtlsNonce, pubKey such that $\langle$mtlsNonce, pubKey$\rangle \equiv m_{\text{dec}}$ if possible; otherwise stop
        if pubKey $\equiv s'$.keyMapping[request.host] then
            let body := grantRequest[request]
            let body[mtlsNonce] := mtlsNonce
            let req := $\langle$HTTPReq, $v_6$, POST, domainAS, /requestGrant, $\langle\rangle$, $\langle\rangle$, body$\rangle$
            let ref := [responseTo:grantResponse, grantID:grantID, sentTo:domainAS]
            call HTTPS_SIMPLE_SEND(ref, req, $s'$, $a$)
        else stop          ▷ Send nonce only to the process that created it
```

This algorithm is continued on the next page.

Continuation of Algorithm A.7 (Client PROCESS_HTTPS_RESPONSE)

```
  else if $reference[\mathtt{responseTo}] \equiv \mathtt{MTLS\_CR}$ then    ▷ A new continuation request is to be sent
      let $m_{\mathrm{dec}} := \mathsf{dec_a}(m.\mathsf{body}, keyRecord.\mathsf{key})$
      let $mtlsNonce, pubKey$ such that $\langle mtlsNonce, pubKey\rangle \equiv m_{\mathrm{dec}}$ if possible; otherwise stop
      if $pubKey \equiv s'.\mathsf{keyMapping}[request.\mathsf{host}]$ then
          let $authHeader := \langle \mathtt{Authorization}, \langle \mathtt{GNAP}, s'.\mathtt{grants}[grantID][\mathtt{continueAT}]\rangle\rangle$
          let $interactRef := s'.\mathtt{grants}[grantID][\mathtt{interactRef}]$
          let $url := s'.\mathtt{grants}[grantID][\mathtt{continueURL}]$
          let $ref := [\mathtt{responseTo}{:}\mathtt{grantResponse}, \mathtt{grantID}{:}grantID, \mathtt{sentTo}{:}url.\mathtt{host}]$
          if $\mathtt{adjustedInquiredValues} \notin s'.\mathtt{grants}[grantID]$ then
              let $body := [\mathtt{interactRef}{:}interactRef, \mathtt{mtlsNonce}{:}mtlsNonce]$
              let $req := \langle \mathtt{HTTPReq}, v_6, \mathtt{POST}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, \langle authHeader\rangle, body\rangle$
          else
              let $inquiredValues := s'.\mathtt{grants}[grantID][\mathtt{requested}]$
              let $body := [\mathtt{interactRef}{:}interactRef, \mathtt{inquiredValues}{:}inquiredValues, \mathtt{mtlsNonce}{:}mtlsNonce]$
              let $req := \langle \mathtt{HTTPReq}, v_6, \mathtt{PATCH}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, \langle authHeader\rangle, body\rangle$
          call HTTPS_SIMPLE_SEND$(ref, req, s', a)$
      else stop    ▷ Send nonce only to the process that created it
  else if $reference[\mathtt{responseTo}] \equiv \mathtt{MTLS\_PR}$ then    ▷ $c$ modifies its grant request
      let $m_{\mathrm{dec}} := \mathsf{dec_a}(m.\mathsf{body}, keyRecord.\mathsf{key})$
      let $mtlsNonce, pubKey$ such that $\langle mtlsNonce, pubKey\rangle \equiv m_{\mathrm{dec}}$ if possible; otherwise stop
      if $pubKey \equiv s'.\mathsf{keyMapping}[request.\mathsf{host}]$ then
          let $\langle authHeader, body, url\rangle := s'.\mathtt{grants}[grantID][\mathtt{patchRequest}]$
          let $body[\mathtt{mtlsNonce}] := mtlsNonce$
          let $req := \langle \mathtt{HTTPReq}, v_6, \mathtt{PATCH}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, \langle authHeader\rangle, body\rangle$
          let $ref := [\mathtt{responseTo}{:}\mathtt{grantResponse}, \mathtt{grantID}{:}grantID, \mathtt{sentTo}{:}url.\mathtt{host}]$
          call HTTPS_SIMPLE_SEND$(ref, req, s', a)$
      else stop    ▷ Send nonce only to the process that created it
  else if $reference[\mathtt{responseTo}] \equiv \mathtt{MTLS\_RR}$ then    ▷ A new resource request is to be sent
      if $\mathtt{key} \in reference$ then    ▷ Access token is bound to its own key
          let $m_{\mathrm{dec}} := \mathsf{dec_a}(m.\mathsf{body}, reference[\mathtt{key}])$
      else    ▷ Access token is bound to key from key record
          let $m_{\mathrm{dec}} := \mathsf{dec_a}(m.\mathsf{body}, keyRecord.\mathsf{key})$
      let $mtlsNonce, pubKey$ such that $\langle mtlsNonce, pubKey\rangle \equiv m_{\mathrm{dec}}$ if possible; otherwise stop
      if $pubKey \equiv s'.\mathsf{keyMapping}[request.\mathsf{host}]$ then
          let $ref := reference[\mathtt{reference}]$
          let $req := reference[\mathtt{request}]$
          let $req.\mathsf{body} := [\mathtt{mtlsNonce}{:}mtlsNonce]$
          call HTTPS_SIMPLE_SEND$(ref, req, s', a)$
      else stop    ▷ Send nonce only to the process that created it
```

Algorithm A.8 Relation of a Client Instance R^c: Processing trigger messages.

```
function PROCESS_TRIGGER($s'$)
    let $startGrantRequest \leftarrow \{\top, \bot\}$
    if $startGrantRequest \equiv \top$ then                                        ▷ Start software-only authorization
        let $domainAS, \bar{i}$ such that $s'.\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{instanceID} \not\equiv \bot$ if possible; otherwise stop
        let $keyRecord := s'.\mathtt{keyRecords}[domainAS].\bar{i}$
        let $inquiredValues := \mathsf{GENERATE\_INQUIRED\_VALUES}(\bot)$
        let $grantID := \nu_7$                                                     ▷ Identifier for this grant request
        let $instanceID := keyRecord.\mathtt{instanceID}$
        let $grantRequest := [\mathtt{inquiredValues}{:}inquiredValues, \mathtt{instanceID}{:}instanceID]$
        let $s'.\mathtt{grants}[grantID] := [\mathtt{request}{:}grantRequest, \mathtt{AS}{:}domainAS, \mathtt{keyRecord}{:}keyRecord, \mathtt{requested}{:}inquiredValues]$
        if $keyRecord.\mathtt{method} \not\equiv \mathtt{mtls}$ then
            let $endpoint := \langle \mathsf{URL}, \mathsf{S}, domainAS, /\mathsf{requestGrant}, \langle\rangle\rangle$
            let $reference := [\mathsf{responseTo}{:}\mathsf{grantResponse}, \mathsf{grantID}{:}grantID, \mathsf{sentTo}{:}domainAS]$
            call SIGN_AND_SEND($\mathsf{POST}$, $endpoint$, $keyRecord.\mathtt{keyID}$, $keyRecord.\mathtt{key}$, $keyRecord.\mathtt{method}$,
              $\hookrightarrow$ $\bot$, $grantRequest$, $reference$, $s'$, $a$)
        else
            let $body := [\mathtt{instanceID}{:}instanceID]$
            let $message := \langle \mathsf{HTTPReq}, \nu_8, \mathsf{POST}, domainAS, /\mathsf{MTLS\text{-}prepare}, \langle\rangle, \langle\rangle, body\rangle$
            let $reference := [\mathsf{responseTo}{:}\mathsf{MTLS\_GR}, \mathsf{grantID}{:}grantID]$
            call HTTPS_SIMPLE_SEND($reference$, $message$, $s'$, $a$)
    else                                                        ▷ Use received access token and/or subject identifier
        let $grantID$ such that $grantID \in s'.\mathtt{receivedValues}$ if possible; otherwise stop
        let $originallyInqValues := s'.\mathtt{grants}[grantID][\mathtt{request}][\mathtt{inquiredValues}]$
        if $\mathsf{accessToken} \notin^{\langle\rangle} originallyInqValues \wedge \mathsf{bearerToken} \notin^{\langle\rangle} originallyInqValues$
          $\hookrightarrow$ $\wedge\, \mathtt{finishRequest} \in s'.\mathtt{browserRequests}[grantID]$ then
            ▷ Originally only a subject identifier was requested and browser is not yet logged in
            call SEND_RESPONSE_TO_BROWSER($grantID$, $s'$)
        else          ▷ An access token was requested. If applicable, the response to the browser is sent once the resource is received.
            if $\mathsf{accessToken} \notin s'.\mathtt{receivedValues}[grantID]$ then stop         ▷ No access token has been received yet
            let $domainRS \leftarrow s'.\mathtt{resourceServers}$                               ▷ Non-det. choose an RS
            if $(\mathsf{accessToken} \in^{\langle\rangle} originallyInqValues \vee \mathsf{bearerToken} \in^{\langle\rangle} originallyInqValues)$
              $\hookrightarrow$ $\wedge\, \mathtt{domainFirstRS} \notin s'.\mathtt{grants}[grantID]$ then
                let $s'.\mathtt{grants}[grantID][\mathtt{domainFirstRS}] := domainRS$
              ▷ Store domain of the first RS used in this flow to be able to return the resource stored on this RS to the browser later on
            let $accessToken := s'.\mathtt{receivedValues}[grantID][\mathtt{accessToken}]$
            let $reference := [\mathsf{responseTo}{:}\mathsf{resourceResponse}, \mathsf{grantID}{:}grantID, \mathsf{domainRS}{:}domainRS]$
            if $accessToken[\mathtt{flags}] \equiv \mathsf{bearer}$ then                          ▷ Access token is a bearer token
                let $s'.\mathtt{grants}[grantID][\mathtt{bearerRSs}] := s'.\mathtt{grants}[grantID][\mathtt{bearerRSs}] +^{\langle\rangle} domainRS$ ▷ Store RSs receiving bearer tkns
                let $authHeader := \langle \mathsf{Authorization}, \langle \mathsf{Bearer}, accessToken\rangle\rangle$
                let $request := \langle \mathsf{HTTPReq}, \nu_8, \mathsf{GET}, domainRS, /\mathsf{resource}, \langle\rangle, \langle authHeader\rangle, \langle\rangle\rangle$
                call HTTPS_SIMPLE_SEND($reference$, $request$, $s'$, $a$)
```

This algorithm is continued on the next page.

Continuation of Algorithm A.8 (Client PROCESS_TRIGGER)

```
else                                                          ▷ Access token is key-bound
    let $url := \langle \mathtt{URL}, \mathtt{S}, domainRS, \mathtt{/resource}, \langle\rangle \rangle$
    let $authHeader := \langle \mathtt{Authorization}, \langle \mathtt{GNAP}, accessToken \rangle \rangle$
    if $\mathtt{key} \in accessToken$ then                      ▷ Access token is bound to its own key
        let $keyData \in^{\langle\rangle} s'.\mathtt{tokenKeys}[domainAS]$ such that $\mathtt{pub}(keyData.\mathtt{key}) \equiv accessToken[\mathtt{key}][\mathtt{pubKey}]$
        let $method := keyData.\mathtt{method}$
        let $privateKey := keyData.\mathtt{key}$
        if $method \equiv \mathtt{sign}$ then
            let $keyID := keyData.\mathtt{keyID}$
            call SIGN_AND_SEND($\mathtt{GET}, url, keyID, privateKey, \mathtt{sign}, authHeader, \langle\rangle, reference, s', a$)
        else if $method \equiv \mathtt{mtls}$ then
            let $request := \langle \mathtt{HTTPReq}, v_8, \mathtt{GET}, domainRS, \mathtt{/resource}, \langle\rangle, \langle authHeader \rangle, \langle\rangle \rangle$
            let $body := [\mathtt{publicKey}{:}\mathtt{pub}(privateKey)]$
            let $message := \langle \mathtt{HTTPReq}, v_9, \mathtt{POST}, domainRS, \mathtt{/MTLS\text{-}prepare}, \langle\rangle, \langle\rangle, body \rangle$
            let $ref := [\mathtt{responseTo}{:}\mathtt{MTLS\_RR}, \mathtt{grantID}{:}grantID, \mathtt{key}{:}privateKey, \mathtt{reference}{:}reference, \mathtt{request}{:}request]$
            call HTTPS_SIMPLE_SEND($ref, message, s', a$)
        else stop                                               ▷ Unsupported method
    else                                        ▷ Access token is bound to client instance key
        let $keyRecord := s'.\mathtt{grants}[grantID][\mathtt{keyRecord}]$
        let $key := keyRecord.\mathtt{key}$
        if $keyRecord.\mathtt{method} \equiv \mathtt{sign}$ then
            let $keyID := keyRecord.\mathtt{keyID}$
            call SIGN_AND_SEND($\mathtt{GET}, url, keyID, key, \mathtt{sign}, authHeader, \langle\rangle, reference, s', a$)
        else if $keyRecord.\mathtt{method} \equiv \mathtt{mac}$ then
            let $keyID := keyRecord.\mathtt{keyID}$         ▷ We have to use the RS with which this symmetric key is shared
            let $domainRS := keyRecord.\mathtt{rs}$
            let $reference[\mathtt{domainRS}] := domainRS$
            let $url' := \langle \mathtt{URL}, \mathtt{S}, domainRS, \mathtt{/resource}, \langle\rangle \rangle$
            call SIGN_AND_SEND($\mathtt{GET}, url', keyID, key, \mathtt{mac}, authHeader, \langle\rangle, reference, s', a$)
        else                                                    ▷ $keyRecord.\mathtt{method} \equiv \mathtt{mtls}$
            let $request := \langle \mathtt{HTTPReq}, v_8, \mathtt{GET}, domainRS, \mathtt{/resource}, \langle\rangle, \langle authHeader \rangle, \langle\rangle \rangle$
            let $body := [\mathtt{publicKey}{:}\mathtt{pub}(key)]$
            let $message := \langle \mathtt{HTTPReq}, v_9, \mathtt{POST}, domainRS, \mathtt{/MTLS\text{-}prepare}, \langle\rangle, \langle\rangle, body \rangle$
            let $ref := [\mathtt{responseTo}{:}\mathtt{MTLS\_RR}, \mathtt{grantID}{:}grantID, \mathtt{reference}{:}reference, \mathtt{request}{:}request]$
            call HTTPS_SIMPLE_SEND($ref, message, s', a$)
```

Algorithm A.9 Relation of a Client Instance R^c: Generating inquired values.

```
function GENERATE_INQUIRED_VALUES($ROpresent$)
    let $inquiredValues := \langle\rangle$
    let $requestAccessToken \leftarrow \{\top, \bot\}$
    if $requestAccessToken \equiv \top \vee ROpresent \equiv \bot$ then
        let $setBearerFlag \leftarrow \{\top, \bot\}$
        if $setBearerFlag \equiv \top$ then
            let $inquiredValues := inquiredValues +^{\langle\rangle}$ bearerToken    ▷ Requested access token is a bearer token
        else
            let $inquiredValues := inquiredValues +^{\langle\rangle}$ accessToken    ▷ Requested access token is bound to a key
    if $ROpresent \equiv \top$ then
            ▷ Subject identifiers can only be requested if an RO is present. In software-only authorization there are no subject identifiers
        let $requestSubjectID \leftarrow \{\top, \bot\}$
        if $requestSubjectID \equiv \top \vee requestAccessToken \equiv \bot$ then
            let $inquiredValues := inquiredValues +^{\langle\rangle}$ subjectID
    return $inquiredValues$
```

Algorithm A.10 Relation of a Client Instance R^c: Sending a continuation request to finish interaction.

```
function SEND_CONTINUATION_REQUEST($grantID$, $interactRef$, $hash$, $s'$, $a$)
    let $CIfinishNonce := s'.\mathtt{grants}[grantID][\mathtt{CIfinishNonce}]$
    let $ASfinishNonce := s'.\mathtt{grants}[grantID][\mathtt{ASfinishNonce}]$
    let $domainAS := s'.\mathtt{grants}[grantID][\mathtt{AS}]$
    let $grantEndpoint := \langle \mathsf{URL}, \mathsf{S}, domainAS, \mathtt{/requestGrant}, \langle\rangle \rangle$
    let $controlHash := \mathsf{hash}(\langle CIfinishNonce, ASfinishNonce, interactRef, grantEndpoint \rangle)$
    if $hash \not\equiv controlHash$ then stop
    let $continueURL := s'.\mathtt{grants}[grantID][\mathtt{continueURL}]$
    let $s'.\mathtt{grants}[grantID][\mathtt{finishedInteraction}] := \top$        ▹ Save the information that the interaction is complete
    let $adjustInquiredValues \leftarrow \{\top, \bot\}$
    if $adjustInquiredValues \equiv \top$ then
        let $inquiredValues :=$ GENERATE_INQUIRED_VALUES($\top$)
        let $s'.\mathtt{grants}[grantID][\mathtt{requested}] := inquiredValues$
    let $keyRecord := s'.\mathtt{grants}[grantID][\mathtt{keyRecord}]$
    if $keyRecord.\mathtt{method} \not\equiv \mathtt{mtls}$ then
        let $authHeader := \langle \mathtt{Authorization}, \langle \mathtt{GNAP}, s'.\mathtt{grants}[grantID][\mathtt{continueAT}] \rangle \rangle$ }
        let $reference := [\mathtt{responseTo}{:}\mathtt{grantResponse}, \mathtt{grantID}{:}grantID, \mathtt{sentTo}{:}continueURL.\mathtt{host}]$
        if $adjustInquiredValues \equiv \bot$ then
            let $body := [\mathtt{interactRef}{:}interactRef]$
            call SIGN_AND_SEND($\mathtt{POST}$, $continueURL$, $keyRecord.\mathtt{keyID}$, $keyRecord.\mathtt{key}$, $keyRecord.\mathtt{method}$,
              ↪ $authHeader$, $body$, $reference$, $s'$, $a$)
        else
            let $body := [\mathtt{interactRef}{:}interactRef, \mathtt{inquiredValues}{:}inquiredValues]$
            call SIGN_AND_SEND($\mathtt{PATCH}$, $continueURL$, $keyRecord.\mathtt{keyID}$, $keyRecord.\mathtt{key}$, $keyRecord.\mathtt{method}$,
              ↪ $authHeader$, $body$, $reference$, $s'$, $a$)
    else                                                                         ▹ MTLS was used for grant request
        let $s'.\mathtt{grants}[grantID][\mathtt{interactRef}] := interactRef$
        if $adjustInquiredValues \equiv \top$ then
            let $s'.\mathtt{grants}[grantID][\mathtt{adjustedInquiredValues}] := \top$
        if $keyRecord.\mathtt{instanceID} \not\equiv \bot$ then
            let $body := [\mathtt{instanceID}{:}keyRecord.\mathtt{instanceID}]$
        else
            let $body := [\mathtt{publicKey}{:}\mathsf{pub}(keyRecord.\mathtt{key})]$
        let $message := \langle \mathsf{HTTPReq}, \nu_{10}, \mathtt{POST}, continueURL.\mathtt{host}, \mathtt{/MTLS\text{-}prepare}, \langle\rangle, \langle\rangle, body \rangle$
        let $reference := [\mathtt{responseTo}{:}\mathtt{MTLS\_CR}, \mathtt{grantID}{:}grantID]$
        call HTTPS_SIMPLE_SEND($reference$, $message$, $s'$, $a$)
```

Algorithm A.11 Relation of a Client Instance R^c: Returning resources and service session identifiers to browsers.

```
function SEND_RESPONSE_TO_BROWSER($grantID$, $s'$)
    if subjectID $\in s'$.receivedValues[$grantID$] then
        let $\langle subjectID, domainAS \rangle$ := $s'$.receivedValues[$grantID$][subjectID]
        let $sessionID$ := $s'$.grants[$grantID$][sessionID]
        let $s'$.sessions[$sessionID$][loggedInAs] := $\langle subjectID, domainAS \rangle$
        let $s'$.sessions[$sessionID$][serviceSessionID] := $\nu_{11}$
        let $headers$ := $\langle$Set-Cookie, $\langle\langle\langle$__Host, serviceSessionID$\rangle$, $\langle \nu_{11}, \top, \top, \top \rangle\rangle\rangle\rangle$
    else
        let $headers$ := $\langle\rangle$
    if domainFirstRS $\in s'$.grants[$grantID$] then
        let $domainFirstRS$ := $s'$.grants[$grantID$][domainFirstRS]
        let $body$ := $s'$.grants[$grantID$][resources][$domainFirstRS$]
    else
        let $body$ := ok
    let $\langle key, receiver, sender, nonce \rangle$ := $s'$.browserRequests[$grantID$][finishRequest]
    let $m'$ := $\mathsf{enc_s}(\langle$HTTPResp, $nonce$, 200, $headers$, $body\rangle$, $key$)
    let $s'$.browserRequests[$grantID$] := $s'$.browserRequests[$grantID$] − finishRequest
      ▷ Remove browser request to avoid re-sending a response in case of another grant response in response to a continuation request
    stop $\langle\langle sender, receiver, m' \rangle\rangle$, $s'$
```

Algorithm A.12 Relation of *script_ci_index*.

```
Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets \rangle$
              ▷ Script that models the index page of a client instance
let $switch \leftarrow$ {start, logout, link}      ▷ Non-deterministically decide whether to start a grant request or to follow some link
if $switch \equiv$ start then                     ▷ Start grant request
    let $url$ := GETURL($tree$, $docnonce$)
    let $\langle username, domain \rangle \leftarrow ids$      ▷ Non-det. select identity to specify its domain as AS
    let $url'$ := $\langle$URL, S, $url$.host, /startGrantRequest, $\langle\rangle\rangle$
    let $command$ := $\langle$STARTGRANT, $url'$, $domain\rangle$
    stop $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$
else if $switch \equiv$ logout then               ▷ Log out from the client instance to get a new session ID
    let $url$ := GETURL($tree$, $docnonce$)
    let $url'$ := $\langle$URL, S, $url$.host, /logout, $\langle\rangle\rangle$
    let $command$ := $\langle$FORM, $url'$, POST, $\langle\rangle$, $\bot\rangle$
    stop $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$
else                                              ▷ Follow link
    let $protocol \leftarrow$ {P, S}                   ▷ Non-det. select protocol (HTTP or HTTPS)
    let $host \leftarrow$ Doms                         ▷ Non-det. select host
    let $path \leftarrow \mathbb{S}$                   ▷ Non-det. select path
    let $fragment \leftarrow \mathbb{S}$               ▷ Non-det. select fragment part
    let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$   ▷ Non-det. select parameters
    let $url$ := $\langle$URL, $protocol$, $host$, $path$, $parameters$, $fragment\rangle$   ▷ Assemble URL
    let $command$ := $\langle$HREF, $url$, $\bot$, $\bot\rangle$     ▷ Follow link to the selected URL
    stop $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$
```

A.12 Authorization Servers

An authorization server $as \in \mathsf{AS}$ is a web server modeled as an atomic DY process $(I^{as}, Z^{as}, R^{as}, s_0^{as})$ with the addresses $I^{as} := \mathsf{addr}(as)$.

Definition A.5. We denote by *(public) key data*, a term of one of the following forms:

- $\langle \mathtt{sign}, \mathit{publicKey}, \mathit{keyID} \rangle$
- $\langle \mathtt{mac}, \mathit{key}, \mathit{keyID} \rangle$
- $\langle \mathtt{mtls}, \mathit{publicKey} \rangle$

with $\mathit{keyID} \in \mathsf{KeyIDs}$, $\mathit{key} \in K_{\mathrm{KP}}$, and $\mathit{publicKey} \in \mathcal{T}_{\mathcal{N}}$. *keyID* is the key ID that will be used by the client instance if signatures or MACs are used. If the client instance uses signatures or MTLS key proofs, *publicKey* is the public key that *as* will use to verify them. If MACs are used, *key* is the symmetric key that will be used to verify them.

For a key data r, we will use $r.\mathtt{method}$ as notation for $r.1$ and $r.\mathsf{pubKey}$ as notation for $r.2$.

In our modeling, ASs store data about registered client instances in *client registration records* and data about their users in *user records*:

Definition A.6. A *client registration record* is a term of the form $\langle \mathit{instanceID}, \mathit{kd} \rangle$ with $\mathit{instanceID} \in \mathbb{S}$ and *kd* public key data.

instanceID is the instance identifier that the registered client instance will use. For a client registration record r we will use $r.\mathtt{keyData}$ as notation for $r.2$.

Definition A.7. A *user record* is a term of the form

$$\langle \mathit{identity}, \mathit{password} \rangle$$

with $\mathit{identity} \in \mathsf{ID}$ and $\mathit{password} \in \mathsf{Passwords}$, where $\mathit{password} \equiv \mathsf{secretOfID}(\mathit{identity})$.

User records are used to store the credentials of the ROs that own resources protected by *as*.

Next, we define the set Z^{as} of states of *as* and the initial state s_0^{as} of *as*.

Definition A.8. A *state* $s \in Z^{as}$ *of AS as* is a term of the form $\langle$*DNSaddress*, *pendingDNS*, *corrupt*, *pendingRequests*, *keyMapping*, *tlskeys*, *registrations*, *users*, *mtlsRequests*, *sigNonces*, *grantRequests*, *tokenBindings*$\rangle$ with $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{pendingRequests} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$, $\mathit{registrations} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{clientTokenKeys} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{users} \in \left[\mathsf{ID} \times \mathcal{N}\right]$, $\mathit{mtlsRequests} \in \left[\mathcal{N} \times \mathcal{N}\right]$, $\mathit{sigNonces} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{grantRequests} \in \left[\mathcal{N} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$, and $\mathit{tokenBindings} \in \left[\mathcal{N} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$.

An *initial state* s_0^{as} *of as* is a state of *as* with $s_0^{as}.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^{as}.\mathtt{corrupt} \equiv \bot$, $s_0^{as}.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^{as}.\mathtt{keyMapping}$ being the same as the keymapping for browsers, $s_0^{as}.\mathtt{tlskeys} \equiv \mathit{tlskeys}^{as}$, and $s_0^{as}.\mathtt{mtlsRequests} \equiv s_0^{as}.\mathtt{sigNonces} \equiv s_0^{as}.\mathtt{grantRequests} \equiv s_0^{as}.\mathtt{tokenBindings} \equiv \langle\rangle$.

mtlsRequests is a dictionary that maps MTLS nonces to the keys used for MTLS. It is used to store the information required for validating MTLS key proofs.

sigNonces stores all nonces obtained from received valid signatures or MACs, thus enabling the replay protection required by GNAP's security considerations.

grantRequests works like the *grants* dictionary of the client instances, except that here information required by the AS is stored. As key for the outer dictionary again a nonce called *grantID* is used. Note that the grant IDs used for *grantRequests* are only used by *as* to internally distinguish grant requests. They are not equivalent (w.r.t. the equational theory) to the grant IDs of any client instances. Generally, grant IDs are not sent from any honest process to any other process.

tokenBindings maps from the values of issued access tokens (nonces) to dictionaries containing the information required by *as* for token introspection, such as the key proofing method to which the access token is bound or whether it is a bearer token.

$s_0^{as}.\mathtt{registrations}$ is a dictionary containing client registration records, with the instance identifiers of the client registration records functioning as keys. For any two distinct client registration records $r, r' \in^{\langle\rangle} s_0^{as}.\mathtt{registrations}$ it must hold that $r.\mathtt{instanceID} \not\equiv r'.\mathtt{instanceID}$. For each client registration record $r \in^{\langle\rangle} s_0^{as}.\mathtt{registrations}$ there must be exactly one client instance $c \in \mathsf{CI}$ such that for a domain $d \in \mathsf{dom}(as)$ there is a key record $r' \in^{\langle\rangle} s_0^{c}.\mathtt{keyRecords}[d]$ with $r'.\mathtt{instanceID} \equiv r.\mathtt{instanceID}$. If $r.\mathtt{keyData.method} \equiv \mathtt{sign}$ it must hold that

$$\begin{aligned} & r'.\mathtt{method} \equiv \mathtt{sign} \\ & \wedge r.\mathtt{keyData.keyID} \equiv r'.\mathtt{keyID} \\ & \wedge r.\mathtt{keyData.publicKey} \equiv \mathsf{pub}(r'.\mathtt{key})\ . \end{aligned}$$

If $r.\mathtt{keyData.method} \equiv \mathtt{mac}$ it must hold that

$$\begin{aligned} & r'.\mathtt{method} \equiv \mathtt{mac} \\ & \wedge r.\mathtt{keyData.keyID} \equiv r'.\mathtt{keyID} \\ & \wedge r.\mathtt{keyData.key} \equiv r'.\mathtt{key}\ . \end{aligned}$$

If $r.\mathtt{keyData.method} \equiv \mathtt{mtls}$ it must hold that

$$\begin{aligned} & r'.\mathtt{method} \equiv \mathtt{mtls} \\ & \wedge r.\mathtt{keyData.publicKey} \equiv \mathsf{pub}(r'.\mathtt{key})\ . \end{aligned}$$

$s_0^{as}.\mathtt{clientTokenKeys}$ is a dictionary containing a sequence of public key data that can be used to bind access tokens to. The key is the client instance identifier. We only have entries for registered clients. That is, we require for every $id \in s_0^{as}.\mathtt{clientTokenKeys}$ that there is an entry $r \in s_0^{as}.\mathtt{registrations}$ with $id \equiv r.\mathtt{instanceID}$. Further, every public key in this set has to have a corresponding private key at the associated CI: For every $id \in s_0^{as}.\mathtt{clientTokenKeys}$ and every $kd \in s_0^{as}.\mathtt{clientTokenKeys}[id]$ there has to be exactly one client instance c with a token key record $kr \in s_0^{c}.\mathtt{tokenKeys}[as]$ such that $kr.\mathtt{instanceID} \equiv id$ and $\mathsf{pub}(kr.\mathtt{key}) \equiv k.\mathsf{pubKey}$. Fur such associated key records kr and key data kd, we need the $\mathtt{method}$ to be the same. Let $kd \in s_0^{as}.\mathtt{clientTokenKeys}[id]$ be a key data stored at *as* for some client instance c with the instance ID id, and $kr \in s_0^{c}.\mathtt{tokenKeys}[as]$ be the corresponding key record stored at c for *as*, then we have $kd.\mathtt{method} \equiv kr.\mathtt{method}$.

$s_0^{as}.\mathtt{users}$ is a dictionary containing user records. For each user record $u \in^{\langle\rangle} s_0^{as}.\mathtt{users}$ it must hold that $u.\mathtt{identity.domain} \in$

dom(as). We also require that for all users $u \neq u'$ in $s_0^{as}.\texttt{users}$ we have $u.\texttt{identity} \not\equiv u'.\texttt{identity}$.

For each $b \in \mathsf{B}$, $i \in^{\langle\rangle} s_0^{b}.\texttt{ids}$ there must be an AS as that contains a user record $u \in^{\langle\rangle} s_0^{as}.\texttt{users}$ such that $i \equiv u.\texttt{identity}$ and vice versa. $\mathsf{secretOfID}(i)$ must initially be stored only in b and as.

To allow us to examine whether GNAP satisfies our security properties even when key-bound access tokens leak, we have the ASs send them not only to the client instance whose key the access token is bound to, but also to another randomly chosen IP address. For this we use an arbitrary IP address $leak \in \mathsf{IPs}$ as in Fett et al. [21]. We leak all continuation access tokens, because they are always bound to the client instances key, as well as all access tokens for accessing resources that are not bearer tokens.

We now specify the relation R^{as}: This relation is again based on the generic HTTPS server model defined in [29]. Table 3 shows a list of all placeholders used in the algorithms.

	Usage
ν_1	new grant ID
ν_2	new continuation access token
ν_3	nonce to generate a unique redirect URL
ν_4	new user code
ν_5	nonce for the calculation of the interaction finish hash
ν_6	nonce for MTLS
ν_7	new interaction reference
ν_8	new HTTP request nonce
ν_9	new key for the *pendingDNS* dictionary
ν_{10}	new access token
ν_{11}	new continuation access token

Table 3: List of placeholders used in the AS algorithms.

The script that is used by the authorization server is described in Algorithm A.19. It is used when the RO logs in to the AS after a redirect or using a user code.

The following algorithms are used for modeling the authorization servers:

- Algorithm A.13 processes HTTPS requests to AS originating either from a client instance or, in the case of introspection requests, from an RS.
  For the AS, each flow starts with the reception of a grant request. How a grant request is handled depends on the information in it. If neither an access token nor a subject identifier is requested, the request can be answered directly, since no values are returned. If an access token for an RO or a subject identifier of an RO is requested, the AS initiates the interaction with the RO. In case the client instance uses software-only authorization, the AS returns the requested values directly if the client instance is registered and the key proof has been validated successfully.
  Continuation requests can be either HTTP POST requests or HTTP PATCH requests. A POST request finishes an interaction, while a PATCH request adjusts the values requested by the client instance. A PATCH request can also be used to finish the interaction along with the adjustment of the grant request. In this case, the correct interaction reference must be included in the PATCH request, as it is the case with a POST request. Continuation requests must always contain the continuation access token in the `Authorization` header.
  At the `/interact` path, the AS accepts redirects that are sent as part of the redirect interaction start mode. The login process is then simulated by the script *script_as_login*. We do not model a specific authorization process for grant requests, since such a process is out of scope for GNAP. Instead, in our model, the RO automatically authorizes the request by successfully logging in. The nonce with which the AS can associate the login process with the grant request is passed to *script_as_login* via the *scriptstate*. The script then sends the login data to the `/redirectLogin` path and includes that nonce in the body of the login request.
  The `/interactUC` path enables logins using user codes. The user code is modeled similarly to the nonce used to uniquely identify the request when using the redirect interaction start mode since the user code essentially has the same function. The user code is passed to *script_as_login* via the *scriptinputs* and then included in the body when logging in via the `/userCodeLogin` path.
  For requests to the introspection endpoint (`/introspect`), the key proof of the requesting RS is validated first. If successful, the access token is taken from the request body and the values stored in the *tokenBindings* subterm for this access token are returned to the RS.
- Algorithm A.14 performs key proofing methods using VALIDATE_KEY_PROOF to validate that a request received by *as* was actually sent by the owner of the key specified in the corresponding grant request. In order to call VALIDATE_KEY_PROOF, in the case of a grant request, the algorithm extracts the key information contained in the request and stores it in the *grantRequests* subterm. In the case of a continuation request, the algorithm then reloads the previously stored key information from *grantRequests* using the grant ID.
- Algorithm A.15 checks the RO's login credentials and, if successful, performs the interaction finish mode specified by the client instance in the grant request. For this purpose, the algorithm creates a new interaction reference in the form of a nonce and uses it to calculate the hash value that is used by the client instance for verification of the interaction finish.
- Algorithm A.16 is used when the AS needs to send a grant response to a client instance after completing the interaction with the RO.
- Algorithm A.17 creates grant responses depending on the values requested in the grant request. If an access token is generated, the algorithm stores the values required for token introspection in the *tokenBindings* subterm. If a subject identifier is requested, the identity with which the RO has logged in to *as* is included in the grant response. If the client instance requests a key-bound access token, a non-deterministic decision is made whether to bind it to the client instance's key and key proofing method or to bind the access token to its own key. In the latter case, a key is chosen non-deterministically from the `clientTokenKeys` entry of the respective CI. Hence, this is only possible for pre-registered

CIs. Otherwise, the CI had to send the corresponding allowed keys to AS during the initial grant request. However, GNAP does not specify how this sharing of keys should be handled.

- Algorithm A.18 emits grant responses and leaks access tokens in doing so. If the grant response contains a continuation access token, it is leaked. If it contains a key-bound access token for resources, this is also leaked. The receiver address of the events with which access tokens are leaked is *leak*.

Algorithm A.13 Relation of an AS R^{as}: Processing HTTPS requests.

```
function PROCESS_HTTPS_REQUEST($m, k, a, f, s'$)
    if $m$.path ≡ /requestGrant ∧ $m$.method ≡ POST then                                   ▷ New grant request
        let $grantID := v_1$
        let $s'$ := PERFORM_KEY_PROOF($m, grantID, s'$)
        let $grantRequest := m$.body
        let $grantResponse$ := [ ]
        let $continueAT := v_2$
        let $continueURL$ := ⟨URL, S, $m$.host, /continue, ⟨⟩⟩
        let $s'$.grantRequests[$grantID$] := [inquiredValues:$grantRequest$[inquiredValues]]
        if $grantRequest$[inquiredValues] ≡ ⟨⟩ then                                  ▷ Client instance requested nothing
            let $allowContinuation \leftarrow \{\top, \bot\}$
            if $allowContinuation \equiv \top$ then
                let $grantResponse$[continue] := [accessToken:$continueAT$, url:$continueURL$]
                let $s'$.grantRequests[$grantID$][continueAT] := $continueAT$
        else if $grantRequest$[interaction] ≡ ⊤ then                                 ▷ Interaction with RO is possible
            let $grantResponse$[continue] := [accessToken:$continueAT$, url:$continueURL$]
            let $redirectNonce := v_3$
            let $userCode := v_4$
            let $ASfinishNonce := v_5$
            let $redirectURL$ := ⟨URL, S, $m$.host, /interact, [request:$redirectNonce$]⟩
            let $grantResponse$[interact] := [redirect:$redirectURL$, userCode:$userCode$, finish:$ASfinishNonce$]
            let $s'$.grantRequests[$grantID$][continueAT] := $continueAT$
            let $s'$.grantRequests[$grantID$][redirectNonce] := $redirectNonce$
            let $s'$.grantRequests[$grantID$][userCode] := $userCode$
            let $s'$.grantRequests[$grantID$][ASfinishNonce] := $ASfinishNonce$
            let $s'$.grantRequests[$grantID$][CIfinishNonce] := $grantRequest$[finish][nonce]
            let $s'$.grantRequests[$grantID$][finishMode] := $grantRequest$[finish][finishMode]
            let $s'$.grantRequests[$grantID$][finishURL] := $grantRequest$[finish][finishURL]
            let $s'$.grantRequests[$grantID$][grantEndpoint] := ⟨URL, S, $m$.host, $m$.path, ⟨⟩⟩
            if user ∈ $grantRequest$ then                                        ▷ Client instance included subject identifier
                let $s'$.grantRequests[$grantID$][user] := $grantRequest$[user]
        else                                                                 ▷ No end user is present at the client instance
            if instanceID ∉ $grantRequest$ then stop      ▷ Only a registered client instance can get an access token for its resources
            let ⟨$grantResponse, s'$⟩ := CREATE_GRANT_RESPONSE($grantID$, CI, $grantRequest$[inquiredValues], ⊥, $m$.host, $s'$)
        if client ∈ $grantRequest$ then                                                   ▷ Client instance is not registered
            let $registerCI \leftarrow \{\top, \bot\}$                                  ▷ Non-det. decide whether to register the client instance
            if $registerCI \equiv \top$ then
                let $instanceID \leftarrow \mathbb{S}$ such that $instanceID \notin s'$.registrations
                if $grantRequest$[client][method] ≡ sign then
                    let $keyID := grantRequest$[client][keyID]
                    let $publicKey := grantRequest$[client][key]
                    let $s'$.registrations[$instanceID$] := ⟨sign, $keyID, publicKey$⟩
                else                                                                ▷ $grantRequest$[client][method] ≡ mtls
                    let $publicKey := grantRequest$[client][key]
                    let $s'$.registrations[$instanceID$] := ⟨mtls, $publicKey$⟩
                let $grantResponse$[instanceID] := $instanceID$
        let $m' := \mathsf{enc_s}$(⟨HTTPResp, $m$.nonce, 200, ⟨⟩, $grantResponse$⟩, $k$)
        call STOP_WITH_LEAKS($f, a, m', grantResponse, s'$)
```

This algorithm is continued on the next page.

Continuation of Algorithm A.13 (AS PROCESS_HTTPS_REQUEST)

```
else if $m.\mathtt{path} \equiv \mathtt{/continue} \wedge m.\mathtt{method} \equiv \mathtt{POST}$ then    ▷ Continuation request after interaction completed
    if $m.\mathtt{headers}[\mathtt{Authorization}].1 \not\equiv \mathtt{GNAP}$ then stop    ▷ Wrong Authentication scheme was used
    let $continueAT := m.\mathtt{headers}[\mathtt{Authorization}].2$
    if $continueAT \equiv \langle\rangle$ then stop    ▷ Access token for continuation must be a nonce
    let $grantID$ such that $s'.\mathtt{grantRequests}[grantID][\mathtt{continueAT}] \equiv continueAT$ if possible; otherwise stop
    call PERFORM_KEY_PROOF($m, grantID, s'$)
    if $\mathtt{interactRef} \not\in s'.\mathtt{grantRequests}[grantID]$ then stop    ▷ An interaction reference must exist when using this endpoint
    if $\mathtt{interactRef} \not\in m.\mathtt{body}$ then stop    ▷ The client instance must specify an interaction reference
    let $interactRef := m.\mathtt{body}[\mathtt{interactRef}]$
    let $inquiredValues := s'.\mathtt{grantRequests}[grantID][\mathtt{inquiredValues}]$
    call SEND_GRANT_RESPONSE($grantID, interactRef, inquiredValues, m, k, a, f, s'$)
else if $m.\mathtt{path} \equiv \mathtt{/continue} \wedge m.\mathtt{method} \equiv \mathtt{PATCH}$ then    ▷ Grant request modification
    if $m.\mathtt{headers}[\mathtt{Authorization}].1 \not\equiv \mathtt{GNAP}$ then stop    ▷ Wrong Authentication scheme was used
    let $continueAT := m.\mathtt{headers}[\mathtt{Authorization}].2$
    if $continueAT \equiv \langle\rangle$ then stop    ▷ Access token for continuation must be a nonce
    let $grantID$ such that $s'.\mathtt{grantRequests}[grantID][\mathtt{continueAT}] \equiv continueAT$ if possible; otherwise stop
    call PERFORM_KEY_PROOF($m, grantID, s'$)
    let $inquiredValues := m.\mathtt{body}[\mathtt{inquiredValues}]$
    if $\mathtt{interactRef} \in s'.\mathtt{grantRequests}[grantID]$ then    ▷ Interaction is not yet complete
        if $\mathtt{interactRef} \not\in m.\mathtt{body}$ then stop    ▷ The interaction reference is required to finish the interaction
        let $interactRef := m.\mathtt{body}[\mathtt{interactRef}]$
        call SEND_GRANT_RESPONSE($grantID, interactRef, inquiredValues, m, k, a, f, s'$)
    else    ▷ Interaction has already been completed or software-only authorization
        if $\mathtt{interactRef} \in m.\mathtt{body}$ then stop    ▷ Request is not allowed to contain an interaction reference
        let $s'.\mathtt{grantRequests}[grantID] := s'.\mathtt{grantRequests}[grantID] - \mathtt{continueAT}$
        if $\mathtt{finishMode} \in s'.\mathtt{grantRequests}[grantID]$ then    ▷ End user is present
            let $\langle grantResponse, s'\rangle :=$ CREATE_GRANT_RESPONSE($grantID, \mathtt{endUser}, inquiredValues, continueAT, m.\mathtt{host}, s'$)
        else    ▷ Software-only authorization
            let $\langle grantResponse, s'\rangle :=$ CREATE_GRANT_RESPONSE($grantID, \mathtt{CI}, inquiredValues, continueAT, m.\mathtt{host}, s'$)
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\rangle, grantResponse\rangle, k)$
        call STOP_WITH_LEAKS($f, a, m', grantResponse, s'$)
else if $m.\mathtt{path} \equiv \mathtt{/interact} \wedge m.\mathtt{method} \equiv \mathtt{GET}$ then    ▷ Interaction using redirect
    if $\mathtt{request} \in m.\mathtt{parameters}$ then
        let $headers := [\mathtt{ReferrerPolicy{:}origin}]$
        let $request := m.\mathtt{parameters}[\mathtt{request}]$
        let $referrer := m.\mathtt{headers}[\mathtt{Referer}]$
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, headers, \langle \mathtt{script\_as\_login}, [\mathtt{request}{:}request, \mathtt{referrer}{:}referrer]\rangle\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle, s'$
    else stop    ▷ The parameters need to contain a request identifier
else if $m.\mathtt{path} \equiv \mathtt{/interactUC} \wedge m.\mathtt{method} \equiv \mathtt{GET}$ then    ▷ Interaction using user code
    if $\mathtt{user\text{-}code} \not\in m.\mathtt{parameters}$ then stop    ▷ The parameters need to contain a user code
    let $headers := [\mathtt{ReferrerPolicy{:}origin}]$
    let $userCode := m.\mathtt{parameters}[\mathtt{user\text{-}code}]$
    let $grantID$ such that $s'.\mathtt{grantRequests}[grantID][\mathtt{userCode}] \equiv userCode$ if possible; otherwise stop
    let $domainCI := s'.\mathtt{grantRequests}[grantID][\mathtt{finishURL}].\mathtt{host}$
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, headers, \langle \mathtt{script\_as\_login}, domainCI\rangle\rangle, k)$
    stop $\langle\langle f, a, m'\rangle\rangle, s'$
else if $m.\mathtt{path} \equiv \mathtt{/redirectLogin} \wedge m.\mathtt{method} \equiv \mathtt{POST} \wedge m.\mathtt{headers}[\mathtt{Origin}] \equiv \langle m.\mathtt{host}, \mathtt{S}\rangle$ then
    let $redirectNonce := m.\mathtt{body}[\mathtt{request}]$
    let $grantID$ such that $s'.\mathtt{grantRequests}[grantID][\mathtt{redirectNonce}] \equiv redirectNonce$ if possible; otherwise stop
    if $s'.\mathtt{grantRequests}[grantID][\mathtt{finishMode}] \equiv \mathtt{push}$
     ↪ $\wedge\, m.\mathtt{body}[\mathtt{referrer}].\mathtt{host} \not\equiv s'.\mathtt{grantRequests}[grantID][\mathtt{finishURL}].\mathtt{host}$ then
        stop    ▷ Check that browser was redirected by the client that sent the grant request to prevent client instance mix-up attack
    call FINISH_INTERACTION($grantID, m, k, a, f, s'$)
```

This algorithm is continued on the next page.

Continuation of Algorithm A.13 (AS PROCESS_HTTPS_REQUEST)

```
    else if $m.\mathtt{path} \equiv \mathtt{/userCodeLogin} \wedge m.\mathtt{method} \equiv \mathtt{POST} \wedge m.\mathtt{headers}[\mathtt{Origin}] \equiv \langle m.\mathtt{host}, \mathtt{S}\rangle$ then
        let $userCode := m.\mathtt{body}[\mathtt{userCode}]$
        let $grantID$ such that $s'.\mathtt{grantRequests}[grantID][\mathtt{userCode}] \equiv userCode$ if possible; otherwise stop
        call FINISH_INTERACTION($grantID, m, k, a, f, s'$)
    else if $m.\mathtt{path} \equiv \mathtt{/introspect} \wedge m.\mathtt{method} \equiv \mathtt{POST}$ then        ▷ Token introspection
        let $method := m.\mathtt{body}[\mathtt{RS}][\mathtt{method}]$
        let $key := m.\mathtt{body}[\mathtt{RS}][\mathtt{key}]$
        if $method \equiv \mathtt{sign}$ then
            let $keyID := m.\mathtt{body}[\mathtt{RS}][\mathtt{keyID}]$
            let $s' :=$ VALIDATE_KEY_PROOF($method, m, keyID, key, s'$)
        else if $method \equiv \mathtt{mtls}$ then
            let $s' :=$ VALIDATE_KEY_PROOF($method, m, \bot, key, s'$)
        else stop        ▷ Unsupported method
        let $accessToken := m.\mathtt{body}[\mathtt{accessToken}]$
        if $accessToken \notin s'.\mathtt{tokenBindings}$ then
            let $body := [\mathtt{active}{:}\bot]$        ▷ Unknown access token
        else
            let $body := [\mathtt{active}{:}\top]$
            let $binding := s'.\mathtt{tokenBindings}[accessToken]$
            let $type := binding[\mathtt{type}]$
            let $grantID := binding[\mathtt{grantID}]$
            let $grantRequest := s'.\mathtt{grantRequests}[grantID]$
            if $type \equiv \mathtt{newSign}$ then
                let $body[\mathtt{key}] := [\mathtt{keyID}{:}binding[\mathtt{keyID}], \mathtt{key}{:}binding[\mathtt{publicKey}], \mathtt{method}{:}\mathtt{sign}]$
            else if $type \equiv \mathtt{newMTLS}$ then
                let $body[\mathtt{key}] := [\mathtt{key}{:}binding[\mathtt{publicKey}], \mathtt{method}{:}\mathtt{mtls}]$
            else if $type \equiv \mathtt{CIKey}$ then
                let $method := grantRequest[\mathtt{method}]$
                if $method \equiv \mathtt{sign}$ then
                    let $body[\mathtt{key}] := [\mathtt{keyID}{:}grantRequest[\mathtt{keyID}], \mathtt{key}{:}grantRequest[\mathtt{publicKey}], \mathtt{method}{:}\mathtt{sign}]$
                else if $method \equiv \mathtt{mac}$ then        ▷ RS must already know the key since a symmetric key is used
                    let $body[\mathtt{instanceID}] := grantRequest[\mathtt{instanceID}]$
                else        ▷ $method \equiv \mathtt{mtls}$
                    let $body[\mathtt{key}] := [\mathtt{key}{:}grantRequest[\mathtt{clientKey}], \mathtt{method}{:}\mathtt{mtls}]$
            else        ▷ $type \equiv \mathtt{bearer}$
                let $body[\mathtt{flags}] := \mathtt{bearer}$
            if $binding[\mathtt{for}] \equiv \mathtt{endUser}$ then        ▷ Access token is used to access resources of an end user
                let $body[\mathtt{access}] := [\mathtt{identity}{:}grantRequest[\mathtt{subjectID}]]$
            else        ▷ Access token is used to access resources of a client instance
                let $body[\mathtt{access}] := [\mathtt{instanceID}{:}grantRequest[\mathtt{instanceID}]]$
        let $m' := \mathsf{enc_s}(\langle\mathtt{HTTPResp}, m.\mathtt{nonce}, \mathtt{200}, \langle\rangle, body\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle, s'$
    else if $m.\mathtt{path} \equiv \mathtt{/MTLS\text{-}prepare} \wedge m.\mathtt{method} \equiv \mathtt{POST}$ then
        let $mtlsNonce := \nu_6$
        if $\mathtt{instanceID} \in m.\mathtt{body}$ then        ▷ Client instance is registered
            let $instanceID := m.\mathtt{body}[\mathtt{instanceID}]$
            if $instanceID \notin s'.\mathtt{registrations}$ then stop        ▷ $as$ does not know this instance identifier
            if $s'.\mathtt{registrations}[instanceID].\mathtt{method} \not\equiv \mathtt{mtls}$ then stop        ▷ This client instance does not use MTLS
            let $clientKey := s'.\mathtt{registrations}[instanceID].\mathtt{publicKey}$
        else if $\mathtt{publicKey} \in m.\mathtt{body}$ then        ▷ Client instance is not registered
            let $clientKey := m.\mathtt{body}[\mathtt{publicKey}]$
        else stop        ▷ Information to determine $clientKey$ is missing
        let $s'.\mathtt{mtlsRequests} := s'.\mathtt{mtlsRequests} +^{\langle\rangle} \langle mtlsNonce, clientKey\rangle$
        let $m' := \mathsf{enc_s}(\langle\mathtt{HTTPResp}, m.\mathtt{nonce}, \mathtt{200}, \langle\rangle, \mathsf{enc_a}(\langle mtlsNonce, s'.\mathtt{keyMapping}[m.\mathtt{host}]\rangle, clientKey)\rangle, k)$
        stop $\langle\langle f, a, m'\rangle\rangle, s'$
    else stop        ▷ Unsupported operation
```

Algorithm A.14 Relation of an AS R^{as}: Check signature or MTLS nonce.

```
function PERFORM_KEY_PROOF($m$, $grantID$, $s'$)
    if $grantID \notin s'$.grantRequests then                              ▷ New grant request
        let $grantRequest := m$.body
        if instanceID $\in grantRequest \wedge$ client $\notin grantRequest$ then
            let $instanceID := grantRequest$[instanceID]
            if $instanceID \notin s'$.registrations then stop               ▷ Instance identifier is unknown to $as$
            let $s'$.grantRequests[$grantID$][instanceID] := $instanceID$
            let $keyData := s'$.registrations[$instanceID$]
            let $method := keyData$.method
            if $method \equiv$ sign then
                let $keyID := keyData$.keyID
                let $publicKey := keyData$.publicKey
            else if $method \equiv$ mac then
                let $keyID := keyData$.keyID
                let $key := keyData$.key
            else                                                            ▷ MTLS
                let $clientKey := keyData$.publicKey
        else if instanceID $\notin grantRequest \wedge$ client $\in grantRequest$ then
            let $method := grantRequest$[client][method]
            if $method \equiv$ sign then
                let $keyID := grantRequest$[client][keyID]
                let $publicKey := grantRequest$[client][key]
            else if $method \equiv$ mtls then
                let $clientKey := grantRequest$[client][key]
            else stop                                                       ▷ Invalid method or no method specified
        else stop                                          ▷ client or instanceID must be specified, but not both
        let $s'$.grantRequests[$grantID$][method] := $method$
        if $method \equiv$ sign then
            let $s'$.grantRequests[$grantID$][keyID] := $keyID$
            let $s'$.grantRequests[$grantID$][publicKey] := $publicKey$
        else if $method \equiv$ mac then
            let $s'$.grantRequests[$grantID$][keyID] := $keyID$
            let $s'$.grantRequests[$grantID$][key] := $key$
        else                                                                ▷ MTLS
            let $s'$.grantRequests[$grantID$][clientKey] := $clientKey$
    else                                                                    ▷ Continuation request
        let $method := s'$.grantRequests[$grantID$][method]
        if $method \equiv$ sign then
            let $keyID := s'$.grantRequests[$grantID$][keyID]
            let $publicKey := s'$.grantRequests[$grantID$][publicKey]
        else if $method \equiv$ mac then
            let $keyID := s'$.grantRequests[$grantID$][keyID]
            let $key := s'$.grantRequests[$grantID$][key]
        else                                                                ▷ MTLS
            let $clientKey := s'$.grantRequests[$grantID$][clientKey]
    if $method \equiv$ sign then
        return VALIDATE_KEY_PROOF($method, m, keyID, publicKey, s'$)
    else if $method \equiv$ mac then
        return VALIDATE_KEY_PROOF($method, m, keyID, key, s'$)
    else                                                                    ▷ MTLS
        return VALIDATE_KEY_PROOF($method, m, \perp, clientKey, s'$)
```

Algorithm A.15 Relation of an AS R^{as}: Check login and perform interaction finish mode.

```
function FINISH_INTERACTION($grantID, m, k, a, f, s'$)
    if subjectID $\in s'$.grantRequests[$grantID$] then stop    ▷ Interaction has already been completed for this request
    let $identity$ := $m$.body[identity]
    let $password$ := $m$.body[password]
    if $identity \notin s'$.users then stop    ▷ Identity is not registered at this AS
    if user $\in s'$.grantRequests[$grantID$] $\wedge$ $identity \not\equiv s'$.grantRequests[$grantID$][user] then
        stop    ▷ Identity that logged in does not match identity specified in grant request
    if $password \not\equiv s'$.users[$identity$] then stop    ▷ Incorrect password was provided
    let $interactRef$ := $\nu_7$
    let $CIfinishNonce$ := $s'$.grantRequests[$grantID$][CIfinishNonce]
    let $ASfinishNonce$ := $s'$.grantRequests[$grantID$][ASfinishNonce]
    let $finishMode$ := $s'$.grantRequests[$grantID$][finishMode]
    let $finishURL$ := $s'$.grantRequests[$grantID$][finishURL]
    let $grantEndpoint$ := $s'$.grantRequests[$grantID$][grantEndpoint]
    let $hash$ := hash($\langle CIfinishNonce, ASfinishNonce, interactRef, grantEndpoint\rangle$)
    let $s'$.grantRequests[$grantID$][interactRef] := $interactRef$
    let $s'$.grantRequests[$grantID$][subjectID] := $identity$    ▷ Once subject ID is set, interaction is treated as complete.
    if $finishMode \equiv$ redirect then
        let $finishURL$.parameters[interactRef] := $interactRef$
        let $finishURL$.parameters[hash] := $hash$
        let $m'$ := $\mathsf{enc_s}$($\langle$HTTPResp, $m$.nonce, 303, [Location:$finishURL$], $\langle\rangle\rangle$, $k$)
        stop $\langle\langle f, a, m'\rangle\rangle, s'$
    else if $finishMode \equiv$ push then
        let $body$ := [interactRef:$interactRef$, hash:$hash$]
        let $message$ := $\langle$HTTPReq, $\nu_8$, POST, $finishURL$.host, $finishURL$.path, $finishURL$.parameters, $\langle\rangle$, $body\rangle$
        let $s'$.pendingDNS[$\nu_9$] := $\langle\bot, message\rangle$    ▷ Simulate HTTPS_SIMPLE_SEND because we have to emit two events
        let $dataURL$ := $\langle$URL, S, $finishURL$.host, /getData, $finishURL$.parameters$\rangle$
        let $m'$ := $\mathsf{enc_s}$($\langle$HTTPResp, $m$.nonce, 303, [Location:$dataURL$], $\langle\rangle\rangle$, $k$)
                ▷ Redirect browser to client instance in order to be able to send data to browser
        stop $\langle\langle s'$.DNSaddress, $a$, $\langle$DNSResolve, $message$.host, $\nu_9\rangle\rangle, \langle f, a, m'\rangle\rangle, s'$
    else stop    ▷ Invalid interaction finish mode
```

Algorithm A.16 Relation of an AS R^{as}: Send grant response after interaction has finished.

```
function SEND_GRANT_RESPONSE($grantID, interactRef, inquiredValues, m, k, a, f, s'$)
    if $interactRef \not\equiv s'$.grantRequests[$grantID$][interactRef] then
        stop    ▷ Received interaction reference does not match the stored one
    let $\langle grantResponse, s'\rangle$ := CREATE_GRANT_RESPONSE($grantID$, endUser, $inquiredValues$,
      ↪ $s'$.grantRequests[$grantID$][continueAT], $m$.host, $s'$)
    let $s'$.grantRequests[$grantID$] := $s'$.grantRequests[$grantID$] $-$ interactRef    ▷ Prevent reuse
    let $m'$ := $\mathsf{enc_s}$($\langle$HTTPResp, $m$.nonce, 200, $\langle\rangle$, $grantResponse\rangle$, $k$)
    call STOP_WITH_LEAKS($f, a, m', grantResponse, s'$)
```

Algorithm A.17 Relation of an AS R^{as}: Creating a grant response.

```
function CREATE_GRANT_RESPONSE($grantID$, $for$, $inquiredValues$, $oldContinueAT$, $host$, $s'$)
    if subjectID $\notin s'$.grantRequests[$grantID$] then stop    ▷ Interaction for this request is not yet complete
    if subjectID $\in^{\langle\rangle}$ $inquiredValues \wedge for \equiv$ CI then stop ▷ No request of subject identifiers when using software-only authorization
    let $grantResponse$ := []
    if accessToken $\in^{\langle\rangle}$ $inquiredValues$ then
        let $accessToken := v_{10}$
        let $grantResponse$[accessToken] := [value:$accessToken$]
        if instanceID $\in^{\langle\rangle}$ $s'$.grantRequests[$grantID$] then
            let $bindToNewKey \leftarrow \{\top, \bot\}$    ▷ Non-det. decide to bind access token to different key for pre-registered CIs
            if $bindToNewKey \equiv \top$ then    ▷ Access token is bound to its own key
                let $instanceID$ := $s'$.grantRequests[$grantID$][instanceID]
                let $keyData \leftarrow s'$.clientTokenKeys[$instanceID$]    ▷ Non-det. choose a key from this client
                let $pubKey$ := $keyData$.pubKey
                let $keyID$ := $keyData$.keyID
                if $keyData$.method $\equiv$ sign then
                    let $s'$.tokenBindings[$accessToken$] := [grantID:$grantID$, for:$for$, type:newSign, keyID:$keyID$,
                  ↪ publicKey:$pubKey$]
                    let $grantResponse$[accessToken][key] := [pubKey:$pubKey$]
                else
                    let $s'$.tokenBindings[$accessToken$] := [grantID:$grantID$, for:$for$, type:newMTLS, publicKey:$pubKey$)]
                    let $grantResponse$[accessToken][key] := [pubKey:$pubKey$]
        else    ▷ Access token is bound to client instances key
            let $s'$.tokenBindings[$accessToken$] := [grantID:$grantID$, for:$for$, type:CIKey]
    else if bearerToken $\in^{\langle\rangle}$ $inquiredValues$ then
        let $bearerToken := v_{10}$
        let $s'$.tokenBindings[$bearerToken$] := [grantID:$grantID$, for:$for$, type:bearer]
        let $grantResponse$[accessToken] := [value:$bearerToken$, flags:bearer]
    if subjectID $\in^{\langle\rangle}$ $inquiredValues$ then
        let $grantResponse$[subjectID] := [value:$s'$.grantRequests[$grantID$][subjectID]]
    let $allowContinuation \leftarrow \{\top, \bot\}$
    if $allowContinuation \equiv \top$ then
        let $keepOldAT \leftarrow \{\top, \bot\}$
        if $keepOldAT \equiv \top \wedge oldContinueAT \not\equiv \bot$ then
            let $continueAT := oldContinueAT$    ▷ Continue access token does not change
        else
            let $continueAT := v_{11}$
        let $continueURL$ := $\langle$URL, S, $host$, /continue, $\langle\rangle\rangle$
        let $grantResponse$[continue] := [accessToken:$continueAT$, url:$continueURL$]
        let $s'$.grantRequests[$grantID$][continueAT] := $continueAT$    ▷ Overwrites old continueAT
    return $\langle grantResponse, s'\rangle$
```

Algorithm A.18 Relation of an AS R^{as}: Leaking access tokens.

```
function STOP_WITH_LEAKS($f$, $a$, $m'$, $grantResponse$, $s'$)
    let $events := \langle\langle f, a, m'\rangle\rangle$
    if continue $\in grantResponse$ then    ▷ Leakage of continuation access token
        let $events := events +^{\langle\rangle} \langle leak, a, \langle$LEAK, $grantResponse$[continue][accessToken]$\rangle\rangle$
    if accessToken $\in grantResponse \wedge grantResponse$[accessToken][type] $\not\equiv$ bearer then    ▷ Leakage of key-bound access token
        let $events := events +^{\langle\rangle} \langle leak, a, \langle$LEAK, $grantResponse$[accessToken]$\rangle\rangle$
    stop $events, s'$
```

Algorithm A.19 Relation of *script_as_login*.

```
Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$
        ▷ Script that models the login page of an AS
let $switch \leftarrow \{\mathtt{login}, \mathtt{link}\}$        ▷ Non-deterministically decide whether to log in or to follow some link
if $switch \equiv \mathtt{login}$ then        ▷ Log in to the AS
    let $url := \mathsf{GETURL}(tree, docnonce)$
    if $\mathtt{request} \in scriptstate$ then        ▷ redirect interaction finish mode is used
        let $url' := \langle \mathsf{URL}, \mathsf{S}, url.\mathtt{host}, \mathtt{/redirectLogin}, \langle\rangle\rangle$
        let $formData := scriptstate$        ▷ Contains redirect request identifier + referrer
    else        ▷ user code interaction start mode is used
        let $url' := \langle \mathsf{URL}, \mathsf{S}, url.\mathtt{host}, \mathtt{/userCodeLogin}, \langle\rangle\rangle$
        let $formData := scriptinputs$        ▷ Contains user code if client instance domain matched
    let $identity \leftarrow ids$
    let $secret \leftarrow secrets$
    let $formData[\mathtt{identity}] := identity$
    let $formData[\mathtt{password}] := secret$
    let $command := \langle \mathsf{FORM}, url', \mathsf{POST}, formData, \perp\rangle$
    stop $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$
else        ▷ Follow link
    let $protocol \leftarrow \{\mathsf{P}, \mathsf{S}\}$        ▷ Non-det. select protocol (HTTP or HTTPS)
    let $host \leftarrow \mathsf{Doms}$        ▷ Non-det. select host
    let $path \leftarrow \mathbb{S}$        ▷ Non-det. select path
    let $fragment \leftarrow \mathbb{S}$        ▷ Non-det. select fragment part
    let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$        ▷ Non-det. select parameters
    let $url := \langle \mathsf{URL}, protocol, host, path, parameters, fragment\rangle$        ▷ Assemble URL
    let $command := \langle \mathsf{HREF}, url, \perp, \perp\rangle$        ▷ Follow link to the selected URL
    stop $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$
```

A.13 Resource Servers

A resource server $rs \in \mathsf{RS}$ is a web server modeled as an atomic DY process $(I^{rs}, Z^{rs}, R^{rs}, s_0^{rs})$ with the addresses $I^{rs} := \mathsf{addr}(rs)$.

To verify MACs created by client instances, the RSs store *symmetric key records*:

Definition A.9. A *symmetric key record* is a term of the form

$$\langle instanceID, \langle keyID, key\rangle\rangle$$

with $instanceID \in \mathbb{S}$, $keyID \in \mathsf{KeyIDs}$, and $key \in K_{\mathrm{KP}}$.

Symmetric key records are used to store the symmetric keys of the client instances registered with the various ASs that *rs* is configured to use.

Next, we define the set Z^{rs} of states of *rs* and the initial state s_0^{rs} of *rs*.

Definition A.10. A state $s \in Z^{rs}$ of RS rs is a term of the form $\langle$*DNSaddress*, *pendingDNS*, *corrupt*, *pendingRequests*, *keyMapping*, *tlskeys*, *authServers*, *symKeys*, *signingKeyID*, *signingKey*, *mtlsKey*, *identities*, *instanceIDs*, *userResources*, *clientResources*, *resourceRequests*, *mtlsRequests*, *sigNonces*$\rangle$ with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $pendingRequests \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $tlskeys \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$, $authServers \in \mathcal{T}_{\mathcal{N}}$, $symKeys \in \left[\mathsf{Doms} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$, $signingKeyID \in \mathcal{N}$, $signingKey \in \mathcal{N}$, $mtlsKey \in \mathcal{N}$, $identities \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $instanceIDs \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $userResources \in \left[\mathsf{ID} \times \mathcal{N}\right]$, $clientResources \in \left[\mathsf{Doms} \times \left[\mathbb{S} \times \mathcal{N}\right]\right]$, $resourceRequests \in \left[\mathcal{N} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$, $mtlsRequests \in \left[\mathcal{N} \times \mathcal{N}\right]$, and $sigNonces \in \mathcal{T}_{\mathcal{N}}$.

An *initial state s_0^{rs} of rs* is a state of *rs* with $s_0^{rs}.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^{rs}.\mathtt{corrupt} \equiv \bot$, $s_0^{rs}.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^{rs}.\mathtt{keyMapping}$ being the same as the keymapping for browsers, $s_0^{rs}.\mathtt{tlskeys} \equiv tlskeys^{rs}$, and $s_0^{rs}.\mathtt{resourceRequests} \equiv s_0^{rs}.\mathtt{mtlsRequests} \equiv s_0^{rs}.\mathtt{sigNonces} \equiv \langle\rangle$.

resourceRequests will store various information about ongoing requests. The different requests are distinguished by a nonce called *requestID*, which acts as a key for the outer dictionary.

mtlsRequests and *sigNonces* work the same way as for the authorization servers in Appendix A.12.

$s_0^{rs}.\mathtt{authServers}$ is a sequence of domains representing the ASs that *rs* is configured to use. *rs* thus manages resources for the ASs in $s_0^{rs}.\mathtt{authServers}$ and sends introspection requests to them. For all domains $d \in^{\langle\rangle} s_0^{rs}.\mathtt{authServers}$ there must be an AS $as \in \mathsf{AS}$ with $d \in \mathsf{dom}(as)$. To simplify the following notations and algorithms, we assume that each RS uses each AS under only one domain, i.e., for all domains $d, d' \in^{\langle\rangle} s_0^{rs}.\mathtt{authServers}, d \not\equiv d'$ it holds that $\mathsf{dom}^{-1}(d) \neq \mathsf{dom}^{-1}(d')$.

$s_0^{rs}.\mathtt{symKeys}$ is used to store the symmetric keys of the client instances registered with the ASs in $s_0^{rs}.\mathtt{authServers}$. GNAP only requires that an RS must be able to dereference key references (subsumed with instance identifiers in our model) provided by the client instances. However, the protocol does not specify how this dereferencing should work. Since symmetric keys cannot be transmitted from an AS to an RS as part of token introspection (because GNAP allows arbitrary servers to use the token introspection endpoint and thus the symmetric keys could otherwise be leaked to arbitrary servers via token introspection), we store the keys in $s_0^{rs}.\mathtt{symKeys}$ using symmetric key records. The keys of the outer dictionary are the domains of the various ASs that *rs* is configured to use. For each domain $d \in s_0^{rs}.\mathtt{symKeys}$ it must hold that $d \in^{\langle\rangle} s_0^{rs}.\mathtt{authServers}$. The inner dictionaries, which are the values of the outer dictionary, consist of symmetric key records.

For each client instance $c \in \mathsf{CI}$, each domain $d \in^{\langle\rangle} s_0^{c}.\mathtt{authServers}$, and each key record $r \in^{\langle\rangle} s_0^{c}.\mathtt{keyRecords}[d]$ with $r.\mathtt{method} \equiv \mathtt{mac} \wedge r.\mathtt{rs} \in \mathsf{dom}(rs)$ there must be exactly one symmetric key record $r' \in^{\langle\rangle} s_0^{rs}.\mathtt{symKeys}[d']$ for the $d' \in^{\langle\rangle} s_0^{rs}.\mathtt{authServers}$ which is in $\mathsf{dom}(\mathsf{dom}^{-1}(d))$ (if such a d' exists) such that

$$\begin{aligned}&r'.\mathtt{instanceID} \equiv r.\mathtt{instanceID}\\ \wedge\, &r'.2.\mathtt{keyID} \equiv r.\mathtt{keyID}\\ \wedge\, &r'.2.\mathtt{key} \equiv r.\mathtt{key}\ .\end{aligned}$$

For all domains $d \in s_0^{rs}.\mathtt{symKeys}$, there should be no symmetric key records in $s_0^{rs}.\mathtt{symKeys}[d]$ other than those mentioned above.

$s_0^{rs}.\mathtt{signingKeyID} \in \mathsf{keyIDs}$ represents the key ID for the key $s_0^{rs}.\mathtt{signingKey}$.

$s_0^{rs}.\mathtt{signingKey}$ is a key in K_{KP} that must initially be stored in *rs* only. It will be used by *rs* to sign its introspection requests to the ASs, if MTLS is not used.

$s_0^{rs}.\mathtt{mtlsKey}$ is a key in K_{KP} that must initially be stored in *rs* only. It will be used by *rs* for MTLS key proofs for its introspection requests to the ASs, if signatures are not used.

In $s_0^{rs}.\mathtt{identities}$ *rs* stores the identities for which it stores resources. The keys are domains of the ASs *rs* is configured to use. The values are sequences of identities. For each domain $d \in s_0^{rs}.\mathtt{identities}$ it must hold that $d \in^{\langle\rangle} s_0^{rs}.\mathtt{authServers}$. For each identity $i \in^{\langle\rangle} s_0^{rs}.\mathtt{identities}[d]$ it must hold that there is a user record $r \in^{\langle\rangle} s_0^{\mathsf{dom}^{-1}(d)}.\mathtt{users}$ such that $i \equiv r.\mathtt{identity}$.

$s_0^{rs}.\mathtt{instanceIDs}$ stores the instance identifiers of the client instances for which *rs* stores resources. The keys are domains of the ASs that *rs* is configured to use. The values are sequences of instance identifiers used by the AS that belongs to the domain used as key. For each domain $d \in s_0^{rs}.\mathtt{instanceIDs}$ it must hold that $d \in^{\langle\rangle} s_0^{rs}.\mathtt{authServers}$. For each instance identifier $i \in^{\langle\rangle} s_0^{rs}.\mathtt{instanceIDs}[d]$ it must hold that there is a client registration record $r \in^{\langle\rangle} s_0^{\mathsf{dom}^{-1}(d)}.\mathtt{registrations}$ such that $i \equiv r.\mathtt{instanceID}$. If $r.\mathtt{keyData}.\mathtt{method} \equiv \mathtt{mac}$ it must additionally hold that $i \in s_0^{rs}.\mathtt{symKeys}[d]$. This is because when using symmetric keys, only the RS with which the client instance shares its symmetric key can manage resources for that client instance (when using i as instance ID), since the other RSs do not know the client instance's symmetric key and thus cannot validate key proofs of that client instance.

$s_0^{rs}.\mathtt{userResources}$ contains the nonces representing the resources *rs* manages for specific identities. $s_0^{rs}.\mathtt{userResources}$ maps identities to nonces in $\mathsf{ProtectedResources}$. All nonces in $\mathsf{ProtectedResources}$ that are subterms of $s_0^{rs}.\mathtt{userResources}$ must only be stored in *rs* initially. For each $d \in s_0^{rs}.\mathtt{identities}$ and each $i \in^{\langle\rangle} s_0^{rs}.\mathtt{identities}[d]$ there must be a nonce

$n \in \mathsf{ProtectedResources}$ such that $s_0^{rs}.\mathtt{userResources}[i] \equiv n$. For each $i \in s_0^{rs}.\mathtt{userResources}$ there must be a domain $d \in s_0^{rs}.\mathtt{identities}$ such that $i \in^{\langle\rangle} s_0^{rs}.\mathtt{identities}[d]$. For all $i, i' \in s_0^{rs}.\mathtt{userResources}$, $i \not\equiv i'$ we require that $s_0^{rs}.\mathtt{userResources}[i] \not\equiv s_0^{rs}.\mathtt{userResources}[i']$.

$s_0^{rs}.\mathtt{clientResources}$ contains the nonces representing the resources *rs* manages for the client instances that are registered at the ASs that *rs* is configured to use. $s_0^{rs}.\mathtt{clientResources}$ maps domains representing ASs to dictionaries that map the instance identifiers used by the AS to nonces in ProtectedResources. All nonces in ProtectedResources that are subterms of $s_0^{rs}.\mathtt{clientResources}$ must only be stored in *rs* initially. For each $d \in s_0^{rs}.\mathtt{instanceIDs}$ and each $i \in^{\langle\rangle} s_0^{rs}.\mathtt{instanceIDs}[d]$ there must be a nonce $n \in \mathsf{ProtectedResources}$ such that $s_0^{rs}.\mathtt{clientResources}[d][i] \equiv n$. For all $d \in s_0^{rs}.\mathtt{clientResources}$ and $i \in s_0^{rs}.\mathtt{clientResources}[d]$ it must hold that $i \in^{\langle\rangle} s_0^{rs}.\mathtt{instanceIDs}[d]$. For each client instance to have its own resource, the following must apply: For all domains $d \in s_0^{rs}.\mathtt{clientResources}$ and all identities $i, i' \in s_0^{rs}.\mathtt{clientResources}[d]$, with $i \not\equiv i'$ we require that $s_0^{rs}.\mathtt{clientResources}[d][i] \not\equiv s_0^{rs}.\mathtt{clientResources}[d][i']$. Furthermore, for all domains $d, d' \in s_0^{rs}.\mathtt{clientResources}$, $d \not\equiv d'$, every identity $i \in s_0^{rs}.\mathtt{clientResources}[d]$, and every identity $i' \in s_0^{rs}.\mathtt{clientResources}[d']$ it must hold that:
$s_0^{rs}.\mathtt{clientResources}[d][i] \not\equiv s_0^{rs}.\mathtt{clientResources}[d'][i']$.

There must not be a nonce $n \in \mathsf{ProtectedResources}$ that is a subterm of both $s_0^{rs}.\mathtt{userResources}$ and $s_0^{rs}.\mathtt{clientResources}$.

Since we allow *rs* to use multiple ASs in our modeling, *rs* must be able to determine which of the ASs in $s_0^{rs}.\mathtt{authServers}$ to use for token introspection for a given access token. Since determining this is out of scope for GNAP, we define the function is_issuer to determine the issuer of an access token in a state S of a configuration (S, E, N) of a run as follows:

Definition A.11. Given a nonce n and a domain d,

$$\mathsf{is\_issuer}(n, d) \equiv \top \Leftrightarrow n \in S(\mathsf{dom}^{-1}(d)).\mathtt{tokenBindings}\,.$$

is_issuer can be used by all processes and is the only such function in our model.

We now specify the relation R^{rs}: This relation is again based on the generic HTTPS server model defined in [29]. Table 4 shows a list of all placeholders used in the algorithms.

	Usage
ν_1	new request identifier used as key for the *resourceRequests* subterm
ν_2	new HTTP request nonce
ν_3	new nonce for MTLS
ν_4	new protected resource for a client instance
ν_5	new HTTP request nonce

Table 4: List of placeholders used in the RS algorithms.

The following algorithms are used for modeling the resource servers:

- Algorithm A.20 accepts requests to *rs*. If *rs* receives a request for a resource, *rs* first uses is_issuer to determine to which of the ASs in the *authServers* subterm it must send the introspection request to. Then it sends the introspection request to this AS, where it is non-deterministically decided whether a signature or MTLS is used as key proofing method. MACs are not modeled here, since *rs* would have to be registered with the AS in order for them to have shared symmetric keys. However, GNAP also allows arbitrary RSs to send introspection requests, so we do not model RSs that are pre-registered at an AS. The information required to process the response to the introspection request is stored in the *resourceRequests* subterm.
- Algorithm A.21 processes responses received by *rs*. If *rs* receives an introspection response, it checks that the authentication scheme used by the client instance in its resource request is correct, and if dealing with a key-bound access token, it validates the key proof. If the key proof was validated successfully or a valid bearer token was used, the associated resource is returned from either the *userResources* subterm or the *clientResources* subterm, depending on whether software-only authorization was used.

Algorithm A.20 Relation of an RS R^{rs}: Processing HTTPS requests.

```
function PROCESS_HTTPS_REQUEST($m, k, a, f, s'$)
    if $m$.path $\equiv$ /resource $\wedge$ $m$.method $\equiv$ GET then                ▷ Client instance wants to receive a resource
        let $type$, $accessToken$ such that $\langle type, accessToken\rangle \equiv m$.headers[Authorization] if possible; otherwise stop
        let $requestID := \nu_1$
        let $domainAS \leftarrow s'$.authServers such that is_issuer($accessToken$, $domainAS$) $\equiv \top$ if possible; otherwise stop
        let $introspectionEndpoint := \langle$URL, S, $domainAS$, /introspect, $\langle\rangle, \bot\rangle$
        let $s'$.resourceRequests[$requestID$] := [request:$m$, key:$k$, receiver:$a$, sender:$f$, type:$type$, AS:$domainAS$]
        let $method \leftarrow$ {sign, mtls}                ▷ Key proofing method for introspection request
        if $method \equiv$ sign then
            let $keyEntry$ := [keyID:$s'$.signingKeyID, key:pub($s'$.signingKey), method:sign]
            let $body$ := [accessToken:$accessToken$[value], RS:$keyEntry$]
            let $reference$ := [responseTo:introspection, requestID:$requestID$]
            call SIGN_AND_SEND(POST, $introspectionEndpoint$, $s'$.signingKeyID, $s'$.signingKey, $\bot$, $\bot$, $body$, $reference$, $s'$, $a$)
        else
            let $keyEntry$ := [key:pub($s'$.mtlsKey), method:mtls]
            let $body$ := [accessToken:$accessToken$[value], RS:$keyEntry$]
            let $s'$.resourceRequests[$requestID$][body] := $body$
            let $request := \langle$HTTPReq, $\nu_2$, POST, $domainAS$, /MTLS-prepare, $\langle\rangle$, $\langle\rangle$, [publicKey:pub($s'$.mtlsKey)]$\rangle$
            let $reference$ := [responseTo:MTLS, requestID:$requestID$]
            call HTTPS_SIMPLE_SEND($reference$, $request$, $s'$, $a$)
    else if $m$.path $\equiv$ /MTLS-prepare $\wedge$ $m$.method $\equiv$ POST then
        let $mtlsNonce := \nu_3$
        if publicKey $\in m$.body then
            let $clientKey := m$.body[publicKey]
        else stop                ▷ $clientKey$ is missing
        let $s'$.mtlsRequests := $s'$.mtlsRequests $+^{\langle\rangle}$ $\langle mtlsNonce, clientKey\rangle$
        let $m' := \mathsf{enc_s}(\langle$HTTPResp, $m$.nonce, 200, $\langle\rangle$, $\mathsf{enc_a}(\langle mtlsNonce, s'$.keyMapping[$m$.host]$\rangle$, $clientKey$)$\rangle$, $k$)
        stop $\langle\langle f, a, m'\rangle\rangle, s'$
    else stop                ▷ Unsupported operation
```

Algorithm A.21 Relation of an RS R^{rs}: Processing HTTPS responses.

```
function PROCESS_HTTPS_RESPONSE($m$, $reference$, $request$, $a$, $f$, $s'$)
    let $requestID := reference[\texttt{requestID}]$
    if $reference[\texttt{responseTo}] \equiv \texttt{introspection}$ then
        let $response := m.\text{body}$
        if $response[\texttt{active}] \equiv \bot$ then stop                                ▷ Access token is invalid
        else
            let $resourceReq := s'.\texttt{resourceRequests}[requestID][\texttt{request}]$
            let $type := s'.\texttt{resourceRequests}[requestID][\texttt{type}]$
            let $domainAS := s'.\texttt{resourceRequests}[requestID][\texttt{AS}]$
            if $response[\texttt{flags}] \not\equiv \texttt{bearer}$ then                    ▷ Access token is bound to a specific key
                if $type \not\equiv \texttt{GNAP}$ then stop                                 ▷ Wrong authentication scheme was used
                if $\texttt{instanceID} \in response$ then                                   ▷ A MAC must be validated
                    if $response[\texttt{instanceID}] \in s'.\text{symKeys}[domainAS]$ then
                        let $\langle keyID, key\rangle := s'.\text{symKeys}[domainAS][response[\texttt{instanceID}]]$
                        let $s' :=$ VALIDATE_KEY_PROOF($\texttt{mac}$, $resourceReq$, $keyID$, $key$, $s'$)
                    else stop                                                                ▷ $rs$ does not know the symmetric key
                else
                    let $method := response[\texttt{key}][\texttt{method}]$
                    let $key := response[\texttt{key}][\texttt{key}]$
                    if $method \equiv \texttt{sign}$ then
                        let $keyID := response[\texttt{key}][\texttt{keyID}]$
                        let $s' :=$ VALIDATE_KEY_PROOF($\texttt{sign}$, $resourceReq$, $keyID$, $key$, $s'$)
                    else if $method \equiv \texttt{mtls}$ then
                        let $s' :=$ VALIDATE_KEY_PROOF($\texttt{mtls}$, $resourceReq$, $\bot$, $key$, $s'$)
                    else stop                                                                ▷ Unsupported method
            else                                                                             ▷ Access token is a bearer token
                if $type \not\equiv \texttt{Bearer}$ then stop                               ▷ Wrong Authentication scheme was used
                if $\texttt{key} \in response$ then stop                                     ▷ For a bearer token no key may be included
        if $\texttt{identity} \in response[\texttt{access}]$ then
            let $identity := response[\texttt{access}][\texttt{identity}]$                   ▷ Identity of the RO
            if $identity \not\in^{\langle\rangle} s'.\texttt{identities}[domainAS]$ then stop ▷ no resources stored for this RO or $identity$ not managed by this AS
            let $resource := s'.\texttt{userResources}[identity]$
        else if $\texttt{instanceID} \in response[\texttt{access}]$ then
            let $instanceID := response[\texttt{access}][\texttt{instanceID}]$
            if $instanceID \not\in^{\langle\rangle} s'.\texttt{instanceIDs}[domainAS]$ then ▷ $rs$ does not yet store resources for this client instance
                let $resource := v_4$
                let $s'.\texttt{instanceIDs}[domainAS] := s'.\texttt{instanceIDs}[domainAS] +^{\langle\rangle} instanceID$
                let $s'.\texttt{clientResources}[domainAS][instanceID] := resource$
            else                                                                             ▷ $rs$ already stores a resource for this client instance
                let $resource := s'.\texttt{clientResources}[domainAS][instanceID]$
        else stop                                                                            ▷ Invalid response
        let $responseKey := s'.\texttt{resourceRequests}[requestID][\texttt{key}]$
        let $sender := s'.\texttt{resourceRequests}[requestID][\texttt{sender}]$
        let $receiver := s'.\texttt{resourceRequests}[requestID][\texttt{receiver}]$
        let $m' := \mathsf{enc_s}(\langle \text{HTTPResp}, resourceReq.\text{nonce}, \texttt{200}, \langle\rangle, resource\rangle, responseKey)$
        stop $\langle\langle sender, receiver, m'\rangle\rangle, s'$
    This algorithm is continued on the next page.
```

Continuation of Algorithm A.21 (RS PROCESS_HTTPS_RESPONSE)

```
    else if $reference[\mathtt{responseTo}] \equiv \mathtt{MTLS}$ then
        let $m_{\text{dec}} := \mathsf{dec_a}(m.\mathsf{body}, s'.\mathtt{mtlsKey})$
        let $mtlsNonce, pubKey$ such that $\langle mtlsNonce, pubKey \rangle \equiv m_{\text{dec}}$ if possible; otherwise stop
        if $pubKey \equiv s'.\mathtt{keyMapping}[request.\mathsf{host}]$ then        ▷ Send nonce only to the process that created it
            let $domainAS := s'.\mathtt{resourceRequests}[requestID][\mathtt{AS}]$
            let $body := s'.\mathtt{resourceRequests}[requestID][\mathtt{body}]$
            let $body[\mathtt{mtlsNonce}] := mtlsNonce$
            let $introspectionRequest := \langle \mathsf{HTTPReq}, v_5, \mathtt{POST}, domainAS, \mathtt{/introspect}, \langle\rangle, \langle\rangle, body \rangle$
            let $ref := [\mathtt{responseTo}{:}\mathtt{introspection}, \mathtt{requestID}{:}requestID]$
            call $\mathsf{HTTPS\_SIMPLE\_SEND}(ref, introspectionRequest, s', a)$
        else stop
```

B GENERAL DEFINITIONS

Definition B.1 (Sending Requests). We say that a DY process p *sent* a request $r \in \mathsf{HTTPRequests}$ (at some point) in a run if p emitted an event $\langle x, y, \mathsf{enc_a}(\langle r, k \rangle, k') \rangle$ in some processing step for some addresses x, y, some $k \in \mathcal{N}$, and some $k' \in \mathcal{T}_{\mathcal{N}}$.

Definition B.2 (End user attempted to authenticate at an AS). For a run ρ of a GNAP web system $\mathcal{GWS}$ we say that the end user of the browser b attempted to authenticate to an authorization server as using an identity u in a GNAP flow identified by a nonce $CIgid$ at the client instance c if there is a processing step Q in ρ with

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

(for some S, S', E, E', N, N') in which the browser b was triggered, selected a document loaded from an origin of as, executed the script *script_as_login* in that document, and in that script, in Line 10 of Algorithm A.19, selected the identity u. If the *scriptstate* of that document, when triggered, contained the key `request`, let $s \equiv$ *scriptstate*[`request`]. Otherwise, let $s' \equiv$ *scriptinputs*[`userCode`]. With $ASgid$ as the grant ID of as, for which $S(as)$.`grantRequests`[$ASgid$][`redirectNonce`] $\equiv s$ respectively $S(as)$.`grantRequests`[$ASgid$][`userCode`] $\equiv s'$, c is the client instance that sent the request m that led to the creation of $ASgid$ in Line 3 of Algorithm A.13. $CIgid$ is the grant ID of c that was created in Line 9 of Algorithm A.6 in the processing step in which m was sent by c. We then write $\mathsf{tryLogin}^{Q}_{\rho}(b, c, u, as, CIgid, ASgid)$.

Definition B.3 (End user successfully authenticated at an AS). For a run ρ of a GNAP web system $\mathcal{GWS}$, we say that the end user of the browser b successfully authenticated to an authorization server as using an identity u in a GNAP flow identified by a nonce $CIgid$ at the client instance c and a nonce $ASgid$ at as if there is a processing step Q in ρ with

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

such that:

- At some step Q' before Q, we have that $\mathsf{tryLogin}^{Q'}_{\rho}(b, c, u, as, CIgid, ASgid)$ holds.
- In the step Q, as calls `FINISH_INTERACTION`, with $ASgid$ as the first argument and the message m sent by b in step Q' as the second argument.
- In the step Q, `FINISH_INTERACTION` successfully returns, outputting at least one event. In particular, Line 17 of Algorithm A.15 gets executed, and the state change that it causes is saved.

We then write $\mathsf{finishLogin}^{Q}_{\rho}(b, c, u, as, CIgid, ASgid)$.

Definition B.4 (End user started a GNAP flow). For a run ρ of a GNAP web system $\mathcal{GWS}$ we say that the end user of the browser b started a GNAP flow identified by a nonce gid at the client instance c using the AS as in a processing step Q in ρ if

(1) in that processing step, the browser b was triggered, selected a document loaded from an origin of c, executed the script *script_ci_index* in that document, and in that script, chose a domain of as in Line 4 and executed Line 6 of Algorithm A.12, and
(2) the request from b to the `/startGrant` endpoint of c resulting from (1) leads to the creation of gid in Line 9 of Algorithm A.6 at c.

We then write $\mathsf{started}^{Q}_{\rho}(b, c, gid, as)$.

Definition B.5 (Client Instance started GNAP flow). Suppose ρ is a run of a GNAP web system $\mathcal{GWS}$, and Q is a processing step in ρ from (S, E, N) to (S', E', N') in which a message m is emitted. We say that a client c starts a GNAP flow at the AS as, identified by the nonces $CIgid$ and $ASgid$ on the client and AS side, respectively, written

$$\mathsf{client\_started}^{Q}_{\rho}(c, CIgid, m, as, ASgid),$$

if:

(1) c sent an HTTP request m to the `/requestGrant` path of as
(2) The reference for m contains the mappings `responseTo` : `grantResponse` and `grantID` : $CIgid$.
(3) When processing m, as chooses $grantID := ASgid$ in Line 3 of Algorithm A.13.

C PROPERTIES OF CORE WIM

The following lemma was adapted from [21].

Lemma C.1 (Host of HTTP Request). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, and every process $p \in \mathsf{CI} \cup \mathsf{AS} \cup \mathsf{RS}$ that is honest in S it holds true that if the generic HTTPS server calls $\mathsf{PROCESS\_HTTPS\_REQUEST}(m_{dec}, k, a, f, s')$ in Algorithm 18 of the WIM [29], then $m_{dec}.\mathsf{host} \in \mathsf{dom}(p)$, for all values of k, a, f, and s'.*

Proof. For the proof we refer to [21] as it is the same. □

Lemma C.2 (Every response processed has a matching request.). *Let ρ be a run of a web system $\mathcal{WS}$, and p be an honest instance of the generic HTTPS server model.*

Suppose that:

(1) In a processing step $s_i \rightarrow s_{i+1}$, p accepts a response resp and executes $\mathsf{PROCESS\_HTTPS\_RESPONSE}(resp, ref, req, \_, f, \_)$ for some ref, req, f.
(2) The instance model for p does not read or write the `pendingRequests` or `pendingDNS` subterms of its state.
(3) The instance model for p only emits messages in the procedure HTTPSRequests via HTTPS_SIMPLE_SEND.
(4) The instance model for p does not call the procedure PROCESS_HTTPS_RESPONSE.

Then, p previously called $\mathsf{HTTPS\_SIMPLE\_SEND}(ref, req, \_, \_)$, and $req.\mathsf{nonce} \equiv resp.\mathsf{nonce}$.

Let q be the process such that $f \in I^q$, and let r be the process such that $req.\mathsf{host} \in \mathsf{dom}(r)$. If p and r satisfy the preconditions of Lemma 4 of [29] (Note that (I) is given by the above, while (IV) and (V) are implied by preconditions of this lemma), then we also have that the response resp was created by the process r that the corresponding request was sent to.

Proof. This proof refers to Algorithms 18 and 13 in [29].

If PROCESS_HTTPS_RESPONSE is called by an honest instance of the generic HTTPS server model, and the instance model does not call this function, then the check on Line 21 of Algorithm 18 must succeed, and so *resp*.nonce ≡ *req*.nonce, giving the second condition that we wish to verify.

Additionally, the check on Line 19 must have also succeeded, meaning in particular that *resp* was received encrypted with some key k stored in pendingRequests. The entry in pendingRequests containing k (and also *ref* and *req*) can only have been written on Line 15, as p does not modify pendingRequests except in the core model. This line is only reached upon receipt of a DNS response (Line 10) whose nonce occurs in pendingDNS, and the corresponding entry in pendingDNS is used to load *ref* and *req* before storing them into pendingRequests. This entry in pendingDNS can only be written on Line 2 in HTTPS_SIMPLE_SEND (Algorithm 13). Here, the *ref* and *req* stored are passed in as arguments to HTTPS_SIMPLE_SEND, and so we have, as desired, that p must have called HTTPS_SIMPLE_SEND(*ref*, *req*, _, _).

Now, if the conditions of Lemma 4 are satisfied, we can simply invoke Lemma 4, case (4) and get that r must be the creator of *resp*. □

As our model of GNAP generally avoids breaking the abstractions of the WIM, the implementations of client instances and resource servers both satisfy the preconditions of Lemma 4 of [29]. Most of these preconditions can be checked by simply observing that our model does not read or write particular subterms of its state that are used by the underlying WIM. The remaining preconditions are also easily checked, but slightly less immediate. For instance, clients and resource servers do not emit messages in HTTPSRequests because all HTTPS requests are sent via HTTPS_SIMPLE_SEND, a function of the WIM, rather than of the instance model built on top of the WIM.

Similarly, the preconditions of Lemma C.2 are all satisfied in the case where the process p is a client instance or a resource server. Note that in Algorithm A.15, the authorization server model modifies the pendingDNS field of its state, so the preconditions of this lemma are not met in the case of authorization servers. However, we do not use either of these lemmas in the case of authorization servers, as the authorization server does not expect a response to the requests it sends out.

For all request/response pairs in our model, then, we can apply both Lemma C.2 and Lemma 4 of [29]. We will use these lemmas implicitly through the rest of this document, in order to justify that every response has a corresponding request (related to it via the reference associated with the request), and that the response comes from the same process to which the request is sent.

D GENERAL PROPERTIES

D.1 Things that don't leak

Lemma D.1 (Private Keys of Client Instances do not leak). *For any run ρ of a GNAP web system $\mathcal{GWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$, every configuration (S, E, N) in ρ, every $c \in \mathsf{CI}$ that is honest in S, every domain dmn $\in S(c).\mathtt{keyRecords}$, every key record $r \in^{\langle\rangle} S(c).\mathtt{keyRecords}[dmn]$ with $r.\mathtt{method} \equiv \mathtt{sign} \vee r.\mathtt{method} \equiv \mathtt{mtls}$, and every process $p \in \mathcal{W} \setminus \{c\}$ it holds true that $r.\mathtt{key} \notin d_\emptyset(S(p))$.*

Proof. There is no code section in which the value of r.method or the value of r.key could change. Thus, it must hold that these values are unchanged since the initial state. By the definitions of the initial states of the processes, it must hold that for all processes $p \in \mathcal{W} \setminus \{c\}$ the nonce r.key appears only as a public key in s_0^p. As the equational theory does not allow the extraction of a private key x from a public key $\mathsf{pub}(x)$, it must hold that $r.\mathtt{key} \notin d_\emptyset(s_0^p)$ for all $p \in \mathcal{W} \setminus \{c\}$. Thus, for p to know r.key in S, there must have been a processing step in which c leaked r.key to another process. Whenever r.key is a subterm of a message emitted by c, the public key to that private key, i.e. $\mathsf{pub}(r.\mathtt{key})$ is sent since there is no code section in which an honest client instance sends a private key. However, since $\mathsf{pub}(r.\mathtt{key})$ cannot be used to derive r.key, $r.\mathtt{key} \notin d_\emptyset(S(p))$ must hold for all $p \in \mathcal{W} \setminus \{c\}$. □

Lemma D.2 (Private Keys of Resource Servers do not leak). *For any run ρ of a GNAP web system $\mathcal{GWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$, every configuration (S, E, N) in ρ, every $rs \in \mathsf{RS}$ that is honest in S, and every process $p \in \mathcal{W} \setminus \{rs\}$ it holds true that $S(rs).\mathtt{signingKey} \notin d_\emptyset(S(p)) \wedge S(rs).\mathtt{mtlsKey} \notin d_\emptyset(S(p))$.*

Proof. There is no code section in which the value of the keys in $s_0^{rs}.\mathtt{signingKey}$ and $s_0^{rs}.\mathtt{mtlsKey}$ could change. Thus, it must hold that these keys are unchanged since the initial state. That is $s_0^{rs}.\mathtt{signingKey} \equiv S(rs).\mathtt{signingKey}$ and analogously $s_0^{rs}.\mathtt{mtlsKey} \equiv S(rs).\mathtt{mtlsKey}$. By the definitions of the initial states of the processes, the nonces $s_0^{rs}.\mathtt{signingKey}$ and $s_0^{rs}.\mathtt{mtlsKey}$ are initially only stored in rs, which means $s_0^{rs}.\mathtt{signingKey} \notin d_\emptyset(s_0^p)$ and $s_0^{rs}.\mathtt{mtlsKey} \notin d_\emptyset(s_0^p)$ for all $p \in \mathcal{W} \setminus \{rs\}$. Thus, for p to know $s_0^{rs}.\mathtt{signingKey}$ or $s_0^{rs}.\mathtt{mtlsKey}$ in S, there must have been a processing step in which rs leaked $s_0^{rs}.\mathtt{signingKey}$ or $s_0^{rs}.\mathtt{mtlsKey}$ to another process. rs only includes $s_0^{rs}.\mathtt{signingKey}$ as a subterm of an emitted event in Line 10 of Algorithm A.20 and $s_0^{rs}.\mathtt{mtlsKey}$ in Line 15 of Algorithm A.20. However, in both lines only the corresponding public key is included ($\mathsf{pub}(s_0^{rs}.\mathtt{signingKey})$ and $\mathsf{pub}(s_0^{rs}.\mathtt{mtlsKey})$). The public keys cannot be used to derive the private keys $s_0^{rs}.\mathtt{signingKey}$ and $s_0^{rs}.\mathtt{mtlsKey}$ because the equational theory does not allow the extraction of a private key x from a public key $\mathsf{pub}(x)$. So it must hold that $S(rs).\mathtt{signingKey} \notin d_\emptyset(S(p)) \wedge S(rs).\mathtt{mtlsKey} \notin d_\emptyset(S(p))$ for all $p \in \mathcal{W} \setminus \{rs\}$. □

Lemma D.3 (Symmetric Keys do not leak). *For any run ρ of a GNAP web system $\mathcal{GWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$, every configuration (S, E, N) in ρ, every $c \in \mathsf{CI}$ that is honest in S, every domain dmn $\in S(c).\mathtt{keyRecords}$, every key record $r \in^{\langle\rangle} S(c).\mathtt{keyRecords}[dmn]$ with $r.\mathtt{method} \equiv \mathtt{mac}$, and every process $p \in \mathcal{W}$ it holds true that if $as = \mathsf{dom}^{-1}(dmn)$ is honest in S, $rs = \mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S, and $p \notin \{c, as, rs\}$, then $r.\mathtt{key} \notin d_\emptyset(S(p))$.*

Proof. Let $\bar{i}$ be an integer (used as a pointer) such that $S(c).\mathtt{keyRecords}[dmn].\bar{i} \equiv r$. Since keys stored in key records never change, we have $r.\mathtt{key} \equiv S(c).\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key} \equiv s_0^c.\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key}$. By definition, only c, as, and rs initially store the value of $s_0^c.\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key}$. This means it must hold that $s_0^c.\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key} \notin d_\emptyset(s_0^p)$ for all $p \notin \{c, as, rs\}$. Thus, for p to know r.key in S, there must have been a processing step in which c, as, or rs leaked r.key to another process. However, this is not possible because symmetric keys are used by client instances only to generate MACs, while they are used by honest authorization servers and honest resource servers only to validate these MACs. Therefore, there is no code section in which c emits $S(c).\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key}$, no code section in which as emits $S(as).\mathtt{registrations}[S(c).\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{instanceID}].\mathtt{key}$, and no code section in which rs emits $S(rs).\mathtt{symKeys}[dmn'][S(c).\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{instanceID}].\mathtt{key}$ (for some dmn') as a subterm of an event. Thus, $r.\mathtt{key}$ cannot be leaked to another process in any processing step, so it must hold that $r.\mathtt{key} \notin d_{\emptyset}(S(p))$. □

Lemma D.4 (Keys for Access Tokens do not leak). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every client instance $c \in \mathsf{CI}$ that is honest in S, every domain $d \in S(c).\mathtt{tokenKeys}$, every token key record $r \in^{\langle\rangle} S(c).\mathtt{tokenKeys}[d]$, and every process $p \in \mathcal{W} \setminus \{c\}$ it holds true that $r.\mathtt{key} \notin d_{\emptyset}(S(p))$.*

Proof. Similar to Lemma D.1, there is no place, where the private key $r.\mathtt{key}$ is sent out. Since initially $r.\mathtt{key}$ appears only as public key for other processes than c, the proof is complete. □

The following lemma and its proof are based on Lemma 7 from Fett et al. [22].

Lemma D.5 (MTLS Nonces do not leak to Third Parties). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every process $p \in \mathsf{AS} \cup \mathsf{RS}$ that is honest in S, every process $c \in \mathsf{CI} \cup \mathsf{RS}$ that is honest in S, every mtlsNonce created in Line 145 of Algorithm A.13 resp. Line 22 of Algorithm A.20 in consequence of a request m received at the* `/MTLS-prepare` *path of p that was sent by c, and every process p' with $p \neq p' \neq c$ it holds true that $mtlsNonce \notin d_{\emptyset}(S(p'))$.*

Proof. We start by showing that the *mtlsNonce* is sent by p only asymmetrically encrypted and only c knows the corresponding private key. In doing so, we distinguish whether p is an AS or an RS.

If p is an authorization server, it sends an *mtlsNonce* created in Line 145 of Algorithm A.13 only in Line 156, where it is asymmetrically encrypted with either the public key

$$\begin{aligned} & clientKey \\ \equiv\ & S(p).\mathtt{registrations}[instanceID].\mathtt{publicKey} \\ \equiv\ & S(p).\mathtt{registrations}[m.\mathtt{body}[\mathtt{instanceID}]].\mathtt{publicKey} \end{aligned}$$

(using Lines 150 and 147) or the public key $m.\mathtt{body}[\mathtt{publicKey}]$ (Line 152).

First, we look at the latter case. There are various code sections where the honest process c can send a message to the `/MTLS-prepare` path that contains the public key under the key `publicKey` in its body. Thereby the following values are sent:

(1) In Line 51 of Algorithm A.6:

$$\begin{aligned} & \mathsf{pub}(key) \\ \equiv\ & \mathsf{pub}(keyRecord.\mathtt{key}) \\ \equiv\ & \mathsf{pub}(S'(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{key}) \end{aligned}$$

(using Lines 41 and 27) for a previous state S', a domain $domainAS \in \mathsf{Doms}$, and an $\bar{i} \in \mathbb{N}$.

(2) In Line 71 of Algorithm A.7:

$$\begin{aligned} & \mathsf{pub}(keyRecord.\mathtt{key}) \\ \equiv\ & \mathsf{pub}(grantRequest[\mathtt{keyRecord}].\mathtt{key}) \\ \equiv\ & \mathsf{pub}(S'(c).\mathtt{grants}[grantID][\mathtt{keyRecord}].\mathtt{key}) \end{aligned}$$

(using Lines 6 and 3) for a previous state S' and a grant ID $grantID \in \mathcal{N}$. The `keyRecord` entry of $S'(c).\mathtt{grants}[grantID]$ must have been stored in Line 28 of Algorithm A.6 or Line 10 of Algorithm A.8. In both cases the stored value *keyRecord* is a key record from $S''(c).\mathtt{keyRecords}[domainAS]$ for a previous state S'' and a domain $domainAS \in \mathsf{Doms}$ (Line 27 of Algorithm A.6 resp. Line 5 of Algorithm A.8).

(3) In Line 31 of Algorithm A.10:

$$\begin{aligned} & \mathsf{pub}(keyRecord.\mathtt{key}) \\ \equiv\ & \mathsf{pub}(S'(c).\mathtt{grants}[grantID][\mathtt{keyRecord}].\mathtt{key}) \end{aligned}$$

(using Line 14) for a previous state S' and a grant ID $grantID \in \mathcal{N}$. Using the same reasoning as for the second point, the stored key record must again have been taken from $S''(c).\mathtt{keyRecords}[domainAS]$.

(4) In Line 18 of Algorithm A.20: $\mathsf{pub}(S'(c).\mathtt{mtlsKey})$ for a previous state S'.

In cases 1 to 3, $m.\mathtt{body}[\mathtt{publicKey}]$ equals the public key of a private key from a key record in $S'(c).\mathtt{keyRecords}[domainAS]$ for a previous state S' and a domain $domainAS \in \mathsf{Doms}$. By Lemma D.1, this private key can only be known to the honest process c. In case 4, the private key is the key used for MTLS by the RS c, which can only be known to c due to Lemma D.2.

Now let's look at the former case. There are two code sections where the honest process c can send a message to the `/MTLS-prepare` path of an AS p that contains the `instanceID` key. Thereby the following instance identifiers are sent:

(1) In Line 36 of Algorithm A.6:

$$\begin{aligned} & keyRecord.\mathtt{instanceID} \\ \equiv\ & S'(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{instanceID} \end{aligned}$$

(using Line 27) for a previous state S', a domain $domainAS \in \mathsf{Doms}$, and an integer $\bar{i} \in \mathbb{N}$.

(2) In Line 16 of Algorithm A.8:

$$\begin{aligned} & instanceID \\ \equiv\ & keyRecord.\mathtt{instanceID} \\ \equiv\ & S'(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{instanceID} \end{aligned}$$

(using Lines 8 and 5) for a previous state S', a domain $domainAS \in \mathsf{Doms}$, and an integer $\bar{i} \in \mathbb{N}$.

Thus, in both cases, the instance identifier sent by c is an instance identifier from a key record $r \in^{\langle\rangle} S'(c).\mathtt{keyRecords}[domainAS]$. Since *domainAS* is used as the host of c's request in both cases, it must hold that $domainAS \in \mathsf{dom}(p)$.

Note that key records are preconfigured in the client, and can only be updated once, to add an instance ID to a key record that was not yet registered at an AS. In particular, this means that a key record either matches the initial state or had its instance identifier set when registering the key at an AS.

If $r \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[\mathit{domainAS}]$ (c was pre-registered at p), it must hold by definition that

$$\begin{aligned} &S(p).\mathtt{registrations}[m.\mathtt{body}[\mathtt{instanceID}]].\mathtt{publicKey} \\ \equiv\ &S(p).\mathtt{registrations}[r.\mathtt{instanceID}].\mathtt{publicKey} \\ \equiv\ &\mathsf{pub}(r.\mathtt{key}) \end{aligned}$$

$r.\mathtt{key}$ can only be known to c due to Lemma D.1.

If $r \notin^{\langle\rangle} s_0^c.\mathtt{keyRecords}[\mathit{domainAS}]$, the instance identifier $r.\mathtt{instanceID}$ must have been set in Line 12 of Algorithm A.7. Since $\mathit{domainAS} \in \mathsf{dom}(p)$ and because of Lemma D.9[4], it must hold that $S(p).\mathtt{registrations}[m.\mathtt{body}[\mathtt{instanceID}]].\mathtt{publicKey} \equiv \mathsf{pub}(r.\mathtt{key})$. Again, $r.\mathtt{key}$ can only be known to c due to Lemma D.1.

If p is a resource server, it sends an *mtlsNonce* created in Line 22 of Algorithm A.20 only in Line 28, where it is asymmetrically encrypted with the public key $m.\mathtt{body}[\mathtt{publicKey}]$ (Line 24). Messages to the /MTLS-prepare path of an RS are only sent at the following lines:

(1) In Line 68 of Algorithm A.8:

$$\begin{aligned} &\mathsf{pub}(\mathit{key}) \\ \equiv\ &\mathsf{pub}(\mathit{keyRecord}.\mathtt{key}) \\ \equiv\ &\mathsf{pub}(S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{keyRecord}].\mathtt{key}) \end{aligned}$$

(using Lines 56 and 55) for a previous state S' and a grant ID $\mathit{grantID} \in \mathcal{N}$. Using the same reasoning as for case 2 of p being an AS, the stored key record must again have been taken from $S''(c).\mathtt{keyRecords}[\mathit{domainAS}]$.

(2) In Line 49 of Algorithm A.8: The public key used is $\mathsf{pub}(\mathit{privateKey}) \equiv \mathsf{pub}(\mathit{keyData}.\mathtt{key})$, where *keyData* is an entry of $S'(c).\mathtt{tokenKeys}[\mathit{domainAS}]$ (Lines 41 and 43) for a previous state S'.

As for the case of p being an authorization server, in case 1 the key is a public key of a private key from a key record, and hence by Lemma D.1 only c knows this key. In case 2, due to Lemma D.4, only c can know the private key associated with the public key.

We have now shown for all possible cases that the *mtlsNonce* sent by p can only be decrypted by c since only c can know the required private key. Now we show that c sends the received *mtlsNonce* back to p only. For this we show that after decrypting the *mtlsNonce* it is always sent to the same domain to which the request to the /MTLS-prepare path was sent that led to the receipt of the *mtlsNonce*. Thus, due to the use of HTTPS for all requests, the *mtlsNonce* can only be sent back to p.

The received *mtlsNonce* can be decrypted only in one of the following sections:

(1) Line 82 of Algorithm A.7:
In this section, the *mtlsNonce* is sent to $\mathit{domainAS} \equiv S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{AS}]$ (for a state S' and a grant ID *grantID* that is taken from the reference). The MTLS_GR reference type is used only in the Lines 53 and 38 of Algorithm A.6 and Line 18 of Algorithm A.8.
In Line 53 and 38 of Algorithm A.6, the request goes to *domainAS*, which is stored in Line 28 under $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{AS}]$. Here, S'' is a state before S' and *grantID* is the grant ID from the reference.
In Line 18 of Algorithm A.8, the request goes to *domainAS*, which in Line 10 is also stored in $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{AS}]$. S'' is again a state before S' and *grantID* is the grant ID from the reference.
It must hold that $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{AS}] \equiv S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{AS}]$, since this value is not overwritten anywhere. Thus, the *mtlsNonce* is sent to the same domain as the request to the /MTLS-prepare path.

(2) Line 92 of Algorithm A.7:
Here, the *mtlsNonce* is sent to the domain $\mathit{url}.\mathtt{host} \equiv S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{continueURL}].\mathtt{host}$ (for a state S' and a grant ID *grantID* that is taken from the reference). The MTLS_CR reference type is used only in Line 33 of Algorithm A.10. In this section, the request is sent to $\mathit{continueURL}.\mathtt{host}$ which has the same value as $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{continueURL}].\mathtt{host}$. Here, S'' is a state before S' and *grantID* is the grant ID used in the reference.
It must hold true that the continue URL stored in $S''(c)$ and $S'(C)$ are equivalent since the continueURL field is never overwritten after its initialization in Line 35 of Algorithm A.7. Thus, the *mtlsNonce* is sent to the same domain as the request to the /MTLS-prepare path.

(3) Line 109 of Algorithm A.7:
The *mtlsNonce* is sent to $\mathit{url}.\mathtt{host} \equiv S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{patchRequest}].3.\mathtt{host}$ (for a state S' and a grant ID *grantID* that is taken from the reference). The MTLS_PR reference type is used only in Line 73 of Algorithm A.7. In this section, the request goes to $\mathit{continueURL}.\mathtt{host}$, where *continueURL* is stored under $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{patchRequest}].3$ in Line 67. Here, S'' is a state before S' and *grantID* is the grant ID used in the reference.
It must hold true that the patch URLs stored in $S''(c)$ and $S'(c)$ are equivalent since the patchRequest value can only be overwritten by a new PATCH request, but c can send a new patch request only after it received a grant response for the previous PATCH request. Thus, the *mtlsNonce* is sent to the same domain as the request to the /MTLS-prepare path.

(4) Line 119 of Algorithm A.7:
In this code section, the request into which the *mtlsNonce* is inserted is loaded from the reference in Line 127. The request must have been inserted into the reference in Line 51 of Algorithm A.8 or in Line 70 of Algorithm A.8. In both cases, the host of the inserted request is *domainRS* (Line 48 resp. Line 67 of Algorithm A.8), which is also used as the host for the request to the /MTLS-prepare path in Line 50 resp. Line 69. Thus, the *mtlsNonce* is sent to the same domain as the request to the /MTLS-prepare path.

(5) Line 47 of Algorithm A.21:
The *mtlsNonce* here is sent to $\mathit{domainAS} \equiv S'(c).\mathtt{resourceRequests}[\mathit{requestID}][\mathtt{AS}]$ (for a state S' and a request ID *requestID* that is taken from the reference). The MTLS reference type is used only in Line 19 of Algorithm A.20. In this section, the request goes to *domainAS*, which is stored in Line 7 in $S''(c).\mathtt{resourceRequests}[\mathit{requestID}][\mathtt{AS}]$. Here,

[4] Note, that Lemma D.9 only depends on Lemma C.2 which in turn does not depend on any other lemma. Hence, we do not introduce a circular dependency at this point.

S'' is a state before S' and *requestID* is the request ID from the reference.

It must hold true that the URLs stored in $S''(c)$ and $S'(c)$ are equivalent since the AS value cannot be overwritten. Thus, the *mtlsNonce* is sent to the same domain as the request to the /MTLS-prepare path.

We have now shown for all cases that c sends the *mtlsNonce* only to the same domain of p to which it sent the request to the /MTLS-prepare path (using HTTPS for both requests). Since p is honest in S, p uses the received *mtlsNonce* only to validate the key proof and does not emit it as a subterm in any further events. So, in summary, the *mtlsNonce* is leaked by its creator p only to c and by c only to p. Thus, for all processes p' for which $p \neq p' \neq c$, it must hold that $mtlsNonce \notin d_\emptyset(S(p'))$. □

Lemma D.6 (Bearer Tokens do not leak). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every AS $as \in \mathsf{AS}$ that is honest in S, and every access token accessToken $\in S(as)$.*`tokenBindings` *with* $S(as)$.`tokenBindings`$[accessToken]$`[type]` $\equiv$ `bearer` *it holds true that if*

1. *the client instance c that sent the grant request that led to the creation of the grant ID grantID $\equiv S(as)$.*`tokenBindings`$[accessToken]$`[grantID]` *in Line 3 of Algorithm A.13 is honest in S,*
2. *if c used a key record r with r.*`method` $\equiv$ `mac` *for this grant request, it holds true that* $\mathsf{dom}^{-1}(r.\mathtt{rs})$ *is honest in S, and*
3. *all RSs rs* $\in \{rs \in \mathsf{RS} \mid \exists dmnRS \in^{\langle\rangle} S(c).\mathtt{grants}[grantID][\mathtt{bearerRSs}]\colon dmnRS \in \mathsf{dom}(rs)\}$ *are honest in S,*

then accessToken is only derivable by proccesses in $\{as, c\} \cup \{rs \in \mathsf{RS} \mid \exists dmnRS \in^{\langle\rangle} S(c).\mathtt{grants}[grantID][\mathtt{bearerRSs}]\colon dmnRS \in \mathsf{dom}(rs)\}$.

Proof. We first show that an access token created by *as* is sent by *as* only to *c*. New access tokens are created only in Line 6 of Algorithm A.17 (CREATE_GRANT_RESPONSE). The generated grant response is returned by Algorithm A.17 in Line 39 and then sent by *as* in one of the following lines of Algorithm A.13 (since only in these sections CREATE_GRANT_RESPONSE is called):

1. Line 48: grant response in response to a continuation request
2. Line 79: adjustment of the requested values via a PATCH request
3. Line 70 (CREATE_GRANT_RESPONSE is called via Algorithm A.16): adjustment of the requested values via a PATCH request including interaction finish
4. Line 59 (CREATE_GRANT_RESPONSE is called via Algorithm A.16): completion of interaction via a POST request

In the first case, it is obvious that the grant response is returned to *c*, since this response is sent directly in response to the grant request from *c*. In cases 2, 3, and 4, the grant response is sent in response to a continuation request, which by Lemma D.10[5] must have been sent by *c*. Thus, *as* sends the *accessToken* back to *c* in all cases. Due to the use of HTTPS it must hold true in all cases that only *c* can decrypt the response containing the *accessToken*. Thus, *as* does not leak the *accessToken* to any process other than *c*.

Since *c* is honest by precondition, *c* sends the obtained *accessToken* only when executing Algorithm A.8. Since $S(as)$.`tokenBindings`$[accessToken]$`[type]` $\equiv$ `bearer`, *as* must have set the value of *grantResponse*`[accessToken][flags]` to `bearer` in Line 26 of Algorithm A.17 (since $S(as)$.`tokenBindings`$[accessToken]$`[type]` cannot change). Thus, when executing Algorithm A.8, if *accessToken* is chosen in Line 30, it must hold that the if statement in Line 32 is true, so the *accessToken* can only be sent by *c* in Line 36. In this case, *accessToken* is sent to *domainRS*, which is added to $S'(c)$.`grants`$[grantID]$`[bearerRSs]` in Line 33 for some state S'. Since a domain can never be removed from $S'(c)$.`grants`$[grantID]$`[bearerRSs]`, it must hold that *c* sends *accessToken* only to RSs in $\{rs \in \mathsf{RS} \mid \exists dmnRS \in^{\langle\rangle} S(c).\mathtt{grants}[grantID][\mathtt{bearerRSs}]\colon dmnRS \in \mathsf{dom}(rs)\}$ (using HTTPS).

Since all rs in this set are honest by precondition, they send the *accessToken* after receiving the request from *c* only to the *domainAS* for which it holds that $\mathsf{is\_issuer}(accessToken, domainAS) \equiv \top$ (due to Line 5 of Algorithm A.20). However, since only honest RSs, *as*, and *c* are able to derive the *accessToken*, it must hold that $domainAS \in \mathsf{dom}(as)$, as by definition $\mathsf{is\_issuer}(accessToken, domainAS) \equiv \top \Leftrightarrow accessToken \in S(\mathsf{dom}^{-1}(domainAS)).\mathtt{tokenBindings}$ and *domainAS* must be a domain of an AS, since *domainAS* is chosen from the *authServers* subterm (Line 5 of Algorithm A.20).

Thus, the *accessToken* is sent only to *as* by all resource servers that received the access token from *c*, also using HTTPS. *as* uses an access token received via token introspection only to determine the associated entry of the *tokenBindings* subterm in Line 120 of Algorithm A.13 and then discards it.

Thus, in total, only processes in

$$\{rs \in \mathsf{RS} \mid \exists dmnRS \in^{\langle\rangle} S(c).\mathtt{grants}[grantID][\mathtt{bearerRSs}]\colon dmnRS \in \mathsf{dom}(rs)\} \cup \{as, c\}$$

are able to derive *accessToken* in *S*, so for all processes p not in that set it must hold that $accessToken \notin d_\emptyset(S(p))$ as long as the first three conditions from the lemma are given as well. □

Lemma D.7 (Passwords do not leak). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every AS $as \in \mathsf{AS}$ that is honest in S, every identity $u \in \mathsf{ID}^{as}$, and every process $p \notin \{as, \mathsf{ownerOfID}(u)\}$ it holds true that* $\mathsf{secretOfID}(u) \notin d_\emptyset(S(p))$ *as long as* $\mathsf{ownerOfID}(u)$ *is not fully corrupted in S.*

Proof. This proof is loosely based on the proof of Lemma 4 from [27]. Let $z = \mathsf{secretOfID}(u)$. According to the definitions of the initial states, z is initially stored only in $\mathsf{ownerOfID}(u)$ and *as*. *as* uses the passwords of its users only in Line 8 of Algorithm A.15 to check whether a submitted password matches the password of the provided identity. Since z is not used elsewhere by *as*, z cannot be leaked by *as* to another process during this process.

In our browser model, only scripts loaded from the origin $\langle dmnAS, \mathsf{S}\rangle$ for a domain $dmnAS \in \mathsf{dom}(as)$ can access z. Since *as* is honest, only *script_as_login* is eligible for this. Also, since $\mathsf{ownerOfID}(u)$ is not fully corrupted, it does not use or leak z in any other way.

[5] Note that Lemma D.10 depends on Lemma D.8 which in turn uses Lemmas D.1, D.3 and D.5. The first two don't have dependencies, and for Lemma D.5 we argued in a previous footnote that there are no circular dependencies.

If *script_as_login* was loaded and has access to z, it must have been loaded from an origin $\langle dmnAS, \mathsf{S}\rangle$ for a domain $dmnAS$ of as. The script sends z to $dmnAS$ in an HTTPS POST request. If $\mathsf{ownerOfID}(u)$ sends this request, as is the only party able to decrypt it due to the use of HTTPS. as uses the received password only for the aforementioned comparison in Line 8 of Algorithm A.15 and then discards it. $\mathsf{ownerOfID}(u)$ is then redirected to the client instance by as in Line 21 or Line 28. Since the 303 redirect status code is used in both cases, $\mathsf{ownerOfID}(u)$ drops the body of the POST request in the resulting request to the client instance and rewrites it to a GET request, so z is not leaked to the client instance with this redirect.

Thus, there is no way z could be leaked to a process p not in $\{as, \mathsf{ownerOfID}(u)\}$, which proves the lemma. □

D.2 Message Integrity

Lemma D.8 (Key Proofs authenticate the Signer and guarantee Integrity). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, and every process $p \in \mathsf{AS} \cup \mathsf{RS}$ that is honest in S it holds true that if p calls* $\mathsf{VALIDATE\_KEY\_PROOF}($*method*, *$m$, keyID, key, s'*$)$ *for some method* $\in \{\mathtt{sign}, \mathtt{mac}, \mathtt{mtls}\}$*, an HTTP request $m \in$* $\mathsf{HTTPRequests}$*, some keyID $\in \mathcal{N}$, some state s', and*

- *some key* $\equiv \mathsf{pub}(S(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{key})$ *or key* $\equiv \mathsf{pub}(S(c).\mathtt{tokenKeys}[domainAS].\bar{i}.\mathtt{key})$ *(if method* $\in \{\mathtt{sign}, \mathtt{mtls}\}$*) or key* $\equiv S(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{key}$ *(if method* $\equiv \mathtt{mac}$*) for some process $c \in$* CI*, some domainAS $\in$* Doms*, and some $\bar{i} \in \mathbb{N}$,*
- *that c is honest in S, and*
- *the call to* $\mathsf{VALIDATE\_KEY\_PROOF}$ *returns (i.e. it does not stop),*

then c previously sent an HTTP request m' with

(1) $m'.\mathtt{method} \equiv m.\mathtt{method}$*,*
(2) $m'.\mathtt{body} \equiv m.\mathtt{body}$*,*
(3) $m'.\mathtt{host} \equiv m.\mathtt{host}$*,*
(4) $m'.\mathtt{path} \equiv m.\mathtt{path}$*,*
(5) $m'.\mathtt{parameters} \equiv m.\mathtt{parameters}$*, and*
(6) $m'.\mathtt{headers}[\mathtt{Authorization}] \equiv m.\mathtt{headers}[\mathtt{Authorization}]$ *(if* $\mathtt{Authorization} \in m.\mathtt{headers}$*).*

If method $\equiv \mathtt{mac}$ *we additionally require that*
the AS $as = \mathsf{dom}^{-1}(domainAS)$ and
the RS $rs = \mathsf{dom}^{-1}(S(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{rs})$ are honest in S (this is already given for as or rs if $p = as$ respectively $p = rs$).

Proof. $\mathsf{VALIDATE\_KEY\_PROOF}$ (Algorithm A.5) only returns in Line 31. Line 31 is reached due to Line 30 only if $method \equiv \mathtt{mtls}$ (Line 25) or if $method \in \{\mathtt{sign}, \mathtt{mac}\}$ (Line 2).

For the algorithm to return in the former case, it must hold that there exists an $mtlsInfo \in^{\langle\rangle} S(p).\mathtt{mtlsRequests}$ such that $mtlsInfo.1 \equiv m.\mathtt{body}[\mathtt{mtlsNonce}] \wedge mtlsInfo.2 \equiv key$. This $mtlsInfo$ must have been stored in Line 154 of Algorithm A.13 (in case of an AS) or in Line 26 of Algorithm A.20 (in case of an RS). In both cases, the $mtlsNonce \equiv mtlsInfo.1 \equiv m.\mathtt{body}[\mathtt{mtlsNonce}]$ is sent only encrypted with the $clientKey \equiv mtlsInfo.2 \equiv key$. Thus, due to Lemmas D.1 and D.4, only c can decrypt the $mtlsNonce$, since only c can know the matching private key. Since $mtlsNonce \equiv m.\mathtt{body}[\mathtt{mtlsNonce}]$, for the sender p' of m, it must hold that $mtlsNonce \in d_{\emptyset}(S(p'))$. Thus, due to Lemma D.5, p' must be c or as. Since the honest AS as does not send HTTP requests, the message m must have been sent by c. Choosing $m' = m$ then shows the statement for the MTLS case.

In the case that $method \in \{\mathtt{sign}, \mathtt{mac}\}$, it must hold that $\mathsf{checksig}(m.\mathtt{headers}[\mathtt{signature}], key) \equiv \top$ (if $method \equiv \mathtt{sign}$) or that $\mathsf{checkmac}(m.\mathtt{headers}[\mathtt{signature}], key) \equiv \top$ (if $method \equiv \mathtt{mac}$). It also must hold that $controlInput \equiv \mathsf{extractmsg}(m.\mathtt{headers}[\mathtt{signature}])$. If we have that $\mathsf{checksig}(m.\mathtt{headers}[\mathtt{signature}], key) \equiv \top$, the signature must have been created by c due to Lemmas D.1 and D.4.
If $\mathsf{checkmac}(m.\mathtt{headers}[\mathtt{signature}], key) \equiv \top$, the MAC must have been created by c due to Lemma D.3 and the assumption that as and rs are honest, since according to the lemma only c, as, and rs can know the symmetric key, but as and rs do not use it to generate MACs. Since the signature or MAC must have been created by c and c only creates signatures or MACs when calling Algorithm A.4 and $controlInput \equiv \mathsf{extractmsg}(m.\mathtt{headers}[\mathtt{signature}])$, c must have sent an HTTP request m' using $\mathsf{SIGN\_AND\_SEND}$ for which the following holds true:

(1) $m'.\mathtt{method} \equiv m.\mathtt{method}$, since the HTTP method is always covered by the *controlInput* (Line 13 of Algorithm A.5),
(2) $m'.\mathtt{body} \equiv m.\mathtt{body}$, since if $m.\mathtt{body} \not\equiv \langle\rangle$ the body is covered by the *controlInput* using a hash of it (Line 15 of Algorithm A.5) and if $m.\mathtt{body} \equiv \langle\rangle$ it must hold that $m'.\mathtt{body} \equiv \langle\rangle$ as otherwise c would have included the hash of the body in the signature in Line 5 of Algorithm A.4 and thus *controlInput* would not match $\mathsf{extractmsg}(m.\mathtt{headers}[\mathtt{signature}])$,
(3) $m'.\mathtt{host} \equiv m.\mathtt{host}$, since the host is always covered by the *controlInput* via the *controlURL* (Line 12 of Algorithm A.5),
(4) $m'.\mathtt{path} \equiv m.\mathtt{path}$, since the path is always covered by the *controlInput* via the *controlURL* (Line 12 of Algorithm A.5),
(5) $m'.\mathtt{parameters} \equiv m.\mathtt{parameters}$, since the parameters are always covered by the *controlinput* via the *controlURL* (see Line 12 of Algorithm A.5), and
(6) if $\mathtt{Authorization} \in m.\mathtt{headers}$, it must hold that $m'.\mathtt{headers}[\mathtt{Authorization}] \equiv m.\mathtt{headers}[\mathtt{Authorization}]$ since when using the $\mathtt{Authorization}$ header, the header is covered by the *controlInput* (Line 8 of Algorithm A.5).

Therefore, all equivalences required by the lemma must hold in this case as well, which proves the lemma. □

D.3 AS - CI flow continuity

Lemma D.9 (Public Key stored during Registration belongs to Client Instance). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every client instance $c \in$* CI *that is honest in S, every AS $as \in$* AS *that is honest in S, and every instance identifier instanceID that has been stored in $S'(c).\mathtt{keyRecords}[domainAS].\bar{i}$ in Line 12 of Algorithm A.7 for some previous state S', a domain $domainAS \in \mathsf{dom}(as)$, and some $\bar{i} \in \mathbb{N}$, it holds true that*

$$\mathsf{pub}(S(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{key}) \equiv S(as).\mathtt{registrations}[instanceID].\mathtt{publicKey}.$$

Proof. If an instance identifier is stored in Line 12 of Algorithm A.7, it must hold that $\mathtt{instanceID} \in m.\mathsf{body}$ (due to Line 9 and Line 8). m is a response to an HTTP request sent to *domainAS*, since $\mathit{domainAS} \equiv S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{AS}]$ (Lines 3 and 4), all grant requests are sent to the domain stored in this value, and this value never changes once it is set. By Lemma C.2, the grant response processed by c must originate from *as*.

as only uses the instanceID key in a response in Lines 46 and 133 of Algorithm A.13. Since c is honest, it doesn't send requests to the /introspect endpoint of *as*, and hence m can not contain the instanceID key from 133. Under the returned *instanceID*, *as* stores the value *publicKey* in the *registrations* subterm (Line 42 resp. Line 45), which is why it must hold that

$$\begin{aligned}
&S'(\mathit{as}).\mathtt{registrations}[\mathit{instanceID}].\mathsf{publicKey} && \\
\equiv\ &\mathit{publicKey} && \text{(L. 42/45)}\\
\equiv\ &\mathit{grantRequest}[\mathtt{client}][\mathtt{key}] && \text{(L. 41/44)}\\
\equiv\ &m.\mathsf{body}[\mathtt{client}][\mathtt{key}] && \text{(Line 5)}
\end{aligned}$$

Note that Line 38 of Algorithm A.13 prevents the stored client registration record from being overwritten (this is the only code section where the client registration records are written to), so it must hold that the value of the $\mathtt{registration}[\mathit{instanceID}]$ entry in $S'(\mathit{as})$ and $S(\mathit{as})$ are equivalent. This means that $S(\mathit{as}).\mathtt{registrations}[\mathit{instanceID}].\mathsf{publicKey}$ must be the value $m.\mathsf{body}[\mathtt{client}][\mathtt{key}]$ from the HTTP request m sent by c to *domainAS* (using HTTPS). The only lines where c sends such a message (a grant request containing a client entry) are Line 44 and Line 49 of Algorithm A.6. In both cases, the public key

$$\begin{aligned}
&\mathsf{pub}(\mathit{key}) && \\
\equiv\ &\mathsf{pub}(\mathit{keyRecord}.\mathsf{key}) && \text{(Line 41)}\\
\equiv\ &\mathsf{pub}(S''(c).\mathsf{keyRecords}[\mathit{domainAS}].\bar{i}.\mathsf{key}) && \text{(Line 27)}
\end{aligned}$$

(for a previous state S'' and some $\bar{i} \in \mathbb{N}$) is transmitted in the grant request. The key record used in Line 27 is stored in $S''(c).\mathtt{grants}[\mathit{grantID}][\mathsf{keyRecord}]$ in Line 28 and this value cannot change. *grantID* thereby is the grant ID which is used under the grantID key in the reference for the request (in Line 46 of Algorithm A.6 resp. in Line 53 of Algorithm A.6 and then again in Line 89 of Algorithm A.7). Thus, the key record read when processing the response from *as* in Line 6 of Algorithm A.7 is the key record chosen in Line 27 of Algorithm A.6 ($S''(c).\mathsf{keyRecords}[\mathit{domainAS}].\bar{i}$), since

$$\begin{aligned}
&\mathit{keyRecord} && \\
\equiv\ &\mathit{grantRequest}[\mathsf{keyRecord}] && \text{(Line 6)}\\
\equiv\ &S'(c).\mathtt{grants}[\mathit{grantID}].\mathsf{keyRecord} && \text{(Line 3)}\\
\equiv\ &S'(c).\mathtt{grants}[\mathit{reference}[\mathtt{grantID}]].\mathsf{keyRecord} && \text{(Line 2)}
\end{aligned}$$

Thus, in Line 12 of Algorithm A.7, the *instanceID* returned by *as* is stored in the key record whose public key was stored at *as* when c was registered. Due to the check in Line 10, this key record cannot change anymore (this is the only code section where key records are written). Thus, it holds that

$$\begin{aligned}
&S(\mathit{as}).\mathtt{registrations}[\mathit{instanceID}].\mathsf{publicKey}\\
\equiv\ &\mathsf{pub}(S'(c).\mathsf{keyRecords}[\mathit{domainAS}].\bar{i}.\mathsf{key})\\
\equiv\ &\mathsf{pub}(S(c).\mathsf{keyRecords}[\mathit{domainAS}].\bar{i}.\mathsf{key}).
\end{aligned}$$

□

Lemma D.10 (Continuation Request must stem from the same Client Instance). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, and every AS $as \in \mathsf{AS}$ that is honest in S, it holds true that if as calls $\mathsf{PERFORM\_KEY\_PROOF}(m, \mathit{grantID}, s')$ (for some m, grantID, s') in Line 54 of Algorithm A.13 or in Line 65 of Algorithm A.13 (i.e. when as receives a continuation request) and this call returns, then the request m must have been sent by the client instance c that sent the grant request that led to the creation of the grant ID grantID in Line 3 of Algorithm A.13 as long as c is an honest client instance in S. If c used a key record r with $r.\mathsf{method} \equiv \mathsf{mac}$ for this grant request, we additionally require that $rs = \mathsf{dom}^{-1}(r.\mathsf{rs})$ is honest in S.*

Proof. If *as* calls PERFORM_KEY_PROOF (Algorithm A.14) in one of these lines, it must hold, based on the checks in Line 53 and Line 64 respectively, that $\mathit{grantID} \in S(\mathit{as}).\mathtt{grantRequests}$. *grantID* must have been stored in $S'(\mathit{as}).\mathtt{grantRequests}$ in Line 9 of Algorithm A.13 for a previous state S', because only in this code section new grant IDs are assigned by an AS. This can only have happened if the call to PERFORM_KEY_PROOF in Line 4 returned, so if the key proof for the grant request was successfully validated. During this call to PERFORM_KEY_PROOF, it must have held in Line 2 of Algorithm A.14 that $S'(\mathit{as}).\mathtt{grantRequests}[\mathit{grantID}] \equiv \langle\rangle$, since *as* just created the *grantID* in Line 3 of Algorithm A.13. Thus, in this call, the key proofing method and the key used by c are stored in Lines 27-35 of Algorithm A.14 in $S'(\mathit{as}).\mathtt{grantRequests}[\mathit{grantID}]$. Since this is the only code section where these subterms are written to and this write operation implies that the condition $S(\mathit{as}).\mathtt{grantRequests}[\mathit{grantID}] \equiv \langle\rangle$ now no longer holds, this information cannot be overwritten. This means that in S it must hold that the key proofing method and the key stored in $S(\mathit{as}).\mathtt{grantRequests}[\mathit{grantID}]$ are still the same.

In the calls to PERFORM_KEY_PROOF in Lines 54 and 65 of Algorithm A.13 it thus must still hold that $S(\mathit{as}).\mathtt{grantRequests}[\mathit{grantID}] \not\equiv \langle\rangle$, so now the previously stored key proofing method and the used key are loaded again from $S(\mathit{as}).\mathtt{grantRequests}[\mathit{grantID}]$ in Lines 37-45 of Algorithm A.14.
So PERFORM_KEY_PROOF calls VALIDATE_KEY_PROOF in Lines 47-51 with the same method and key as when handling the grant request. Since PERFORM_KEY_PROOF returns only if VALIDATE_KEY_PROOF returns and c, *as*, and possibly *rs* are honest by precondition, it must thus hold according to Lemma D.8 that m was sent by c.[6] It is also not possible that m is a continuation request replayed by the attacker, since replays are detected within VALIDATE_KEY_PROOF (Algorithm A.5) as follows. If MTLS is used, a replay is not possible because a new *mtlsNonce* is used by the AS for each request. If signatures or MACs are used, a replay of

[6] Regarding headers, according to Lemma D.8, only the Authorization header must be equivalent, others can potentially differ. However, since *as* only accesses the Authorization header here, this is sufficient.

the request is detected in Line 11, since the replayed request must contain the same nonce in *sigParams* as the original request, this nonce was already stored in Line 24 in the *sigNonces* subterm of the state of *as* when the original request was validated, and there is no code section where nonces are removed from *sigNonces*. □

Lemma D.11 (Only one AS for each Client Flow). *Let ρ be a run of a GNAP web system $\mathcal{GWS}$, with $i \leq j$ and S^i, S^j two states of ρ. Suppose c is a client instance and as is an AS, both honest in S^j.*

Suppose also that in step i, c sends a grant request m^i to as and that in step j, c sends a grant or continuation request m^j to some process p, both with references containing the mapping `grantID` : *CIgid.*

Then, $p = as$.

Proof. We will argue by induction on j, assuming for induction that all grant or continuation requests m^k sent by c at times k with $k < j$ in the flow identified by *CIgid* were sent to *as*.

An honest client instance c only sets the reference for a grant request on Lines 33,46 of Algorithm A.6, Line 89 of Algorithm A.7, or Line 13 of Algorithm A.8, and the reference for a continuation request on Lines 64,99,116 of Algorithm A.7 or Line 17 of Algorithm A.10.

First, we consider the case of grant requests. We claim that an honest client c will only send one grant request per grant ID *CIgid*, and so since m^i and m^j are sent with the same grant ID, $m^i \equiv m^j$ and $i \equiv j$. This then immediately gives that $p = as$.

In three of the places that a grant request reference can be set (Lines 33,46 of Algorithm A.6 and Line 13 of Algorithm A.8), the grant ID used is freshly generated shortly earlier (Line 9 of Algorithm A.6, or Line 7 of Algorithm A.8), and so cannot be a reuse of some other grant ID. The fourth, Line 89 of Algorithm A.7, is used to send out a grant request after completing an MTLS handshake. Here, c is processing a response to some prior request, whose reference satisfied *reference*[`responseTo`] ≡ `MTLS_GR` and *reference* [`grantID`] ≡ *CIgid* (Line 2 of Algorithm A.7). Such references are set in three places, each of which is immediately followed by sending the corresponding message: Lines 38,53 of Algorithm A.6 and Line 18 of Algorithm A.8. An honest client c will only handle one response to each request, and so it suffices to show that each of these requests does not reuse a grant ID that was previously used for a grant request. This is easy to see, as in each case, the grant ID used is freshly generated shortly earlier, as in the case of directly sent grant requests.

Next, we will show that for each type of continuation request, the request m^j is sent to a domain m.body[`continue`][`url`].host, where m is a response to a grant or continuation request sent in the flow identified by *CIgid*.

- If the reference for m^j is set on Line 64 of Algorithm A.7, then m^j is sent in the next line to a domain *continueURL*, which is loaded on Lines 60 and 8 from m.body[`continue`] [`url`] for some response m being processed. We know from the check on Line 7 that the request m^k corresponding to this response is either a grant request or a continuation request, and that (from Line 2) it was sent in the flow identified by *CIgid*. Moreover, since the check on Line 51 must have succeeded, the `continue` field of m.body must have been set. If m^k is a grant request, then we know that $k = i$, and so m^k is sent to *as*. If m^k is instead a continuation request, since $k < j$, the induction hypothesis applies and tells us that m^k is sent to *as*. In either case, the corresponding response m is sent by *as*.
- If the reference for m^j is set on Line 99 of Algorithm A.7, then m^j is sent on Line 107 to a domain loaded on Line 98 from $S^j(c)$.`grants`[*CIgid*][`continueURL`]. This field of the state is only written to on Line 35, when c is processing a response m to a grant or continuation request. As shown before, it must be the case that this m was sent by *as* and processed in the flow identified by *CIgid*.
- If the reference for m^j is set on Line 116 of Algorithm A.7, then m^j is sent to a domain loaded on Line 113 from $S^j(c)$.`grants`[*CIgid*][`patchRequest`].3.`host`. This field of the state is only written to on Line 67, and a similar argument to the previous case shows that the value of this field is received in an m sent by *as* as desired.
- If the reference for m^j is set on Line 17 of Algorithm A.10, then on Line 20 or 23, m^j is sent to *continueURL*, which was loaded on Line 8 from $S^j(c)$.`grants`[*CIgid*][`continueURL`]. As shown before, it must be the case that this URL is loaded from a message m sent by *as* and processed in the flow identified by *CIgid*.

In each case, we have that the domain to which m^j was sent is loaded from m.body[`continue`][`url`].host for some response m received from *as* in response to a grant or continuation request sent in the flow identified by *CIgid*. We can therefore examine the behaviour of an honest AS to see what values this domain can have. In particular, if an honest AS only ever sets this field of a response to domains that it controls, we obtain our result, as we then ensure that m^j can only be sent to a domain belonging to *as*. There are two places where the `continue` field of a response can be set.

If the [`continue`][`url`] field is set at Line 13 or 16 of Algorithm A.13, the host used for this URL is m.`host`, where m is the request currently being processed by *as*. By Lemma C.1, this is necessarily a domain belonging to *as*.

If the [`continue`][`url`] field is set at Line 37 of Algorithm A.17, its host is simply taken from the argument of the call to CREATE_GRANT_RESPONSE, and we need to examine in turn the callsites of this algorithm. One of these is contained in SEND_GRANT_RESPONSE, where the host passed in is m.host, with m being the fourth argument to SEND_GRANT_RESPONSE itself. Between these two functions, there are five callsites, which we consider in turn:

- On Lines 34,75,77 of Algorithm A.13: CREATE_GRANT_RESPONSE is called, with host argument being taken directly from the request that *as* is processing. We can apply Lemma C.1 and immediately conclude that this must be a domain of *as*.
- On Lines 59 and 70 of Algorithm A.13: SEND_GRANT_RESPONSE is called, with its fourth argument being exactly the request that *as* is processing. As in the previous case, Lemma C.1 ensures that the domain passed in to CREATE_GRANT_RESPONSE is a domain of *as* as desired.

□

Lemma D.12 (Client-perspective flows agree with AS-perspective flows). *Let ρ be a run of a GNAP web system $\mathcal{GWS}$, with $i < j$ and S^i, S^j two states of ρ. Suppose c is a client instance and as is an AS, both honest in S^j.*

Suppose also that in step i, c sends a grant start request m^i to the `/requestGrant` *endpoint of as and that in step j, c sends a continuation request m^j to the* `/continue` *path of as, both with references containing the mapping* `grantID` : *CIgid.*

If m^j contains a continuation access token AT with $AT \equiv S(as)$.`grantRequests`*[ASgid]*[`continueAT`]*, then ASgid was created on Line 3 of Algorithm A.13 while as was processing m^i.*

Proof. We proceed by induction on j.

Since c is honest, m^j can only be sent at Lines 65 and 107 of Algorithm A.7 or Lines 20 and 23 of Algorithm A.10. We will examine in each case where c loads AT. We claim that in each case, AT is originally received by c in *grantResponse*[`continue`][`accessToken`] in a response whose corresponding reference satisfies *reference* [`grantID`] ≡ *CIgid* and *reference*[`responseTo`] ≡ `grantResponse` — that is, either another, earlier continuation request or a grant start request.

In the first case, where m^j is sent on Line 65 of Algorithm A.7, this is direct, with AT being received on Line 59.

In the case where m^j is sent on Line 117 of Algorithm A.7, the AT that is sent is loaded (along with the rest of the authorization header containing it) on Line 113 from $S^j(c)$.`grants`[*CIgid*] [`patchRequest`]. This field is only written to on Line 67, with the stored authorization header coming from Line 61 , using the access token received on Line 59.

In the three remaining cases, AT is loaded from $S^j(c)$.`grants` [*CIgid*][`continueAT`], either on Line 96 of Algorithm A.7 or Line 16 of Algorithm A.10. This field is only written to on Line 34 of Algorithm A.7, where the value written is taken directly from an appropriate grant response.

The request corresponding to this response must have been either a grant start request or a continuation request (as these are exactly the request types for which *reference*[`responseTo`] ≡ `grantResponse`), and must have been sent at some earlier time $k < j$. Moreover, this request must have been sent in the same grant ID *CIgid*.

We first consider the case where this request, m^k, is a continuation request. By Lemma D.11, since m^i was sent to *as*, so was m^k. Since m^k is sent to *as* and we know that *as* responds to m^k (yielding AT), it must be the case that *as* successfully executed CREATE_GRANT_RESPONSE at one of the four lines 59,70,75 or 77 of Algorithm A.13 (with the first two being indirect, within SEND_GRANT_RESPONSE). In each case, m^k must contain a cont. access token AT' such that $S^k(as)$.`grantRequests`[*ASgid′*] [`continueAT`] ≡ AT' for some *ASgid′*, due to checks on Lines 53 and 64. Moreover, we know that the reference corresponding to m^k satisfies *reference*[`grantID`] ≡ *CIgid*. At this point, it will suffice to show that *ASgid′* ≡ *ASgid*, as this will allow us to invoke the inductive hypothesis (with $k < j$, and using AT' as the token) and immediately reach the desired result.

For each case of CREATE_GRANT_RESPONSE where continuation is allowed (which must be the case — otherwise the response to m^k would not contain AT), the new token AT is stored in $S^k(as)$.`grantRequests`[*ASgid′*][`continueAT`] on Line 38 of Algorithm A.17. Since we also know that $AT \equiv S^j(as)$.`grantRequests` [*ASgid*][`continueAT`], we can conclude that *ASgid′* ≡ *ASgid* if a given continuation access token can never be stored under two different grant IDs. Continuation access tokens are stored in three places: Lines 14 and 22 of Algorithm A.6 and Line 38 of Algorithm A.17. The first two of these only store freshly generated continuation access tokens, which therefore cannot already be stored under a different grant ID. The final case, can either store a freshly generated token, or a reused token coming from Line 38 of Algorithm A.17. This reused token is passed in as an argument to CREATE_GRANT_RESPONSE, as is the grant ID under which the reused token may be stored. We therefore must show that all calls to CREATE_GRANT_RESPONSE that provide an old continuation access token for potential reuse also pass in the grant ID under which that token is stored, ensuring that it can only be re-stored under the same grant ID. There are four places where CREATE_GRANT_RESPONSE is called: Lines 34 ,75 and 77 of Algorithm A.13 and Line 4 of Algorithm A.16. The first of these, passes ⊥ in place of an old continuation access token. This cannot be reused, and so no check is needed. The next two, pass in the correct grant ID for the token, as validated by the check on Line 64. Finally, Line 4 of Algorithm A.16 loads the continuation access token that it uses as an argument directly from $S(as)$.`grantRequests`[*grantID*] [`continueAT`], ensuring that the token and grant ID arguments match. As such, we have that *ASgid′* ≡ *ASgid*, since the same access token was stored under both grant IDs, and this tells us that AT' is stored under the correct *ASgid* to allow us to invoke our inductive hypothesis, concluding the case where AT was received in a response to a continuation request.

If, instead, AT is received in response to a grant start request, we claim that this request is m^i, which immediately gives the desired result. We will show that this grant start request must be m^i by showing that an honest client instance c never sends two distinct grant start requests with the same grant ID *CIgid*.

Grant start requests are sent at four places, three of which (Lines 34 and 47 of Algorithm A.6 and Line 14 of Algorithm A.8) immediately follow the creation of a fresh grant ID. The remaining source of grant start requests, Line 90 of Algorithm A.7, is only reached when receiving a response with reference containing `responseTo` : `MTLS_GR`. Corresponding requests are only sent in three places (Lines 39 and 54 of Algorithm A.6 and Line 19 of Algorithm A.8), all of which immediately follow the creation of a fresh grant ID. Moreover, each of these cases is mutually exclusive with the immediate sent of a grant start request on Lines 34 and 47 of Algorithm A.6 and Line 14 of Algorithm A.8, respectively. It will therefore suffice to show that only one start grant request is sent per `MTLS_GR` request. Because c only handles one response to each `MTLS_GR` request, it sends only one start grant request per `MTLS_GR` request. As such, the start grant request whose response contained AT must have been exactly m^i, and so we have the desired result in the case where AT is received in response to a start grant request. □

Lemma D.13 (Access Tokens and Subject IDs are returned only to authorized Client Instances). *Suppose ρ is a run of a*

GNAP web system $\mathcal{GWS}$, and $Q : (S, E, N) \rightarrow (S', E', N')$ is a step in ρ. Let c be a client instance and as an AS, both honest in S.

Suppose

- *c receives a message m from as in step Q.*
- *Either* $\texttt{accessToken} \in m.\mathsf{body}$ *or* $\texttt{subjectID} \in m.\mathsf{body}$. *Let data* $= m.\mathsf{body}[\texttt{accessToken}]$ *or data* $= m.\mathsf{body}[\texttt{subjectID}]$ *(if both exist, either can be chosen).*
- *c stores data under* $S(c).\texttt{receivedValues}[CIgid]$.
- *For all client instances* c' *that are honest in S and all key records* $r \in s_0^c.\texttt{keyRecords}[dmnAS]$ *(for some domain dmnAS* $\in \mathsf{dom}(\mathsf{governor}(u))$*) with* $r.\mathsf{method} \equiv \texttt{mac}$*, it holds that* $\mathsf{dom}^{-1}(r.\mathsf{rs})$ *is honest in S. (honest client instances share symmetric keys only with* $\mathsf{governor}(u)$ *and honest RSs), and*

In the case of data being an access token we further require that as is the issuer of data and $S(as).\texttt{tokenBindings}[data][\texttt{for}] \equiv \texttt{endUser}$.
We then set $ASgid = S(as).\texttt{tokenBindings}[data][\texttt{grantID}]$ *and* $u = S(as).\texttt{grantRequests}[ASgid][\texttt{subjectID}]$.

In the other case, where data is a subject ID, we require that c stores $\langle data, d\rangle$ *in* $S(c).\texttt{receivedValues}[CIgid]$ *for a domain d of as, and set u = data and ASgid with* $S(as).\texttt{grantRequests}[ASgid][\texttt{subjectID}] \equiv u$.

Then, at some step Q' before Q, $\mathsf{finishLogin}_\rho^{Q'}(\mathsf{ownerOfID}(u), c, u, as, CIgid, ASgid)$.

Proof. Since *as* is an honest AS, it only generates a message with the subjectID flag on Line 28 of Algorithm A.17, and likewise only generates a message with the accessToken flag on Line 7 of Algorithm A.17.

If we take $data = m.\mathsf{body}[\texttt{accessToken}]$, we note that the tokenBindings field of the state of *as* is only set in CREATE_GRANT_RESPONSE, and in each case, the for field of this dictionary is set to the *for* parameter passed in to the algorithm. As such, the *for* parameter must be endUser.

If instead, $data = m.\mathsf{body}[\texttt{subjectID}]$, we see from the check on Line 3 of Algorithm A.17 that $for \not\equiv \texttt{CI}$.

In either case, then, we have that CREATE_GRANT_RESPONSE must have been called with the *for* parameter not being CI.

Moreover, from the check on Line 2 of Algorithm A.17, we see that $\texttt{subjectID} \in S(as).\texttt{grantRequests}[gid]$ for *gid* being the grant ID from the call of Algorithm A.17. We show that $gid = ASgid$. If *data* is an access token, this token is newly generated in this function and associated with *gid* (the first argument of the function call) by $\texttt{tokenBindings}[data][\texttt{grantID}] : gid$. By the definition of *ASgid*, we thus have $gid = ASgid$. If *data* is a subject ID, then the value is chosen according to *gid* (Line 28) and the definition of *ASgid* shows the claim.

The subjectID field of $S(as).\texttt{grantRequests}[ASgid]$ is only written to in Algorithm A.15, on Line 17, and we know from the preconditions of the lemma that this subject ID is u. As such, at some previous step Q', *as* executed Algorithm A.15, and since the state change must have been persistent, as it is visible in S, the algorithm must have returned successfully.

Since u is stored on Line 17, the password check on Line 8 must have succeeded, so the message m' passed to FINISH_INTERACTION must contain $\mathsf{secretOfID}(u)$. This message m' must be a request received by *as* at one of its Login endpoints, as these are the only places where FINISH_INTERACTION is called. By Lemma D.7, $\mathsf{secretOfID}(u)$ is only derivable by *as* and $\mathsf{ownerOfID}(u)$, and as *as* is honest, it does not send requests, so m' must have been sent by $\mathsf{ownerOfID}(u)$, which, being honest, will only send a request to one of the Login endpoints of *as* while executing *script_as_login*. At this point, we know the second and third parts of $\mathsf{finishLogin}_\rho^{Q'}(\mathsf{ownerOfID}(u), c', u, as, CIgid', ASgid)$ for some c' and $CIgid'$, and that for some prior step Q'', $\mathsf{tryLogin}_\rho^{Q''}(\mathsf{ownerOfID}(u), c', as, CIgid', ASgid,)$.

It now remains to show that $c' = c$ and $CIgid' = CIgid$. We know that c stores *data* under $S(c).\texttt{receivedValues}[CIgid]$, and that *as* must have created the message m containing *data* in a call to CREATE_GRANT_RESPONSE for which the *for* parameter was not CI. This restricts the set of possible places that this call could have happened to two places, one of which is inside SEND_GRANT_RESPONSE, which is in turn called in two places. The three resulting callsites are on Lines 59, 70, and 75, each of which can only be reached in response to *as* receiving a request $\hat{m}$ to its /continue endpoint, and after successfully executing PERFORM_KEY_PROOF($\hat{m}, ASgid, s'$) on one of Lines 54 and 65. Since the response m to $\hat{m}$ is sent to c, $\hat{m}$ must have been sent by c, and as c is honest, it will only use its own keys in key proofs. As PERFORM_KEY_PROOF succeeds, so does VALIDATE_KEY_PROOF called inside, which uses as its *key* argument the public key sent to *as* by c' initially when *ASgid* was created. As in the proof of Lemma D.8, we can check that in all cases, the key proof succeeding means that the key proof created by c for $\hat{m}$ must use the private key corresponding to the public key sent to *as* by c', and as a key proof must succeed initially for this public key to be stored, c' must have this same private key. As such, $c = c'$, and it only remains to show that $CIgid = CIgid'$. By definition, $CIgid'$ is the grant ID of c that was created in the processing step in which c sent the initial grant request to *as* that created *ASgid*. Since c and *as* are both honest, and both the initial grant request and the continuation request $\hat{m}$ are handled by *as* in *ASgid*, both must have been sent in the same *CIgid* (as c, being honest, will only use continuation access tokens in the same flow, as identified by *CIgid*, in which it received them). □

D.4 CI - Browser flow continuity

Lemma D.14 (Same End User must be present after Interaction). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, and every client instance $c \in \mathsf{CI}$ that is honest in S it holds true that if in Line 61 or Line 79 of Algorithm A.6 c stores* $\langle k, a, f, m.\mathsf{nonce}\rangle$ *under* $S(c).\texttt{browserRequests}[grantID][\texttt{finishRequest}]$ *(for some grantID, k, a, f, m), then the request m must have been sent by the browser b that sent the request to the* /startGrantRequest *path of c that led to the creation of grantID in Line 9 of Algorithm A.6 as long as b is not fully corrupted in S.*

Proof. If c executes Line 61 or Line 79 of Algorithm A.6, it must hold that

$$m.\texttt{headers}[\texttt{Cookie}][\langle\texttt{\_\_Host}, \texttt{sessionID}\rangle] \\ \equiv S(c).\texttt{grants}[grantID][\texttt{sessionID}]$$

due to the check in Line 59 resp. Line 77. So the session ID transferred by b to c in the headers of m is sent in a cookie with the `__Host` prefix set. Since the `__Host` prefix is set, this cookie must have been transmitted by c to b using HTTPS and with the `secure` attribute set. The $\langle$`__Host`, `sessionID`$\rangle$ cookie is set by honest client instances only when answering a browsers request to start a grant in Line 38 of Algorithm A.7 and in case of a logout in Line 88 of Algorithm A.6. In both cases the `secure` attribute, the `session` attribute and the `httpOnly` attribute are set. Thus, only b is able to decrypt a session ID when it is transmitted within the `Set-Cookie` header from c to b. Since b is not fully corrupted by precondition, b again only transmits the session ID to c and does so using HTTPS. In case of a close corruption of b, the session ID cannot get leaked to the attacker because the `session` attribute is set. Furthermore, honest client instances transmit session IDs only to the browser for which they were issued and only using the $\langle$`__Host`, `sessionID`$\rangle$ cookie. Thus, a session ID created by c for b can only be derivable for c and b in S under the assumed conditions.

The value to which the session ID transferred by b is compared, $S(c)$.`grants`[*grantID*][`sessionID`], must have been stored during a call to the `/startGrantRequest` path of c in Line 28 of Algorithm A.6, since this is the only line where this entry is written to. Since grant IDs do not change, this must also have been the call to the `/startGrantRequest` path in which *grantID* was created in Line 9 of Algorithm A.6. The value stored in Line 28 of Algorithm A.6 is either a session ID that was transferred by the process that called the `/startGrantRequest` path (Line 22) or a new session ID generated by c for this session (Line 26) that is later transferred to the browser in Line 38 of Algorithm A.7. So in any case, it must be a session ID that was transferred to the caller in a $\langle$`__Host`, `sessionID`$\rangle$ cookie. As seen, only b and c are able to derive this value in S, so if Line 61 or Line 79 of Algorithm A.6 is executed by c, it must hold that m was sent by the browser that sent the request to the `/startGrantRequest` path that led to the creation of *grantID*, otherwise the corresponding session ID could not have been included in the headers of m. □

Lemma D.15 (Granted Grant Request must have been started by RO). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S^j, E^j, N^j) in ρ, every client instance $c \in$ CI that is honest in S^j, every grant ID grantID $\in S^j(c)$.*`grants`*, and every identity $u \in$ ID it holds true that if* $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u),$ *grantID, ASgid) (for some grant ID ASgid and a previous processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ and some integer $i < j$) and c calls* SEND_CONTINUATION_REQUEST(*grantID, interactRef, hash, s′, a) (for some interactRef, hash, s′, a) in the processing step $(S^j, E^j, N^j) \rightarrow (S^{j+1}, E^{j+1}, N^{j+1})$, then* ownerOfID$(u)$ *must have sent the request to the* `/startGrantRequest` *path of c that led to the creation of grantID in Line 9 of Algorithm A.6 as long as* ownerOfID(u) *is not fully corrupted in S^j and* governor(u) *is honest in S^j.*

Proof. For this proof, we need to distinguish the different interaction modes that can get used.

Redirect Start Mode + Redirect Finish Mode: We have $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID}, \mathit{ASgid})$ and c has thereupon called SEND_CONTINUATION_REQUEST(*grantID*, *interactRef*, *hash*, s', a). Since the redirect interaction finish mode is used, the call to SEND_CONTINUATION_REQUEST must have occurred in Line 64 of Algorithm A.6. So there must have been a call to the `/finish` path of c (Line 55). We will now show that this call must have been made by ownerOfID(u). In Line 57, the *grantID* is determined using the *finishURLnonce* from the parameters of the request. The *finishURLnonce* is created by c in Line 15 of Algorithm A.6 and then transferred to the AS within the grant request using HTTPS. Since the AS governor(u) is honest by precondition, governor(u) must have forwarded ownerOfID(u) to the finish URL with the *finishURLnonce* of c immediately after the authentication in processing step Q, which happens in Line 21 of Algorithm A.15. Since c and governor(u) are honest and the *finishURLnonce* is not otherwise used or sent, the call to the `/finish` path must thus have been made by ownerOfID(u), otherwise it could not contain the *finishURLnonce* in the parameters.

If SEND_CONTINUATION_REQUEST is called in Line 64 of Algorithm A.6, Line 61 must also have been executed. Thus, by Lemma D.14, the call to the `/finish` path must have been made by the same browser that sent the associated grant request to the `/startGrantRequest` path, which, as seen, must be ownerOfID(u).

Redirect Start Mode + Push Finish Mode: Since the push interaction finish mode is used, the call to SEND_CONTINUATION_REQUEST must have occurred in Line 71 of Algorithm A.6. So there must have been a call to the `/push` path of c (Line 65). This call must come from the AS to which c sent the grant request with the grant ID *grantID*, since only this AS can know the *finishURLnonce*, which is used in Line 67 to determine the grant ID, since c created this nonce (in Line 15 of Algorithm A.6) and only transmits it within the grant request.

Since $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID}, \mathit{ASgid})$, this AS must be governor(u), as an honest client instance c will only send a grant request with a given grant ID to one AS, and we know that c sends a grant request with the grant ID *grantID* to governor(u). The request to the `/push` path made by governor(u), which is honest by precondition, is sent in Line 25 of Algorithm A.15 (FINISH_INTERACTION). FINISH_INTERACTION must have been called in Line 101 of Algorithm A.13 due to the use of the redirect interaction start mode, i.e., as a result of a request to the `/redirectLogin` path of governor(u).

Since $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID}, \mathit{ASgid})$ holds, this request to the `/redirectLogin` path of governor(u) must have been sent by ownerOfID(u) in processing step Q. When a request to the `/redirectLogin` path is received, Line 99 ensures that ownerOfID(u) was redirected from the client instance that sent the grant request by verifying that the host of the `Referer` header matches the host of the finish URL from the grant request (the value of the `Referer` header is transmitted in the body of the request by *script_as_login* after being passed to *script_as_login* via the *scriptstate* in Line 85 of Algorithm A.13). The host of the finish URL must be a domain of the client instance that sent the grant request, which follows from Line 15 of Algorithm A.6 using Lemma C.1. Thus, ownerOfID(u) must have been forwarded by c in the context of this grant (which is uniquely identified by the *redirectNonce* in Line 98 of Algorithm A.13).

This redirect happens in Line 44 of Algorithm A.7. The recipient of the response that receives the redirect is thereby loaded in Line 37 from the `startRequest` entry. This entry must have been written in Line 20 of Algorithm A.6, as this is the only line where this happens. This is done when processing the request to the `/startGrantRequest` path of c which led to the creation of *grantID* in Line 9. Thus, the browser forwarded to governor(u) must thus have sent this request. Since the forwarded browser is ownerOfID(u), ownerOfID(u) must have sent the request to the `/startGrantRequest` path of c that led to the creation of *grantID* in Line 9.

User Code Start Mode: In the case of the user code interaction start mode, the proof is independent of the chosen interaction finish mode. Since tryLogin$^{Q}_{\rho}$(ownerOfID(u), c, u, governor(u), *grantID*, *ASgid*) and SEND_CONTINUATION_REQUEST is called by c, the interaction for the grant with grant ID *grantID* must have finished successfully. Since the user code interaction start mode is used, a user code *uc* must have been included in the *scriptinputs* under the `userCode` key during authentication when *script_as_login* was run. The user code is read in Line 9 of Algorithm A.19 (*script_as_login*) and then transferred inside *formData* in the request to the `/userCodeLogin` path of governor(u) (Line 14). governor(u) retrieves the user code *uc* from the request to the `/userCodeLogin` path in Line 103 of Algorithm A.13. Since the interaction completed successfully, *uc* must be the user code that governor(u) generated in Line 24 of Algorithm A.13 upon receiving the grant request. Otherwise, Algorithm A.13 would have stopped in Line 104 and the interaction would not have been finished.

The user code must have been included in the *scriptinputs* in Line 47 of Algorithm A.2 since only in this line such an entry can be added to the *scriptinputs*. Due to Line 46 this happens only if *domainCI* ≡ *domainUsedCI*. *domainCI* is the domain of the client instance that sent the grant request associated with *uc* to governor(u). This value was returned by governor(u) along with `script_as_login` in Line 94 of Algorithm A.13. *domainUsedCI* is the domain of the client instance to which ownerOfID(u) sent the request to start a grant request (to the `/startGrantRequest` path) that resulted in receiving the user code *uc*. This domain is stored in Line 38 of Algorithm A.2 in the *usedCIs* subterm under the key *uc*. Since *domainCI* must be equivalent to *domainUsedCI*, the client instance to which ownerOfID(u) sent the request to the `/startGrantRequest` path must be the client instance that sent the grant request associated with *uc* to governor(u), which is c. Because c is honest, it leaks the user code *uc* only to the browser that sent the request to the `/startGrantRequest` path (in Line 49 of Algorithm A.7 and using HTTPS). This holds true because the sender of the grant request is stored in Line 20 of Algorithm A.6 in the *browserRequests* subterm, and the recipient of the response in which *uc* is returned by c is loaded in Line 37 of Algorithm A.7 from this *browserRequests* entry, which cannot be overwritten. Since ownerOfID(u) has received *uc*, ownerOfID(u) thus must have sent the request to the `/startGrantRequest` path of c. Hence, ownerOfID(u) must be responsible for the creation of *grantID* in Line 9 of Algorithm A.6.

The lemma could be shown for all possible combinations of interaction start modes and interaction finish modes covered by our model, so it must hold regardless of the interaction modes used. □

E AUTHORIZATION

E.1 Definitions

We will use the function ownerOfResource to determine the owner of a protected resource stored at an honest resource server.

*Definition E.1 (*ownerOfResource*).* Given a GNAP web system $\mathcal{GWS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$, a run ρ of $\mathcal{GWS}$, a configuration (S, E, N) in ρ, an RS $rs \in$ RS that is honest in S, and a nonce $n \in \mathcal{N}$ that is either a subterm of $S(rs)$.`clientResources` or a subterm of $S(rs)$.`userResources`, ownerOfResource: $\mathcal{N} \to \mathcal{W}$ is defined as follows:

- If $n \equiv S(rs)$.`clientResources`$[d][i]$ for a domain d and an instance identifier i, then ownerOfResource(n) depends on whether $i \in s_0^{rs}$.`clientResources`$[d]$.
  If $i \in s_0^{rs}$.`clientResources`$[d]$, ownerOfResource(n) is defined to be the client instance c for which there is a key record $r \in^{\langle\rangle} s_0^{c}$.`keyRecords`$[d']$ for some $d' \in \text{dom}(\text{dom}^{-1}(d))$ such that r.`instanceID` $\equiv i$. Otherwise, n must have been stored in $S(rs)$.`clientResources`$[d][i]$ in Line 38 of Algorithm A.21. In this case, ownerOfResource(n) is the process to which rs returned the newly created n in Line 46 of Algorithm A.21, i.e., ownerOfResource(n) = addr^{-1}(*sender*) for the address *sender* from that line.
- If $n \equiv S(rs)$.`userResources`$[u]$ for an identity u, ownerOfResource(n) is defined to be ownerOfID(u).

Definition E.2 (Successful Use of Token). For a run ρ of a GNAP web system $\mathcal{GWS}$, a client instance c, an RS rs, an AS as, a grant ID *CIgid*, and an access token *AT* we say that *"c successfully used the token AT at rs under CIgid in processing step Q"* written

$$\text{successfulUseOfToken}^{Q}(c, \mathit{CIgid}, \mathit{AT}, rs, as)$$

if

(1) there is a processing step Q' before Q in ρ where c sends an HTTPS request m to the `/resource` path of a domain of rs with
   a) m contains an `Authorization` header with the access token *AT*
   b) the reference of m contains `grantID` : *CIgid*, and
(2) as is the issuer of the token *AT*,
(3) when processing m in A.20, rs sends an introspection request $m_{\text{intro}}^{\text{req}}$ to as and when processing the corresponding response $m_{\text{intro}}^{\text{resp}}$ from as in step Q, rs reaches Line 45 of Algorithm A.21.

Definition E.3 (Resource Access). For a run ρ of a GNAP web system $\mathcal{GWS}$ we say that a browser b accesses a resource of identity u stored at resource server rs in a GNAP flow identified by the nonce *gid* by client instance c in processing step Q with state S in ρ written as

$$\text{accessResource}^{Q}_{\rho}(b, r, u, c, rs, gid)$$

if

(1) $r \equiv s_0^{rs}$.`userResources`$[u]$,

(2) $S(c).\mathtt{grants}[gid][\mathtt{resources}][domainRS] \equiv r$ with $domainRS \in \mathsf{dom}(rs)$, and

(3) in processing step Q, c emits a message m that is an HTTPS response sent from c to b in Line 18 of Algorithm A.11 with $m.\mathsf{body} \equiv r$.

Definition E.4 (Client Instance Resource Access). Suppose ρ is a run of a GNAP web system $\mathcal{GWS}$, and Q is a processing step in ρ from (S, E, N) to (S', E', N'). We say that a client c accesses a resource r stored at a resource server rs under the identifier $instanceID$ and managed by an AS as in a GNAP flow identified at c by the nonce $CIgid$, written

$$\mathsf{sw\_accessesResource}^{Q}_{\rho}(c, r, CIgid, rs, as, instanceID)$$

if:

(1) There is some $domAS \in \mathsf{dom}(as)$ such that $r = S(rs).\mathtt{clientResources}[domAS][instanceID]$

(2) There is some $domainRS \in \mathsf{dom}(rs)$ such that c stores the resource r under $s'.\mathtt{grants}[CIgid][\mathtt{resources}][domainRS]$ in Line 78 of Algorithm A.7

E.2 Security Properties

Definition E.5 (Authorization Property for Software-only Authorization). Let $\mathcal{GWS}$ be a GNAP web system. We say that *$\mathcal{GWS}$ fulfills the authorization property for software-only authorization* iff for every run ρ of $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every RS $rs \in \mathsf{RS}$ that is honest in S, every domain $dmnAS \in S(rs).\mathtt{clientResources}$, and every instance identifier $i \in S(rs).\mathtt{clientResources}[dmnAS]$ it holds true that if $n \equiv S(rs).\mathtt{clientResources}[dmnAS][i]$ is derivable from the attacker's knowledge in S (i.e., $n \in d_{\emptyset}(S(na))$), it follows that

(1) $\mathsf{dom}^{-1}(dmnAS)$ (the responsible AS) is corrupted in S, or

(2) the client instance $c = \mathsf{ownerOfResource}(n)$ that owns this resource is corrupted in S, or

(3) there exists a key record k in $s^{c}_{0}.\mathsf{keyRecords}[dmnAS']$ (for some domain $dmnAS' \in \mathsf{dom}(\mathsf{dom}^{-1}(dmnAS))$) such that $k.\mathtt{method} \equiv \mathtt{mac}$ and $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is corrupted in S (c shares a symmetric key with the responsible AS and a corrupted RS), or

(4) there exist a grant ID gid and a domain $y \in^{\langle\rangle} S(c).\mathtt{grants}[gid][\mathtt{bearerRSs}]$ such that $\mathtt{sessionID} \notin S(c).\mathtt{grants}[gid]$ (software-only authorization was used) and $\mathsf{dom}^{-1}(y)$ is corrupted in S (a bearer token was sent to a corrupted resource server).

Definition E.6 (Authorization Property for End Users). Let $\mathcal{GWS}$ be a GNAP web system. We say that *$\mathcal{GWS}$ fulfills the authorization property for end users* iff for every run ρ of $\mathcal{GWS}$, every configuration (S^j, E^j, N^j) in ρ, every RS $rs \in \mathsf{RS}$ that is honest in S^j, and every identity $u \in S^j(rs).\mathtt{userResources}$ it holds true that if

(1) $\mathsf{governor}(u)$ (the responsible AS) is honest in S^j,

(2) the browser $b = \mathsf{ownerOfResource}(n)$ that owns this resource is not fully corrupted in S^j,

(3) for all client instances c that are honest in S^j and all key records $k \in s^{c}_{0}.\mathsf{keyRecords}[dmnAS]$ (for some domain $dmnAS \in \mathsf{dom}(\mathsf{governor}(u))$) it holds true that $k.\mathtt{method} \not\equiv \mathtt{mac}$ or $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is honest in S^j (honest client instances share symmetric keys only with $\mathsf{governor}(u)$ and honest RSs), and

(4) there do not exist a client instance c, two grant IDs $CIgid$ and $ASgid$, and a processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$, such that $i < j$, $\mathsf{tryLogin}^{Q}_{\rho}(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), CIgid, ASgid)$, and

  (a) c is corrupted in S^j (a grant request from a corrupted client instance was granted), or

  (b) there exists a domain $y \in^{\langle\rangle} S^j(c).\mathtt{grants}[gid][\mathtt{bearerRSs}]$ such that $\mathsf{dom}^{-1}(y)$ is corrupted in S^j (an authorized client instance sent a bearer token to a corrupted RS),

then $n \equiv S^j(rs).\mathtt{userResources}[u]$ is not derivable from the attacker's knowledge in S^j (i.e., $n \notin d_{\emptyset}(S^j(na))$).

Definition E.7 (Session Integrity Property for Authorization (Software-only)). Let $\mathcal{GWS}$ be a GNAP web system. We say that *$\mathcal{GWS}$ is secure w.r.t. session integrity for software-only authorization* iff for every run ρ of $\mathcal{GWS}$, every processing step Q in ρ with

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

(for some S, S', E, E', N, N'), every client instance $c \in \mathsf{CI}$ that is honest in S, every rs that is honest in S, every nonces r and $CIgid$, every AS as, every instance ID $instanceID$ if $\mathsf{sw\_accessesResource}^{Q}_{\rho}(c, r, CIgid, rs, as, instanceID)$ then

(1) In some processing step Q' earlier than Q, there are a message m, an AS as' and a grant ID $ASgid$ such that $\mathsf{client\_started}^{Q'}_{\rho}(c, CIgid, m, as', ASgid)$

(2) If, additionally, as' is honest in S, then:

  (a) $as = as'$ — that is, the AS to which the client instance sent its initial request is the same as the AS controlling access to the resource that was sent to the client instance.

  (b) m contains the mapping $\mathtt{instanceID}: instanceID$ — that is, m was a request beginning a software-only GNAP flow for the client instance identified by $instanceID$.

Intuitively, if a client instance accesses some resource, then it previously started a flow requesting a resource. If the AS that it requested a resource from is honest, then the accessed resource is governed by the AS to which the initial request was made, and has the same instance ID that was used in the initial request.

This security property captures that (a) an end user of a client instance should only access resources when the end user actually expressed the wish to start a GNAP flow before and (b) if the end user expressed the wish to start a GNAP flow using some honest authorization server and a specific identity, then the end user is not accessing resources owned by a different identity. We note that for this, we require that the resource server which the client instance uses is honest, as otherwise, the attacker can trivially return any resource.

Definition E.8 (Session Integrity Property for Authorization (End-User)). Let $\mathcal{GWS}$ be a GNAP web system. We say that *$\mathcal{GWS}$ is secure w.r.t. session integrity for authorization for end users* iff for every run ρ of $\mathcal{GWS}$, every processing step Q in ρ with

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

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client instance $c \in \mathsf{CI}$ that is honest in S, every $rs \in \mathsf{RS}$ that is honest in S, every nonce r, and every nonce CIgid, we have that if $\mathsf{accessesResource}^Q_\rho(b, r, u, c, rs, \mathit{CIgid})$, then

1. there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_\rho(b, c, \mathit{CIgid}, as)$, and
2. Suppose that additionally, as and $\mathsf{governor}(u)$ are honest in S and for all client instances c' that are honest in S and all key records $r \in s^c_0.\mathtt{keyRecords}[\mathit{dmnAS}]$ (for some domain $\mathit{dmnAS} \in \mathsf{dom}(as)$) with $r.\mathtt{method} \equiv \mathtt{mac}$, it holds that $\mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S. Then there exists a grant ID ASgid and a processing step Q'' in ρ (before Q) such that $\mathsf{finishLogin}^{Q''}_\rho(b, c, u, as, \mathit{CIgid}, \mathit{ASgid})$.

E.3 Auxiliary Lemmas

Lemma E.9 (Tokens are only issued to clients that request them). *Suppose ρ is a run of a GNAP web system $\mathcal{GWS}$, and Q is a processing step in ρ from (S, E, N) to (S', E', N'). Suppose also that c is a client instance honest in S. We will suggestively denote by as^1, as^2, and as^3 several other processes that c treats as AS's.*

Suppose that in Q,

- *c receives a message $\tilde{m}$ from some process as^3, containing an access token tok.*
- *c stores tok into $S(c).\mathtt{receivedValues}[\mathit{CIgid}][\mathtt{accessToken}]$ on Line 29 of Algorithm A.7*
- *$S(as^2).\mathtt{tokenBindings}[tok][\mathtt{grantID}] \equiv \mathit{ASgid}$ (Note that this requires that as^2 was honest at the time that it issued tok).*

In other words, during step Q, c receives an access token issued by as^2 from as^3 and stores it.

Then,

- *There is an earlier step Q', a message $\hat{m}$, an AS as^1 and a grant ID ASgid' such that $\mathsf{client\_started}^{Q'}_\rho(c, \mathit{CIgid}, \hat{m}, as^1, \mathit{ASgid}')$.*
- *If as^1 is honest in S, then also $as^1 = as^2 = as^3$, and $\mathit{ASgid} \equiv \mathit{ASgid}'$.*

In other words, c previously requested an access token from some as^1, and if as^1 is honest, c indeed received a token from as^1 and the token was issued by as^1 for the same flow in which c received it.

Proof. Suppose c stores tok into $s'.\mathtt{receivedValues}[\mathit{CIgid}][\mathtt{accessToken}]$ on Line 29 of Algorithm A.7 and that $S(as^2).\mathtt{tokenBindings}[tok][\mathtt{grantID}] \equiv \mathit{ASgid}$, for some nonces CIgid and ASgid.

In order to reach this line, the check on Line 7 must succeed, and so we know that $\mathit{reference}[\mathtt{responseTo}] \equiv \mathtt{grantResponse}$. From Line 2, we learn that also $\mathit{reference}[\mathtt{grantID}] \equiv \mathit{CIgid}$.

By Lemma C.2, c previously called HTTPS_SIMPLE_SEND with $\mathit{reference}$ and some request m as the first two arguments. Moreover, if the process p such that $m.\mathtt{host} \in \mathsf{dom}(p)$ is honest, we can conclude that $p = as^3$. The reference for such a message can be set in eight different places: Lines 33 and 46 of Algorithm A.6, Lines 64 , 89, 99, 116 of Algorithm A.7, Line 13 of Algorithm A.8, and Line 17 of Algorithm A.10. In each case, the request m is either a grant request (sent to the `/requestGrant` endpoint of p) or a continuation request (sent to some other endpoint of p).

We consider first the cases where m is a grant request, and note that in this case, we can take $\hat{m} = m$ and $as^1 = p$ to conclude the first part of our result. Moreover, by Lemma D.11, $as^1 = as^3$. We then also claim that in each such case, if $as^1 = as^3$ is honest, then ASgid is created in response to m. First, we observe that an honest AS, upon receiving a grant request, immediately creates a new grant ID ASgid' on Line 3 of Algorithm A.13. Now, if the response to m contains an access token, since the `accessToken` field of a grant response is only written to in CREATE_GRANT_RESPONSE, and only a single call to this function exists in the `/requestGrant` endpoint, we see that CREATE_GRANT_RESPONSE must have been called on Line 34 of Algorithm A.13, with ASgid' as the first argument. Since CREATE_GRANT_RESPONSE uses this first argument as the grant ID associated with newly created tokens in `tokenBindings` (on Lines 16 ,19 or 25 of Algorithm A.17), we see that $S(as).\mathtt{tokenBindings}[tok][\mathtt{grantID}] \equiv \mathit{ASgid}'$ (Implicitly using that `tokenBindings` entries are never modified once written). This then means that $\mathit{ASgid}' \equiv \mathit{ASgid}$, and so we are done in this case (Note that $as^2 = as^3$, as an honest AS only emits tokens that it issued itself).

Now, we consider the cases where m is a continuation request. We claim in this case that there is some earlier processing step Q', an AS as^1 with a grant ID ASgid, and a message m' such that $\mathsf{client\_started}^{Q'}_\rho(c, \mathit{CIgid}, m', as^1, \mathit{ASgid})$. Moreover, if as^1 is honest, we see by Lemma D.11 that $as^1 = p$ (and so also $p = as^3$).

We then wish to invoke Lemma D.12 to show that m and m' are handled by as^1 under the same ASgid. For this, we need to know that m contains a continuation access token contAT with $S(as).\mathtt{grantRequests}[\mathit{ASgid}][\mathtt{continueAT}] \equiv \mathit{contAT}$.
At this point, we are done, concluding that m is the message that led to the creation of ASgid.

We will therefore proceed to show that c sent such an m'. If m is a continuation request, its reference was set in one of Lines 64, 99, 116 of Algorithm A.7, or Line 17 of Algorithm A.10, as the other four possible locations correspond to grant requests. We examine each of these cases in turn:

1. Case Line 64 of Algorithm A.7: In this case, we are again receiving a response with the same fields set in the *reference*, and simply fall back to that previous request, which is either again a continuation request (and we continue the argument with that request) or a grant request (which is then as desired). Formally, we can give an inductive argument to show that we eventually get back to the first such request, which must be as desired.
2. Case Line 99 of Algorithm A.7: In this case, we have received a response with $\mathit{reference}[\mathtt{responseTo}] \equiv \mathtt{MTLS\_CR}$ and the same CIgid as before. The request corresponding to this response can only have been sent on Line 73, and we fall back to the previous case.
3. Case Line 17 of Algorithm A.10: In this case, we are in the SEND_CONTINUATION_REQUEST function, which is only called at Lines 64 or 71 of Algorithm A.6. In both cases, this call only happens if the check on Lines 57 or 67 succeeds, meaning that $s'.\mathtt{grants}[\mathit{CIgid}][\mathtt{finishURLnonce}]$ exists and matches some specified nonce. In particular, this means that c must have written the finish URL nonce at Line 28, the only line where this field is written, at some earlier step.

Moreover, the request being processed must contain the same finish URL nonce. This means that c must have emitted the nonce at some prior step. There are only two places this can occur. In the first case, the nonce is emitted directly in a grant request in the first case of Algorithm A.6. The reference of this grant request contains *CIgid*, as we know that the grant ID used for this reference is the same as the grant ID used to store the finish URL nonce under `grants`. Otherwise, c stores the nonce (inside *grantRequest*) into $s'.\texttt{grants}$ $[\mathit{CIgid}][\texttt{request}]$ This field is only read from in full (or in portions that contain the finish URL nonce) on Line 86 of Algorithm A.7, after which the nonce is emitted (as part of the request) on Line 90, where it is part of a grant request. Here, again, the reference of this request contains *CIgid*, as the nonce is loaded from $s'.\texttt{grants}$ (on Line 3) using the same grant ID that is included in the reference.

(4) Case 116 of Algorithm A.7: In this case, we have received a response with $\mathit{reference}[\texttt{responseTo}] \equiv \texttt{MTLS\_CR}$ and the same *CIgid* as before. The request corresponding to this response is only sent in Line 33 of Algorithm A.10, and we can fall back to the previous case.

As in the case of a grant request, since we know that $as^1 = as^3$ and c receives an access token from as^3, if $as^1 = as^3$ is honest, it is also the case that $as^1 = as^2 = as^3$.

We now need only show that m contains a continuation access token *contAT* with $S(as).\texttt{grantRequests}[\mathit{ASgid}][\texttt{continueAT}] \equiv \mathit{contAT}$, and then Lemma D.12 tells us that *ASgid* was created in response to $\hat{m}$.

We know already that c received an access token *tok* from as^3 in response to the message m, and that (in this case) m is a continuation request. When handling m, an honest AS will attempt to load a continuation access token *contAT* from m Line 51 or 62 of Algorithm A.13, and then only proceeds to potentially emit a message if $\mathit{contAT} \equiv s'.\texttt{grantRequests}[\mathit{ASgid}'][\texttt{continueAT}]$ for some ASgid', as enforced by checks on Lines 53 and 64. As such, since as^3 replied to m, we can conclude that m contains such a continuation access token *contAT*, and we conclude. □

E.3.1 Software-only.

Lemma E.10 (Client Resources are returned only to owning Client Instance). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every RS $rs \in \mathsf{RS}$ that is honest in S, every $dmn \in S(rs).\texttt{clientResources}$, and every $\mathit{instanceID} \in S(rs).\texttt{clientResources}[dmn]$ it holds true that if $S(rs).\texttt{clientResources}[dmn][\mathit{instanceID}]$ is included as resource in the response m' in Line 45 of Algorithm A.21 by rs, then m' is a response to an HTTP request m sent by $c = \mathsf{ownerOfResource}(\mathit{resource})$ as long as*

(1) c is honest in S,
(2) the AS $as = \mathsf{dom}^{-1}(dmn)$ is honest in S,
(3) for all domains $dmnAS \in \mathsf{dom}(as)$ with $dmnAS \in s_0^c.\texttt{keyRecords}$ and all key records $r \in s_0^c.\texttt{keyRecords}[dmnAS]$ with $r.\texttt{method} \equiv \texttt{mac}$ it holds true that $\mathsf{dom}^{-1}(r.\texttt{rs})$ is honest in S, and
(4) for every grant ID gid in $S(c).\texttt{grants}$ for which it holds that $\texttt{sessionID} \notin S(c).\texttt{grants}[gid]$ (software-only authorization is used) and that $S(c).\texttt{grants}[gid][\texttt{AS}] \in \mathsf{dom}(\mathsf{dom}^{-1}(dmn))$, and every $dmnRS \in S(c).\texttt{grants}[gid][\texttt{bearerRSs}]$ it holds true that $\mathsf{dom}^{-1}(dmnRS)$ is honest in S.

Proof. If $\mathit{resource} \equiv S(rs).\texttt{clientResources}[dmn][\mathit{instanceID}]$ in Line 45 of Algorithm A.21, then *dmn* is the domain of the AS *as* to which *rs* sent the token introspection request in response to the resource request *m*. *instanceID* is the instance identifier returned by *as* in the response to the token introspection request under $[\texttt{access}][\texttt{instanceID}]$ (Line 34).

If Line 45 of Algorithm A.21 is executed, it must hold that in the introspection response the bearer flag has been set (Line 26) or *as* has returned information about the key to which the used access token is bound and *rs* has successfully validated the key proof against it (i.e. the call to VALIDATE_KEY_PROOF in Line 15, 22 or 24 has returned).

The introspection response that *rs* received in response to its introspection request must have been sent by *as* since only HTTPS is used for introspection requests (as for all requests). The introspection request contains the access token *accessToken* that *rs* received from the `Authorization` header of the resource request *m* (Line 11 resp. Line 16 of Algorithm A.20).

First, we consider the case that the resource request *m* was authorized using a bearer token. Since *as* is honest by precondition, *as* must have set the bearer flag in the introspection response in Line 137 of Algorithm A.13.

This only happens if $S'(as).\texttt{tokenBindings}[\mathit{accessToken}][\texttt{type}] \equiv \texttt{bearer}$ for some previous state S' (in Line 136 `bearer` is the only remaining type used). The sender of *m* must know the bearer token *accessToken* in S, otherwise it could not have included it inside the `Authorization` header. By Lemma D.6, only *as*, certain honest RSs, and the client instance that sent the grant request that led to the creation of the grant ID in $S'(as).\texttt{tokenBindings}[\mathit{accessToken}]$ $[\texttt{grantID}]$ are able to derive this bearer token. Thus, *m* must also have been sent by this client instance, since honest RSs and honest ASs do not send resource requests. During token introspection, this grant ID is loaded by *as* in Line 122 of Algorithm A.13 and then used in Line 123 to load the grant request. The stored instance identifier is then loaded from the grant request in Line 141, which is then sent to *rs* under $[\texttt{access}][\texttt{instanceID}]$, so this is the *instanceID* used by *rs*. This instance identifier must have been stored in Line 7 of Algorithm A.14 (PERFORM_KEY_PROOF) when the grant request was processed by *as* (since this is the only place where this happens). So *instanceID* must be the instance identifier of the client instance that sent the grant request to *as* and this client instance must have sent *m* (headers other than the `Authorization` header may differ, but are irrelevant). Together this means that the sender of *m* must also be $\mathsf{ownerOfResource}(\mathit{resource})$: If *resource* is not in the initial state of *rs*, this follows immediately as *rs* sends *resource* to *c* in m' as a response to *m*. If $\mathit{resource} \equiv s_0^{rs}.\texttt{clientResources}[dmn][\mathit{instanceID}]$, then, by the definition of the initial state of *rs*, *instanceID* is an entry in $s_0^{rs}.\texttt{instanceIDs}[dmn]$, and further there has to be a client registration record r in $s_0^{as}.\texttt{registrations}$ with $\mathit{instanceID} \equiv r.\texttt{instanceID}$. Finally, from the initial state of *as*, there must be exactly one client instance c' with a key record r' in $s_0^{c'}.\texttt{keyRecords}[dmn]$ where $r.\texttt{instanceID} \equiv r'.\texttt{instanceID} \equiv$

instanceID. This CI is the sender of m and at the same time the owner of *resource*.

Now we consider the case that the resource request m was authorized using a key-bound access token. In this case, VALIDATE_KEY_PROOF must have returned in on one of the following lines of Algorithm A.21:

- Line 15: In this line, VALIDATE_KEY_PROOF validates a MAC. The key for this key proof is loaded by *rs* in Line 14 from the *symKeys* subterm. The instance identifier used was returned by *as* in the introspection response in the [instanceID] entry (using HTTPS). This must be the same instance identifier that was returned by *as* under [access] [instanceID] (*instanceID*), since *as* uses the value of *grantRequest*[instanceID] for both (Line 133 and Line 141 of Algorithm A.13). Thus, the instance identifier for whose key the key proof is validated is the instance identifier whose resource is returned. Since VALIDATE_KEY_PROOF must have returned, it thus holds, using Lemma D.8, that m was sent by ownerOfResource(*resource*) (headers other than the Authorization header may differ, but are irrelevant).
- Line 22: In this line, VALIDATE_KEY_PROOF validates a signature. The key for this key proof is returned by *as* in the introspection response under [key][key] (Line 19). This can be either the key used by the client instance in the grant request (returned by *as* in Line 131 of Algorithm A.13 whereby the returned value was stored in Line 30 of Algorithm A.14) or a different key chosen by AS for binding to this access token (returned in Line 125 of Algorithm A.13).
  If the key is the key from the grant request, this must be the key associated with *instanceID*. As before, the instance identifier whose key is used in the key proof is the instance identifier whose resource is returned. Since VALIDATE_KEY_PROOF returned using this key, it must hold according to Lemma D.8 that m was sent by the client whose resource is returned (again, headers other than the Authorization header may differ, but are irrelevant).
  If the key is a different key chosen by *as* for this access token, it must have been loaded in Line 125 of Algorithm A.13 from $S'(as)$.tokenBindings[*accessToken*][publicKey] for a previous state S'. This key is chosen from a clientTokenKeys entry in Line 12 of Algorithm A.17. The used instance identifier is taken from grantRequests[*grantID*][instanceID], which is *instanceID*. Since VALIDATE_KEY_PROOF returned, the signature validation in Line 21 of the algorithm must have been successful. In particular, the request m must have been sent by a process, knowing the corresponding private key. By Lemma D.4 and the initial states of ASs and CIs, only the client instance associated with *instanceID* knows this private key. As before, using Lemma D.8 yields that m must thus have been sent by this client instance. At the same time *instanceID* is used by *rs* to chose the right resource. In total, we obtain that m was sent by ownerOfResource(*resource*).
- Line 24: In this line, VALIDATE_KEY_PROOF validates an MTLS key proof, which in this context behaves like a key proof for a signature (see the previous point), since public keys are used in both cases and the lemmas used for the proof are independent of whether the key proof is based on MTLS or signatures.

Overall, the lemma must apply to both bearer tokens and key-bound access tokens, and thus all forms of access tokens. □

E.3.2 End-User.

Lemma E.11 (User Resource is returned only if Request contains matching Access Token). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every RS $rs \in$ RS that is honest in S, and every identity $u \in S(rs)$.userResources it holds true that if $S(rs)$.userResources$[u]$ is included as resource in the response m' in Line 45 of Algorithm A.21 by rs, then m' is a response to an HTTP request m for which the following holds true*

1. *m contains an* Authorization *header $\langle type, AT \rangle$,*
2. *when processing m in A.20, rs sends an introspection request m^{req}_{intro} to as* = governor (u), *and when processing the corresponding response m^{resp}_{intro} from as, rs reaches Line 45 of Algorithm A.21,*
3. *if as is honest in S, as is the issuer of the token AT, and*
4. *if as is honest in S we have for*
   gid $\equiv S(as)$.tokenBindings$[AT]$[grantID] *that*
   $S(as)$.grantRequests$[gid]$[subjectID] $\equiv u$.

Proof. If $S(rs)$.userResources$[u]$ is included as *resource* in the response m' in Line 45 of Algorithm A.21 by *rs*, it must hold that *resource* was loaded in Line 32, since this is the only line where *rs* loads a user resource. *rs* reaches this line only when processing an introspection response $m^{\text{resp}}_{\text{intro}}$. The corresponding introspection request $m^{\text{req}}_{\text{intro}}$ must have been sent by *rs* in Line 13 of Algorithm A.20 or Line 56 of Algorithm A.21, since these are the only lines where introspection requests are sent. Those lines are only reached if m contains an Authorization header $\langle type, AT \rangle$ including an access token AT: For Line 13 of A.20 this is immediate from the check in Line 3. For Line 56 of A.21 we observe that the corresponding MTLS request must have been sent in Line 19 of A.20 and thus the same check in Line 3 shows Item (1).

We further observe that the introspection request $m^{\text{req}}_{\text{intro}}$ must have been sent to governor (u): The *identity* used in Line 32 of A.21 is u, and hence by the check in Line 31 u is an element of $S(rs)$.identities[*domainAS*] where *domainAS* is $S(rs)$.resourceRequests[*requestID*][AS]. As the *identities* subterm cannot change we have from the conditions on the initial states of *rs* and $as' = \mathsf{dom}^{-1}(domainAS)$ that there is a user record $rec \in s^{as'}_0$.users with $u \equiv rec$.identity and u.domain $\equiv rec$.identity.domain $\in \mathsf{dom}(as')$. Thus *domainAS* is a domain of governor (u).
Further, the domain saved in resourceRequests[*requestID*][AS] is the same domain that is used to send the introspection request to. Hence, when processing m in A.20, *rs* sends an introspection request $m^{\text{req}}_{\text{intro}}$ to as = governor (u). And since *rs* reaches Line 45 of Algorithm A.21, we get Item (2).

In particular, since *rs* reaches Line 32 of Algorithm A.21, the response must have included active : $\top$. If *as* is honest, it only includes active : $\top$ in the response if it is the issuer of the token AT, showing Item (3).

For Item (4) we see that the identity u used in Line 32 of A.21 is retrieved from $\mathit{response}[\texttt{access}][\texttt{identity}]$ (Line 30), where $\mathit{response} = m^{\text{resp}}_{\text{intro}}$ is the introspection response received by rs from as. Since as is honest by precondition, as must have included the identity u in the introspection response in Line 139 of Algorithm A.13. The value used here for u is taken from $S'(as).\texttt{grantRequests}[\mathit{grantID}][\texttt{subjectID}]$, where $\mathit{grantID} \equiv S'(as).\texttt{tokenBindings}[\mathit{accessToken}][\texttt{grantID}]$ (Line 122) for some previous state S' and the $\mathit{accessToken}$ that was transmitted to as by rs. As seen in Line 11 resp. Line 16 and Line 3 of A.20, this access token passed from rs to as is the token taken from the $\texttt{Authorization}$ header of m, i.e., $\mathit{accessToken} = AT$. This proves the lemma, since the values of $S'(as).\texttt{grantRequests}[\mathit{grantID}][\texttt{subjectID}]$ and $S'(as).\texttt{tokenBindings}[AT][\texttt{grantID}]$ cannot be overwritten and thus must still be the same in S. □

Lemma E.12 (User Resources are returned only to authorized Client Instances). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S^j, E^j, N^j) in ρ, every RS $rs \in \mathsf{RS}$ that is honest in S^j, and every identity $u \in S^j(rs).\texttt{userResources}$ it holds true that if $S^j(rs).\texttt{userResources}[u]$ is included as resource in the response m' in Line 45 of Algorithm A.21 by rs, then m' is a response to an HTTP request m sent by a client instance c for which it holds true that there exists a grant ID ASgid and a processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$, such that $i < j$ and $\mathsf{tryLogin}^{Q}_{\rho}(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{gid}, \mathit{ASgid})$ (with gid being the grant ID of the grant in whose context c sent m) as long as*

1. *(1) $\mathsf{governor}(u)$ is honest in S^j,*
2. *(2) $\mathsf{ownerOfResource}(\mathit{resource})$ is not fully corrupted in S^j,*
3. *(3) for all client instances c' that are honest in S^j, all domains $\mathit{dmnAS} \in \mathsf{dom}(\mathsf{governor}(u))$ and all key records $k \in s^{c'}_0.\texttt{keyRecords}[\mathit{dmnAS}]$ it holds true that $k.\texttt{method} \not\equiv \texttt{mac}$ or $\mathsf{dom}^{-1}(k.\texttt{rs})$ is honest in S^j, and*
4. *(4) there do not exist c', gid', ASgid', and $Q' = (S^{i'}, E^{i'}, N^{i'}) \rightarrow (S^{i'+1}, E^{i'+1}, N^{i'+1})$, such that $i' < j$, $\mathsf{tryLogin}^{Q'}_{\rho}(\mathsf{ownerOfID}(u), c', u, \mathsf{governor}(u), \mathit{gid}', \mathit{ASgid}')$, and*
   - *(a) c' is corrupted in S^j, or*
   - *(b) there exists a domain $y \in^{\langle\rangle} S^j(c').\texttt{grants}[\mathit{gid}'][\texttt{bearerRSs}]$ such that $\mathsf{dom}^{-1}(y)$ is corrupted in S^j.*

Proof. To prove this lemma, we will show several things:

- (I) $\mathsf{ownerOfID}(u)$ must have loaded a document from an origin of $\mathsf{governor}(u)$, executed the script *script_as_login* in that document, and in that script, in Line 10 of Algorithm A.19, selected the identity u,
- (II) c is a client instance that sent a grant request to $\mathsf{governor}(u)$ in a flow in which (I) occurred, and
- (III) m is sent in the same flow as that grant request.

(I) and (II) together imply that $\mathsf{tryLogin}^{Q}_{\rho}(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{gid}, \mathit{ASgid})$ holds for some previous processing step Q and some grant IDs gid and ASgid. (III) ensures that gid is the grant ID of the grant in whose context c sent m.[7]

If $\mathsf{tryLogin}^{Q}_{\rho}(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{gid}, \mathit{ASgid})$ holds, this also means that c is honest in S^j due to precondition (4). This in turn implies, due to precondition (3), that c does not share any of the symmetric keys it shares with $\mathsf{governor}(u)$ with a corrupted RS.

Let $as = \mathsf{governor}(u)$. By the first precondition we have that as is honest.

We start with showing (I). By Lemma E.11, m must have contained an access token $\mathit{accessToken}$ in the $\texttt{Authorization}$ header, such that $S^j(as).\texttt{grantRequests}[\mathit{grantID}][\texttt{subjectID}] \equiv u$ with $\mathit{grantID} \equiv S^j(as).\texttt{tokenBindings}[\mathit{accessToken}][\texttt{grantID}]$. The $[\texttt{subjectID}]$ entry must have been written in Line 17 of Algorithm A.15 since this is the only line where this entry is written. Since u is written in Line 17, it must have hold in Line 8 that $\mathit{password} \equiv s^{c}_{0}.\texttt{users}[u] \equiv \mathsf{secretOfID}(u)$ (the entries of the *users* subterm of an honest AS never change). The value of *password* is taken from the parameter m (Line 4). m must be a request received by as at the $\texttt{/redirectLogin}$ path or at the $\texttt{/userCodeLogin}$ path, since only in these sections Algorithm A.15 is called. According to Lemma D.7, only $\mathsf{ownerOfID}(u)$ and as are able to derive $\mathsf{secretOfID}(u)$ in S^j. Since as does not send requests, m must thus have been sent by $\mathsf{ownerOfID}(u)$. Because $\mathsf{secretOfID}(u)$ was contained in m, based on our browser model, $\mathsf{ownerOfID}(u)$ must have loaded a document from an origin of as, executed the script contained in it, and then this script must have sent $\mathsf{secretOfID}(u)$ to the $\texttt{/redirectLogin}$ path or the $\texttt{/userCodeLogin}$ path. This script must be *script_as_login* as this is the only script used by as and HTTPS is used. Furthermore, since u was written in Line 17 of Algorithm A.15, and u was taken from the $[\texttt{identity}]$ subterm of m (Line 3), it must hold that $\mathsf{ownerOfID}(u)$ selected u in Line 10 of Algorithm A.19 (*script_as_login*), proving (I).

We will show (II) by showing that since c is able to use an access token that must have been created in a flow in which (I) occurred, c must also have sent the grant request to $\mathsf{governor}(u)$ in that flow. Using Lemma E.11, we know that the request m must contain an access token $\mathit{accessToken}$ in the $\texttt{Authorization}$ header, such that for the grant ID $\mathit{grantID} \equiv S^j(as).\texttt{tokenBindings}[\mathit{accessToken}][\texttt{grantID}]$, it holds true that $S^j(as).\texttt{grantRequests}[\mathit{grantID}][\texttt{subjectID}] \equiv u$. Therefore, as seen in the previous paragraph, (I) must hold for the flow in which this access token was created. As in the proof for Lemma E.10, the access token can be either a bearer access token or a key-bound access token.

First, let's consider the case of a bearer access token. If the access token is a bearer token, it must have been stored in Line 25 of Algorithm A.17 in $S^{i'}(as).\texttt{tokenBindings}$ for some previous state $S^{i'}$. Since the entries for an access token in the *tokenBindings* subterm cannot change, it must hold that $S^j(as).\texttt{tokenBindings}[\mathit{accessToken}][\texttt{type}] \equiv \texttt{bearer}$. We now observe that Item (I) means that $\mathsf{tryLogin}^{Q}_{\rho}(\mathsf{ownerOfID}(u), c', u, \mathsf{governor}(u), \mathit{gid}, \mathit{ASgid})$ holds for some c', gid. and ASgid. We can then conclude that this c' must be honest in S^j from precondition (4). By Lemma D.6 it must thus hold that the bearer token is derivable in S^j only for as, c', and some RSs that must be honest according to precondition (4). The sender of m, c, must obviously be able to derive the bearer token in S^j. Since as, being an honest AS, does not send requests and honest RSs only send introspection requests (but m is a resource

[7] This is required to rule out the stolen token replay attack, for example.

request), c must be the client instance c' that sent the grant request to $as = \text{governor}(u)$.

Now we consider the case that the resource request m was authorized using a key-bound access token. In this case, the call of VALIDATE_KEY_PROOF must have returned in one of the following lines of Algorithm A.21:

- Line 15: In this line, VALIDATE_KEY_PROOF validates a MAC. The key for this key proof is loaded by *rs* in Line 14 from the *symKeys* subterm. The instance identifier used was returned by *as* in the introspection response in the [instanceID] entry (using HTTPS). *as* must have selected the value returned under [instanceID] in Line 133 of Algorithm A.13 as this is the only line where this entry is written. The value chosen is the instance identifier specified in the grant request of this run, which must have been stored in Line 7 of Algorithm A.14 when *as* processed the grant request. So when *rs* loads the symmetric key from the *symKeys* subterm in Line 14 of Algorithm A.21, it must be the key of the client instance that sent the grant request to *as*, since the values in the *symKeys* subterm cannot change. Since VALIDATE_KEY_PROOF must have returned, according to Lemma D.8, m must have been sent by the client instance that sent the grant request to *as* (headers other than the Authorization header may differ, but are irrelevant). Lemma D.8 can be applied in this proof because by precondition (4) $\text{ownerOfID}(u)$ authorizes only honest client instances, and by precondition (3) honest client instances do not use symmetric keys shared with *as* and a corrupted RS.
- Line 22: In this line, VALIDATE_KEY_PROOF validates a signature. The key for this key proof is returned by *as* in the introspection response under [key][key] (Line 19). This can be either the key used by the client instance in the grant request (returned by *as* in Line 131 of Algorithm A.13 whereby the returned value was stored in Line 30 of Algorithm A.14) or a different key chosen by AS for binding to this access token (returned in Line 125 of Algorithm A.13).
  If the key is the key from the grant request, it must hold by Lemma D.8 that m was sent by the client instance that sent the grant request since VALIDATE_KEY_PROOF returned using the key from the grant request (again, headers other than the Authorization header may differ, but are irrelevant).
  If the key is a different key chosen by *as* for this access token, it must have been loaded in Line 125 of Algorithm A.13 from $S^{i'}(as)$.tokenBindings[*accessToken*][publicKey] for a previous state $S^{i'}$. This key is chosen from a clientTokenKeys entry in Line 12 of Algorithm A.17. The *instanceID* key used for this entry is taken from the initial grant request, in particular, it is an instance identifier of a client instance that sent the grant request. Again, by Lemma D.8 we get that m was sent by the client instance that sent the grant request
- Line 24: In this line, VALIDATE_KEY_PROOF validates an MTLS key proof, which in this context behaves like a key proof for a signature (see the previous point), since public keys are used in both cases and the lemmas used for the proof are independent of whether the key proof is based on MTLS or signatures.

Thus, (II) must apply to both bearer tokens and key-bound access tokens.

Finally, we will prove (III). In principle, c as an honest client instance uses an access token received from an AS only in the flow in which c also received the access token. This is ensured in the code by storing a received access token in Line 29 of Algorithm A.7 under the used grant ID in the receivedValues subterm, so that when the access token is used in Algorithm A.8, it can be uniquely associated with the flow in which the access token was received. However, this does not mean that the access token received from the AS was also issued for this flow, which is exploited in the stolen token replay attack, for example. If c received the access token contained in m directly from *as* in the flow in which c sent m, it must have been issued for this flow as well, since an honest AS always returns only newly created access tokens (Line 6 resp. Line 24 of Algorithm A.17). If c received an access token issued by *as* from a corrupted AS *cas* in this flow, we must again distinguish by the type of access token. In the following, we will therefore show for each type of access token that it cannot happen that *rs* sends m' in response to m if the access token contained in m was issued by *as* but was transmitted to c by a corrupted AS in a flow other than the flow in which the access token was issued by *as*.

Since bearer tokens do not leak (Lemma D.6), a corrupted AS cannot send a bearer token created by *as* to c.

Now consider the case, where the access token in m is bound to the key that c used for the grant request. We assume that c received the access token that was issued by *as* from a corrupted AS and included this access token in the Authorization header of m. In the introspection response, *as* will return the public key c used in its grant request to *as* in the flow in which the access token was issued (or the instance ID of c if symmetric keys are used). This holds since the public key resp. the instance ID are loaded from the grant request in Line 131 resp. Line 133 of Algorithm A.13. Thus, *rs* will use a key for validating the signature resp. MAC that c uses only for *as*, since an honest client instance by definition uses a specific key only for one single AS. At the same time, however, c will use the key it used in the grant request to the corrupted AS from which it received the access token when creating the signature. This key must therefore be different from the key c uses for *as*, so VALIDATE_KEY_PROOF will not return in Line 15 or Line 22 of Algorithm A.21 and *rs* will not send m'. The same reasoning can be applied to the use of MTLS and Line 24 of Algorithm A.21.

Finally, in the case where the access token is bound to a different key of c, the corrupted AS can only send a key belonging to the corrupted AS along with it. Otherwise, the check by c in Line 27 of Algorithm A.7 prevents c from storing (and using) the token. Similarly to the case of the token being bound to the client instance's key, c uses different keys for different ASs and hence as above the key proof performed by *rs* can not succeed.

This concludes the proof. □

Lemma E.13 (Receipt of a User Resource implies that the RO was present). *For any run ρ of a GNAP web system $\mathcal{GWS}$, every configuration (S, E, N) in ρ, and every client instance $c \in$ CI that is honest in S it holds true that if the client instance c stores m.*body *under* $S(c).\texttt{grants}[\mathit{grantID}][\texttt{resources}][\mathit{domainRS}]$ *in Line 78 of Algorithm A.7 (for some m, grantID, domainRS) where* $m.\mathsf{body} \equiv s_0^{rs}.\texttt{userResources}[u]$ *for* $rs = \mathsf{dom}^{-1}(\mathit{domainRS})$ *and some identity* u*, then* $\mathsf{ownerOfResource}(m.\mathsf{body})$ *led to the creation of grantID at c as long as*

(1) rs is honest in S,
(2) $\mathsf{ownerOfResource}(m.\mathsf{body})$ *(=* $\mathsf{ownerOfID}(u)$*) is not fully corrupted in S,*
(3) $\mathsf{governor}(u)$ *is honest in S, and*
(4) for all client instances c' *that are honest in S, all domains* $\mathit{dmnAS} \in \mathsf{dom}(\mathsf{governor}(u))$*, and all key records* $k \in s_0^{c'}.\texttt{keyRecords}[\mathit{dmnAS}]$ *it holds true that* $k.\texttt{method} \not\equiv \texttt{mac}$ *or* $\mathsf{dom}^{-1}(k.\texttt{rs})$ *is honest in S.*
(5) there do not exist a client instance c, two grant IDs CIgid and ASgid, and a processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$*, such that* $i < j$*,* $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{CIgid}, \mathit{ASgid})$*, and*
  (a) c is corrupted in S^j *(a grant request from a corrupted client instance was granted), or*
  (b) there exists a domain $y \in^{\langle\rangle} S^j(c).\texttt{grants}[\mathit{gid}][\texttt{bearerRSs}]$ *such that* $\mathsf{dom}^{-1}(y)$ *is corrupted in* S^j *(an authorized client instance sent a bearer token to a corrupted RS),*

Proof. In Line 78 of Algorithm A.7, *domainRS* is the domain of *rs* to which c sent the resource request that was answered by m, which must hold since *domainRS* was taken from the reference of the request in Line 77 together with Lemma C.2. Thus, due to the use of HTTPS, $m.\mathsf{body}$ must have been obtained from the honest RS *rs*. *rs* must have sent m in Line 46 of Algorithm A.21, since only in this line matching responses are sent by an honest RS. Since m contains $s_0^{rs}.\texttt{userResources}[u]$, according to Lemma E.12, it must hold that there is a grant ID *ASgid* and a processing step Q such that $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID}, \mathit{ASgid})$.[8]

c can only store a resource in $S(c).\texttt{grants}[\mathit{grantID}][\texttt{resources}][\mathit{domainRS}]$ in Line 78 of Algorithm A.7 if c has sent a resource request in Algorithm A.8. This in turn can only happen if the grant identified by *grantID* has been authorized and c has received an access token from $\mathsf{governor}(u)$. c will only receive an access token from $\mathsf{governor}(u)$ after the interaction is finished, so it must hold that c called SEND_CONTINUATION_REQUEST($\mathit{grantID}, \mathit{interactRef}, \mathit{hash}, s', a$) (for some $\mathit{interactRef}$, hash, s', a) in a processing step after Q in order to finish the interaction. Using Lemma D.15, it must thus hold that $\mathsf{ownerOfID}(u)$ sent the request to the /startGrantRequest path of c that led to the creation of *grantID* in Line 9 of Algorithm A.6. Since by definition $\mathsf{ownerOfResource}(m.\mathsf{body}) = \mathsf{ownerOfID}(u)$, the lemma is shown. □

Lemma E.14 (Honest Client Instances return Resources only to the Resource Owner). *For any run* ρ *of a GNAP web system* $\mathcal{GWS}$*, every configuration* (S, E, N) *in* ρ*, and every client instance* $c \in \mathsf{CI}$ *that is honest in S it holds true that if c emits an event in Line 18 of Algorithm A.11 in the processing step* $(S, E, N) \rightarrow (S', E', N')$ *(for some configuration* (S', E', N')*) that contains an HTTP response* m' *whose body contains a nonce n (as a subterm) for which it holds that* $b = \mathsf{ownerOfResource}(n)$ *for some browser b, then* m' *must be a response to an HTTP request m sent by b as long as*

(1) b is not fully corrupted in S,
(2) the RS $rs = \mathsf{dom}^{-1}(S(c).\texttt{grants}[\mathit{grantID}][\texttt{domainFirstRS}])$ *(with grantID being the grantID that was passed to Algorithm A.11) from which c received n is honest in S,*
(3) with u being the identity for which $s_0^{rs}.\texttt{userResources}[u] \equiv n$*, it holds true that* $\mathsf{governor}(u)$ *is honest in S, and*
(4) for all client instances c' *that are honest in S, all domains* $\mathit{dmnAS} \in \mathsf{dom}(\mathsf{governor}(u))$*, and all key records* $k \in s_0^{c'}.\texttt{keyRecords}[\mathit{dmnAS}]$ *it holds true that* $k.\texttt{method} \not\equiv \texttt{mac}$ *or* $\mathsf{dom}^{-1}(k.\texttt{rs})$ *is honest in S.*
(5) there do not exist a client instance c, two grant IDs CIgid and ASgid, and a processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$*, such that* $i < j$*,* $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{CIgid}, \mathit{ASgid})$*, and*
  (a) c is corrupted in S^j *(a grant request from a corrupted client instance was granted), or*
  (b) there exists a domain $y \in^{\langle\rangle} S^j(c).\texttt{grants}[\mathit{gid}][\texttt{bearerRSs}]$ *such that* $\mathsf{dom}^{-1}(y)$ *is corrupted in* S^j *(an authorized client instance sent a bearer token to a corrupted RS),*

Proof. In Line 18 of Algorithm A.11 we have that *sender* and *receiver* as well as the *key* and the *nonce* used in m' have been loaded from $S(c).\texttt{browserRequests}[\mathit{grantID}][\texttt{finishRequests}]$ (Line 15). The finishRequest entry is written only in Line 61 and Line 79 of Algorithm A.6. Since these are the only sections where the value of this entry is written, Lemma D.14 implies that m must have been sent by the same process that sent the request to the /startGrantRequest path of c that led to the creation of *grantID* in Line 9 of Algorithm A.6.

The resource n is loaded from $S(c).\texttt{grants}[\mathit{grantID}][\texttt{resources}][\mathit{domainFirstRS}]$ in Line 12 of Algorithm A.11 (with $\mathit{domainFirstRS} = S(c).\texttt{grants}[\mathit{grantID}][\texttt{domainFirstRS}]$). This value can only have been stored in Line 78 of Algorithm A.7. Thus, by Lemma E.13, it must hold that the request to the /startGrantRequest path of c that led to the creation of *grantID* in Line 9 of Algorithm A.6 must have been sent by $\mathsf{ownerOfResource}(n) = b$.

Together this means that m must have been sent by b. □

E.4 Security w.r.t. Authorization

Lemma E.15 (Authorization Property for Software-only Authorization). *Let* $\mathcal{GWS}$ *be a GNAP web system. We say that* $\mathcal{GWS}$ fulfills the authorization property for software-only authorization *iff for every run* ρ *of* $\mathcal{GWS}$*, every configuration* (S, E, N) *in* ρ*, every RS* $rs \in \mathsf{RS}$ *that is honest in S, every domain* $\mathit{dmnAS} \in S(rs).\texttt{clientResources}$*, and every instance identifier* $i \in S(rs).\texttt{clientResources}[\mathit{dmnAS}]$ *it holds true that if* $n \equiv S(rs).\texttt{clientResources}[\mathit{dmnAS}][i]$ *is derivable from the attacker's knowledge in S (i.e.,* $n \in d_\emptyset(S(na))$*), it follows that*

(1) $\mathsf{dom}^{-1}(\mathit{dmnAS})$ *(the responsible AS) is corrupted in S, or*
(2) the client instance $c = \mathsf{ownerOfResource}(n)$ *that owns this resource is corrupted in S, or*
(3) there exists a key record k in $s_0^c.\texttt{keyRecords}[\mathit{dmnAS}']$ *(for some domain* $\mathit{dmnAS}' \in \mathsf{dom}(\mathsf{dom}^{-1}(\mathit{dmnAS}))$*) such that*

[8] Since c is honest in S, we can ignore precondition (4) of Lemma E.12, since an honest client instance will not use leaked bearer tokens or the keys of other corrupted client instances.

$k.\mathtt{method} \equiv \mathtt{mac}$ *and* $\mathsf{dom}^{-1}(k.\mathtt{rs})$ *is corrupted in S (c shares a symmetric key with the responsible AS and a corrupted RS), or*

(4) *there exist a grant ID gid and a domain* $y \in^{\langle\rangle} S(c).\mathtt{grants}[gid][\mathtt{bearerRSs}]$ *such that* $\mathtt{sessionID} \notin S(c).\mathtt{grants}[gid]$ *(software-only authorization was used) and* $\mathsf{dom}^{-1}(y)$ *is corrupted in S (a bearer token was sent to a corrupted resource server).*

Proof. We prove this lemma using proof by contradiction. We assume that $n \in d_\emptyset(S(na))$ and that

1. $\mathsf{dom}^{-1}(dmnAS)$ is honest in S,
2. $c = \mathsf{ownerOfResource}(n)$ is honest in S,
3. for all domains $dmnAS' \in \mathsf{dom}(\mathsf{dom}^{-1}(dmnAS))$ with $dmnAS' \in s_0^c.\mathtt{keyRecords}$ and all key records $k \in s_0^c.\mathtt{keyRecords}[dmnAS']$ with $k.\mathtt{method} \equiv \mathtt{mac}$ it holds true that $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is honest in S, and
4. there does not exist a grant ID gid and a domain $y \in^{\langle\rangle} S(c).\mathtt{grants}[gid][\mathtt{bearerRSs}]$ such that $\mathtt{sessionID} \notin S(c).\mathtt{grants}[gid]$ and $\mathsf{dom}^{-1}(y)$ is corrupted in S.

By the definitions of the initial states, n must be initially stored in *rs* only, or it was created before S in Line 36 of Algorithm A.21 and then stored in the state of *rs* in Line 38 (and therefore not contained in any of the initial states). In any case, there must have been a state before S in which n was stored in *rs* only. Since *rs* is honest by precondition, it sends n only in responses to resource requests in Line 45 of Algorithm A.21. Since all the conditions for Lemma E.10 are satisfied, it must hold that *rs* thereby sends n only in response to a request from c. Because HTTPS is used for this response, only c is able to decrypt the response, so *rs* leaks n only to c, but no other process. As c is honest and does not emit events containing received resources when using software-only authorization, it is also not possible for c to leak n to any other process. Thus, in S, n can only be derivable for the two honest processes c and *rs*, which means that, contrary to our assumption, $n \notin d_\emptyset(S(na))$ must hold. □

Lemma E.16 (Authorization Property for End Users). *Let* $\mathcal{GWS}$ *be a GNAP web system. We say that* $\mathcal{GWS}$ fulfills the authorization property for end users *iff for every run* ρ *of* $\mathcal{GWS}$*, every configuration* (S^j, E^j, N^j) *in* ρ*, every RS* $rs \in \mathsf{RS}$ *that is honest in* S^j*, and every identity* $u \in S^j(rs).\mathtt{userResources}$ *it holds true that if*

(1) $\mathsf{governor}(u)$ *(the responsible AS) is honest in* S^j*,*

(2) *the browser* $b = \mathsf{ownerOfResource}(n)$ *that owns this resource is not fully corrupted in* S^j*,*

(3) *for all client instances c that are honest in* S^j *and all key records* $k \in s_0^c.\mathtt{keyRecords}[dmnAS]$ *(for some domain* $dmnAS \in \mathsf{dom}(\mathsf{governor}(u))$*) it holds true that* $k.\mathtt{method} \not\equiv \mathtt{mac}$ *or* $\mathsf{dom}^{-1}(k.\mathtt{rs})$ *is honest in* S^j *(honest client instances share symmetric keys only with* $\mathsf{governor}(u)$ *and honest RSs), and*

(4) *there do not exist a client instance c, two grant IDs CIgid and ASgid, and a processing step* $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$*, such that* $i < j$*,* $\mathsf{tryLogin}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), CIgid, ASgid)$*, and*

(a) *c is corrupted in* S^j *(a grant request from a corrupted client instance was granted), or*

(b) *there exists a domain* $y \in^{\langle\rangle} S^j(c).\mathtt{grants}[gid][\mathtt{bearerRSs}]$ *such that* $\mathsf{dom}^{-1}(y)$ *is corrupted in* S^j *(an authorized client instance sent a bearer token to a corrupted RS),*

then $n \equiv S^j(rs).\mathtt{userResources}[u]$ *is not derivable from the attacker's knowledge in* S^j *(i.e.,* $n \notin d_\emptyset(S^j(na))$*).*

Proof. For a contradiction assume that $n \in d_\emptyset(S^j(na))$. Since, by definition, n is initially stored in *rs* only, it must hold for all processes $p \neq rs$ that $n \notin d_\emptyset(s_0^p)$. Since *rs* is honest by precondition, it emits n only in responses to resource requests in Line 45 of Algorithm A.21. As all conditions for Lemma E.12 are given, it must hold that *rs* sends n only to client instances c for which $\mathsf{tryLogin}_\rho^{Q'}(b, c, u, \mathsf{governor}(u), gid, ASgid)$ holds for some grant IDs *gid* and *ASgid*, and some previous processing step Q'. Due to assumption (a), it must hold for these client instances to be honest in S^j. Due to the use of HTTPS, only these client instances can decrypt the resource responses sent by *rs*, so n is not leaked to any other process when n is transferred from *rs* to client instances. Honest client instances emit events that may contain user resources only in Line 18 of Algorithm A.11. By Lemma E.14, it must therefore hold that all possible client instances c that may have received n from *rs* emit n only in a response to b using HTTPS. Since b is not fully corrupted in S^j and HTTPS is used, only b can decrypt the response, so all c can leak n only to b. A browser that is not fully corrupted does not process received resources any further and, in particular, does not resend them. Thus, in S^j, n can only be derivable for b, *rs*, and some honest client instances, which means that, contrary to our assumption, $n \notin d_\emptyset(S^j(na))$ must hold. □

E.5 Session Integrity for Authorization

Lemma E.17 (Session Integrity for Software-only Authorization). *Let* $\mathcal{GWS}$ *be a GNAP web system. We say that* $\mathcal{GWS}$ *is* secure w.r.t. session integrity for software-only authorization *iff for every run* ρ *of* $\mathcal{GWS}$*, every processing step Q in* ρ *with*

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

(for some S, S', E, E', N, N'*), every client instance* $c \in \mathsf{CI}$ *that is honest in S, every rs that is honest in S, every nonces r and CIgid, every AS as, every instance ID instanceID if* $\mathsf{sw\_accessesResource}_\rho^Q(c, r, CIgid, rs, as, instanceID)$ *then*

(1) *In some processing step* Q' *earlier than Q, there are a message m, an AS* as' *and a grant ID ASgid such that* $\mathsf{client\_started}_\rho^{Q'}(c, CIgid, m, as', ASgid)$

(2) *If, additionally,* as' *is honest in S, then:*

(a) $as = as'$ *— that is, the AS to which the client instance sent its initial request is the same as the AS controlling access to the resource that was sent to the client instance.*

(b) *m contains the mapping* $\mathtt{instanceID} : instanceID$ *— that is, m was a request beginning a software-only GNAP flow for the client instance identified by instanceID.*

Proof. Suppose we are given $\rho, Q, S, c, rs, r, CIgid, as, instanceID$ as specified in the lemma, and that $\mathsf{sw\_accessesResource}_\rho^Q(c, r, CIgid, rs, as, instanceID)$.

By definition, then, we know that there exist domains $domainAS \in \mathsf{dom}(as)$ and $domainRS \in \mathsf{dom}(rs)$ such that:

(1) $r = S(\mathit{rs}).\mathtt{clientResources}[\mathit{domainAS}][\mathit{instanceID}]$
(2) c stores the resource r under $S(c).\mathtt{grants}[\mathit{CIgid}][\mathtt{resources}][\mathit{domainRS}]$ in Line 78 of Algorithm A.7.

Since c stores r on Line 78 of Algorithm A.7, the conditional on Line 76 must have succeeded, so $\mathit{reference}[\mathtt{responseTo}] \equiv \mathtt{resourceResponse}$. We also know that *CIgid* here is equal to $\mathit{reference}[\mathtt{grantID}]$, from Line 2.

A reference with $\mathit{reference}[\mathtt{responseTo}] \equiv \mathtt{resourceResponse}$ is only created on Line 31 of Algorithm A.8, and this reference is used to send a message m either directly on Line 36 (for a bearer token), on Line 46 (for a key-bound token not using MTLS), or indirectly on Line 129 of Algorithm A.7 (for a key-bound token using MTLS, where the real request is only sent after MTLS concludes). In each case, the *CIgid* stored in *reference* is chosen on Line 21 of Algorithm A.8 to be some key in $s'.\mathtt{receivedValues}$. Moreover, on Line 30, an access token *tok* is loaded from $s'.\mathtt{receivedValues}[\mathit{CIgid}][\mathtt{accessToken}]$, and is included in m, under the `Authorization` header.

This field of the state of the client is only written to on Line 29 of Algorithm A.7, when the client receives an access token (*tok*) in a response from some p to a request for which $\mathit{reference}[\mathtt{responseTo}] \equiv \mathtt{grantResponse}$. Then, by Lemma E.9, we can conclude that there are an earlier step Q', a message $\hat{m}$, an AS as', and a nonce *ASgid* such that $\mathsf{client\_started}_{\rho}^{Q'}(c, \mathit{CIgid}, \hat{m}, as', \mathit{ASgid})$. If as' is honest, we can further conclude that $as = as' = p$ is the issuer of *tok* and that $S(as).\mathtt{tokenBindings}[\mathtt{grantID}] \equiv \mathit{ASgid}$.

It remains to show that $\hat{m}$ contains the mapping $\mathtt{instanceID} : \mathit{instanceID}$. We know that the resource request m contained the token *tok* issued by *as*. As such, upon receiving m, *rs* will send an introspection request to *as* (as an honest resource server always sends introspection requests to the issuer of the token it is provided with (Line 5 of Algorithm A.20)), and this request will contain the mapping $\mathtt{accessToken} : \mathit{tok}$.

Since the resource r accessed by c is $S(\mathit{rs}).\mathtt{clientResources}[\mathit{domainAS}][\mathit{instanceID}]$, we know that *rs* received a response *resp* to this introspection request such that $\mathit{resp}[\mathtt{active}] \equiv \top$ and $\mathit{resp}[\mathtt{access}][\mathtt{instanceID}] \equiv \mathit{instanceID}$, so that the checks on Lines 5, 29 of Algorithm A.21 can succeed.

We now examine what *as* as an honest AS must have done to create such a response upon receiving an introspection request containing *tok*. A check on Line 116 of Algorithm A.13 ensures that $\mathit{resp}[\mathtt{active}] \equiv \top$ only if $\mathit{tok} \in S(as).\mathtt{tokenBindings}$. The grant ID *ASgid* is then loaded from this token binding on Line 122, and on Line 141, $\mathit{resp}[\mathtt{access}][\mathtt{instanceID}]$ is set to $s'.\mathtt{grantRequests}[\mathit{ASgid}][\mathtt{instanceID}]$. This field of the AS state is only written to on Line 7 of Algorithm A.14, and only if the check on Line 2 succeeds, so $s'.\mathtt{grantRequests}[\mathit{ASgid}]$ is being created initially. The only place where Algorithm A.14 is called and this is true is on Line 4 of Algorithm A.13, immediately after *ASgid* is created. Since *ASgid* is created in response to $\hat{m}$, we see that $\hat{m}$ is passed in as the first argument to Algorithm A.14, and so the instance ID stored is $\hat{m}.\mathsf{body}[\mathtt{instanceID}]$. As such, $\hat{m}$ contains the mapping $\mathtt{instanceID} : \mathit{instanceID}$, as desired. □

Lemma E.18 (Session Integrity for End User Authorization). *Let $\mathcal{GWS}$ be a GNAP web system. We say that $\mathcal{GWS}$* is secure w.r.t. session integrity for authorization for end users *iff for every run ρ of $\mathcal{GWS}$, every processing step Q in ρ with*

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

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client instance $c \in \mathsf{CI}$ that is honest in S, every $rs \in \mathsf{RS}$ that is honest in S, every nonce r, and every nonce CIgid, we have that if $\mathsf{accessesResource}_{\rho}^{Q}(b, r, u, c, rs, \mathit{CIgid})$, *then*

(1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_{\rho}^{Q'}(b, c, \mathit{CIgid}, as)$, *and*
(2) Suppose that additionally, as and $\mathsf{governor}(u)$ *are honest in S and for all client instances c' that are honest in S and all key records $r \in s_0^c.\mathsf{keyRecords}[\mathit{dmnAS}]$ (for some domain $\mathit{dmnAS} \in \mathsf{dom}(as)$) with $r.\mathsf{method} \equiv \mathtt{mac}$, it holds that $\mathsf{dom}^{-1}(r.\mathsf{rs})$ is honest in S. Then there exists a grant ID ASgid and a processing step Q'' in ρ (before Q) such that* $\mathsf{finishLogin}_{\rho}^{Q''}(b, c, u, as, \mathit{CIgid}, \mathit{ASgid})$.

Proof. **For (1)**: From Item (3) of $\mathsf{accessResource}^{Q}(b, r, u, c, rs, \mathit{CIgid})$ we have that c sends a response to b in Line 18 of Algorithm A.11 that includes $r \equiv c.\mathtt{grants}[\mathit{CIgid}][\mathtt{resources}][\mathit{domainRS}]$. The receiver is loaded from $c.\mathtt{browserRequests}[\mathit{CIgid}][\mathtt{finishRequest}]$. This field is only written into once at Line 61 or Line 79 of Algorithm A.6. In both cases the `receiver` component is set to the sender of the current request m. From Lemma D.14 we have that c stores these values only when processing a request from the browser b' that sent the request to the `/startGrantRequest` path at c that led to the creation of *gid*. As c stored b as receiver, it must hold that $b' = b$. Requests to the `/startGrantRequest` path at c are sent only in Line 6 of *script_ci_index* (Algorithm A.12) loaded from a document of an origin of c. This shows $\mathsf{started}_{\rho}^{Q'}(b, c, \mathit{CIgid}, as)$ for some *as* and a previous processing step Q' as stated.

For (2) we first show that in a processing step Q''' before Q, c used some access token *AT* issued by $\mathsf{governor}(u)$ such that $\mathsf{successfulUseOfToken}^{Q'''}(c, \mathit{CIgid}, \mathit{AT}, rs, \mathsf{governor}(u))$. We continue to show $as = \mathsf{governor}(u)$ and that $\mathsf{ownerOfID}(u)$ authenticated c at *as* in a processing step Q'' before Q''', i.e., $\mathsf{finishLogin}_{\rho}^{Q''}(\mathsf{ownerOfID}(u), c, u, as, \mathit{CIgid}, \mathit{ASgid})$ with *ASgid* being the grant ID of the token *AT* at $\mathsf{governor}(u)$. Finally, we show $b = \mathsf{ownerOfID}(u)$. The last two together show the statement, i.e., $\mathsf{tryLogin}_{\rho}^{Q''}(\mathsf{ownerOfID}(u), c, u, as, \mathit{CIgid}, \mathit{ASgid})$.

$\underline{\mathsf{successfulUseOfToken}^{Q'''}(c, \mathit{CIgid}, \mathit{AT}, rs, \mathsf{governor}(u))}$: We first show that c sent an HTTPS request $m_{\text{resource}}^{\text{req}}$ to the `/resource` path of a domain of *rs* where the reference contains $\mathtt{grantID} : \mathit{CIgid}$.

From Item (2) of $\mathsf{accessesResource}_{\rho}^{Q}(b, r, u, c, rs, \mathit{CIgid})$, we have that c stores r under $S(c).\mathtt{grants}[\mathit{CIgid}][\mathtt{resources}][\mathit{domainRS}]$ where *domainRS* is a domain of *rs*. The `resource` field is written into only in Line 78 of A.7, when processing an HTTPS response $m_{\text{resource}}^{\text{resp}}$ from some process p. Hence, c must have sent a corresponding HTTPS request $m_{\text{resource}}^{\text{req}}$ to a domain of p.

In Line 78, the resource is stored at $\mathtt{grants}[\mathit{reference}[\mathtt{grantID}]][\mathtt{resources}][\mathit{reference}[\mathtt{domainRS}]]$. From Item (2) of $\mathsf{accessesResource}_{\rho}^{Q}(b, r, u, c, rs, \mathit{CIgid})$ we have $\mathit{reference}[\mathtt{grantID}] = \mathit{CIgid}$ and $\mathit{reference}[\mathtt{domainRS}] \in \mathsf{dom}(rs)$. We will now show that $\mathit{reference}[\mathtt{domainRS}]$ is also a domain of p, i.e., a domain of the process that c sent the request $m_{\text{resource}}^{\text{req}}$ to. Line 78 of A.7 is only executed if *reference* contains `responseTo : resourceResponse`. Requests with this reference are sent directly at Lines 36, 46, 59, 65, and using MTLS in Lines 52 and 71 of A.8. In all cases the request is (eventually) sent to the `/resource` path of the domain that is stored under *reference*[`domainRS`] in Line 31 or 63.

This shows that c sent an HTTPS request $m^{\text{req}}_{\text{resource}}$ to the `/resource` path of a domain of *rs* with the reference containing `grantID` : *CIgid* in a processing step before Q.

The resource stored by c under $S(c)$.`grants`[*CIgid*][`resources`][*domainRS*] in Line 78 of A.7 is taken from $m^{\text{resp}}_{\text{resource}}$.body. Hence, *rs* must have included the user resource r in Line 32 of A.21 and sent it in Line 45. By Items (1) to (3) from Lemma E.11 we thus get $\mathsf{successfulUseOfToken}^{Q'''}(c, \mathit{CIgid}, \mathit{AT}, \mathit{rs}, \mathsf{governor}(u))$ for some processing step Q''' before Q (recall that governor (u) is honest).

For the remaining proof let $\mathit{ASgid} = S(\mathsf{governor}(u))$.`tokenBindings`[*AT*][`grantID`]. As governor (u) is honest, we further get from Item (4) of Lemma E.11, that governor (u) stored u at `grantRequests`[*ASgid*][`subjectID`] in a processing step Q'' before Q'''.

$\underline{as = \mathsf{governor}\,(u)}$: We first show that c stores *AT* under $S(c)$.`receivedValues`[*CIgid*].
From Item 1 of $\mathsf{successfulUseOfToken}^{Q'''}(c, \mathit{CIgid}, \mathit{AT}, \mathit{rs}, \mathsf{governor}\,(u))$ we have that c sent a message to the `/resource` path of a domain of *rs* containing *AT* in the `Authorization` header and the reference contains `grantID` : *CIgid*. Since c is honest it only sends messages to the `/resource` path of a domain of *rs* in the last branch of Algorithm A.8. The token used in the `Authorization` header is taken from c.`receivedValues`[*grantID*][`accessToken`], where *grantID* is the same grant ID that is stored in the reference at `grantID`. Thus, c stores *AT* under c.`receivedValues`[*CIgid*][`accessToken`].

Hence, c must have received a response $m^{\text{resp}}_{\text{token}}$ from some process as^3 with $\mathit{AT} \equiv m^{\text{resp}}_{\text{token}}$.body[`accessToken`]. From our preconditions, we further know, that the issuer of *AT* (governor (u)) is honest and that the AS where c starts the flow for *CIgid* at (*as*) is honest. Thus, by Lemma E.9 with as^3, as^2 = governor (u), and $as^1 = as$, we get that $as = \mathsf{governor}\,(u)$.

$\underline{\mathsf{finishLogin}^{Q''}_{\rho}(\mathsf{ownerOfID}(u), c, u, as, \mathit{CIgid}, \mathit{ASgid})}$: From the application of Lemma E.9, we further have that c received *AT* from *as*. Further, from $\mathsf{accessesResource}^{Q}_{\rho}(b, r, u, c, rs, \mathit{CIgid})$ and the arguments in the first part, we have that *rs* sent a user resource to c. Honest RSs only send user resources if the response from the AS includes the `identity` field in the `access` field (see check in Line 29 of Algorithm A.21). From $\mathsf{successfulUseOfToken}^{Q'''}(c, \mathit{CIgid}, \mathit{AT}, \mathit{rs}, \mathit{as})$ we know that introspection was done at *as* and since *as* is honest it only includes the `access` field with an `identity` field in a response if $S(as)$.`tokenBindings`[*AT*][`for`] ≡ `endUser` due to the check in Line 138 in Algorithm A.13. With our definition of *ASgid*, we then get $\mathsf{finishLogin}^{Q''}_{\rho}(\mathsf{ownerOfID}(u), c, u, as, \mathit{CIgid}, \mathit{ASgid})$ by Lemma D.13.

$\underline{b = \mathsf{ownerOfID}\,(u)}$: As in the proof of Item (1), we get from $\mathsf{accessesResource}^{Q}_{\rho}(b, r, u, c, rs, \mathit{CIgid})$ that c stores b in $S(c)$.`browserRequests`[*CIgid*][`finishRequest`]. From Lemma D.14 we get that b must be the browser that started the flow *CIgid* at c. On the other hand, from $\mathsf{finishLogin}^{Q''}_{\rho}(\mathsf{ownerOfID}(u), c, u, as, \mathit{CIgid}, \mathit{ASgid})$ we know that the flow *CIgid* was started by ownerOfID (u). Thus b = ownerOfID (u) must be true. □

E.6 Theorem

Theorem E.19 (Security Authorization). *Every GNAP web system fulfills the authorization property and session integrity for both the software-only and the end user case.*

Proof. This directly follows from Lemmas E.15 and E.17, and Lemmas E.16 and E.18. □

F AUTHENTICATION

F.1 Definitions

Definition F.1 (Service Sessions). We say that there is a *service session identified by an ID ssid for an identity u at some client instance c under grant CIgid* in a configuration (S, E, N) of a run ρ of a GNAP web system, written as

$$\mathsf{serviceSession}^{S}_{\rho}(\mathit{ssid}, u, d, c, \mathit{CIgid}),$$

iff for $\mathit{sessionID} \equiv S(c)$.`grants`[*CIgid*][`sessionID`] we have

- $S(c)$.`sessions`[*ssionID*][`loggedInAs`] ≡ $\langle u, d\rangle$ and
- $S(c)$.`sessions`[*sessionID*][`serviceSessionID`] ≡ *ssid*

Definition F.2 (End user is logged in). For a run ρ of a GNAP web system $\mathcal{GWS}$ we say that a browser b was authenticated to a client instance c using an authorization server *as* and an identity u in a GNAP flow identified by a nonce *gid* in a processing step Q in ρ with

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

(for some S, S', E, E', N, N') and some event $\langle a, f, m\rangle \in E_{\text{out}}$ such that m is an HTTPS response sent by c to b, and we have that in the headers of m there is a header of the form $\langle$`Set-Cookie`, $[\langle$`__Host`, `serviceSessionID`$\rangle$:$\langle \mathit{ssid}, \top, \top, \top\rangle]\rangle$ for some nonce *ssid* such that for $\mathit{sessionID} \equiv S(c)$.`grants`[*gid*][`sessionID`] we have $S(c)$.`sessions`[*sessionID*][`serviceSessionID`] ≡ *ssid* and $S(c)$.`sessions`[*sessionID*][`loggedInAs`] ≡ $\langle u, d\rangle$ with $d \in \mathsf{dom}(as)$. We then write $\mathsf{loggedIn}^{Q}_{\rho}(b, c, u, as, gid)$.

F.2 Security Properties

Definition F.3 (Authentication Property). Let $\mathcal{GWS}$ be a GNAP web system. We say that $\mathcal{GWS}$ *is secure w.r.t. authentication* iff for every run ρ of $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every client instance c honest in S, every identity u with b = ownerOfID (u) honest, every domain d with $as = \mathsf{dom}^{-1}(d)$ honest, and every service session ID *ssid* with $\mathsf{serviceSession}^{S}_{\rho}(\mathit{ssid}, u, d, c, \mathit{CIgid})$ (for some *CIgid*), if for all client instances c' that are honest in S and all key records $r \in s^{c}_{0}$.keyRecords[d'] (for some $d' \in \mathsf{dom}(as)$) with r.`method` ≡ `mac`, it holds that $\mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S, then *ssid* is only derivable to c and b.

Definition F.4 (Session Integrity for Authentication). Let $\mathcal{GWS}$ be a GNAP web system. We say that $\mathcal{GWS}$ *is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{GWS}$, every processing step Q in ρ with

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

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client instance $c \in \mathsf{CI}$ that is honest in S, every nonce *CIgid*, and $\mathsf{loggedIn}^{Q}_{\rho}(b, c, u, as, CIgid)$ we have that

1. there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b, c, CIgid, as)$, and
2. Suppose that additionally, *as* is honest in S and for all client instances c' that are honest in S, all domains $dmnAS \in \mathsf{dom}(as)$, and all key records $r \in s^{c}_{0}.\mathsf{keyRecords}[dmnAS]$ with $r.\mathtt{method} \equiv \mathtt{mac}$, it holds that $\mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S. Then there exists a grant ID *ASgid* and a processing step Q'' in ρ (before Q) such that $\mathsf{finishLogin}^{Q''}_{\rho}(b, c, u, as, CIgid, ASgid)$.

F.3 Auxiliary Lemmas

Proposition F.5 (`sentTo` field contains receiver). *If a request m from an honest client instance contains* `sentTo` : *domainAS in the reference, then m is sent to domainAS.*

Just check all places where `sentTo` is set and compare to corresponding places where the request is sent.

F.4 Security w.r.t. Authentication

Lemma F.6 (Authentication Property). *Let $\mathcal{GWS}$ be a GNAP web system. We say that $\mathcal{GWS}$* is secure w.r.t. authentication *iff for every run ρ of $\mathcal{GWS}$, every configuration (S, E, N) in ρ, every client instance c honest in S, every identity u with $b = \mathsf{ownerOfID}(u)$ honest, every domain d with $as = \mathsf{dom}^{-1}(d)$ honest, and every service session ID ssid with $\mathsf{serviceSession}^{S}_{\rho}(ssid, u, d, c, CIgid)$ (for some CIgid), if for all client instances c' that are honest in S and all key records $r \in s^{c}_{0}.\mathsf{keyRecords}[d']$ (for some $d' \in \mathsf{dom}(as)$) with $r.\mathtt{method} \equiv \mathtt{mac}$, it holds that $\mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S, then ssid is only derivable to c and b.*

Proof. `serviceSessionIDs` are newly created by c in Algorithm A.11. If c does not send out *ssid* it is only derivable by c.

<u>c sends *ssid* only to b</u>: Now assume, that c sends *ssid* to some process p. Since c is honest this happens only in Line 18 of Algorithm A.11. We want to show that p must be $\mathsf{ownerOfID}(u)$. In Line 18, c sends *ssid* to the process stored in `browserRequests` $[CIgid]$`[finishRequest]`. By Lemma D.14 this is the browser that led to the creation of *CIgid* at c. We will now see that $\mathsf{ownerOfID}(u)$ started the flow *CIgid* at c, by showing $\mathsf{finishLogin}^{Q''}_{\rho}(\mathsf{ownerOfID}(u), c, u, as, CIgid, ASgid)$, for some *ASgid*.

In Line 18, *ssid* is sent to p only in the form of a `serviceSessionID` header created in Line 7. Hence, we have $\mathsf{loggedIn}^{Q}_{\rho}(p, c, u, as, CIgid)$ with *ssid* and d (since $d \in \mathsf{dom}(as)$). We first show that c must have received a message m from *as* containing the `subjectID` field with value u and c stores this subjectID under $c.$`receivedValues`$[CIgid]$ together with a domain d of *as*. Applying Lemma D.13 then yields the claim.

From $\mathsf{loggedIn}^{Q}_{\rho}(b, c, u, as, CIgid)$ we have that c stores $c.$`sessions` $[sessionID]$`[loggedInAs]` $= \langle u, d\rangle$ for some domain $d \in \mathsf{dom}(as)$ and $sessionID \equiv c.$`grants`$[CIgid]$`[sessionID]`. c only writes the `loggedInAs` field in Line 5 of Algorithm A.11. This happens only if a `subjectID` field is saved in $c.$`receivedValues`$[CIgid]$. A `subjectID` field is only written into $c.$`receivedValues`$[CIgid]$ in Line 18 of Algorithm A.7, when processing a response m. We show that m came from *as* and contains u in the `subjectID` field.

Since c receives m it must have previously sent a corresponding request m' including `responseTo` : `grantResponse` in the reference. The *subjectID* stored in Line 18 is taken from the `subjectID` field of m. Thus m must contain u in the `subjectID` field.

We are left to show that m was sent by *as*. From Proposition F.5 we know, that the request m' was sent to the domain saved in the `sentTo` field of the reference, and hence the response m was received from the same domain. Since c stores $\langle u, reference[\mathtt{sentTo}] \rangle = \langle u, d\rangle$ in the `subjectID` field and $d \in \mathsf{dom}(as)$, we get that m was sent by *as*. This gives all the preconditions of Lemma D.13 which shows $\mathsf{finishLogin}^{Q''}_{\rho}(\mathsf{ownerOfID}(u), c, u, as, CIgid, ASgid)$.

Thus, $\mathsf{ownerOfID}(u)$ started the flow *CIgid* and hence *ssid* is only sent to the honest $p = \mathsf{ownerOfID}(u)$.

<u>b sends *ssid* only to c</u>: In our browser model an honest browser sends cookies only back to the originating domain, thus b sends *ssid* only back to c. □

F.5 Session Integrity for Authentication

Lemma F.7 (Session Integrity for Authentication). *Let $\mathcal{GWS}$ be a GNAP web system. We say that $\mathcal{GWS}$* is secure w.r.t. session integrity for authentication *iff for every run ρ of $\mathcal{GWS}$, every processing step Q in ρ with*

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

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client instance $c \in \mathsf{CI}$ that is honest in S, every nonce CIgid, and $\mathsf{loggedIn}^{Q}_{\rho}(b, c, u, as, CIgid)$ we have that

1. *there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b, c, CIgid, as)$, and*
2. *Suppose that additionally, as is honest in S and for all client instances c' that are honest in S, all domains $dmnAS \in \mathsf{dom}(as)$, and all key records $r \in s^{c}_{0}.\mathsf{keyRecords}[dmnAS]$ with $r.\mathtt{method} \equiv \mathtt{mac}$, it holds that $\mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S. Then there exists a grant ID ASgid and a processing step Q'' in ρ (before Q) such that $\mathsf{finishLogin}^{Q''}_{\rho}(b, c, u, as, CIgid, ASgid)$.*

Proof. **For (1)**: From $\mathsf{loggedIn}^{Q}_{\rho}(b, c, u, as, CIgid)$ we have that c sends a response to b including a `serviceSessionID` cookie in the headers. Such responses are only sent in Line 7 of Algorithm A.11, where the receiver is loaded from $S(c).$`browserRequests`$[CIgid]$ `[finishRequest]`. The rest of the proof follows the one of Item (1) of Lemma E.18. I.e., the receiver stored there is the browser that started the request *CIgid*.

For (2): Similarly to the first part of the proof of Lemma F.6, we first show that $\mathsf{ownerOfID}(u)$ authenticated c to *as* using *CIgid* and *ASgid* and then we'll see that $b = \mathsf{ownerOfID}(u)$.

<u>$\mathsf{finishLogin}^{Q''}_{\rho}(\mathsf{ownerOfID}(u), c, u, as, CIgid, ASgid)$</u>: This follows from $\mathsf{loggedIn}^{Q}_{\rho}(b, c, u, as, CIgid)$ in the same way as in the second part of the first part of the proof of Lemma F.6.

<u>$b = \mathsf{ownerOfID}(u)$</u>: As in the proof of Item (1), we get from $\mathsf{loggedIn}^{Q}_{\rho}(b, c, u, as, CIgid)$ that c stores b in $c.$`browserRequests`

$[CIgid]$[`finishRequest`]. From Lemma D.14 we get that b must be the browser that started the flow *CIgid* at c.

From $\mathsf{finishLogin}_{\rho}^{Q''}(\mathsf{ownerOfID}(u), c, u, as, CIgid, ASgid)$ we have that the flow *CIgid* was started by $\mathsf{ownerOfID}(u)$. Thus b must be $\mathsf{ownerOfID}(u)$.

In total we showed $\mathsf{finishLogin}_{\rho}^{Q''}(b, c, u, as, CIgid, ASgid)$ concluding the proof. □

F.6 Theorem

Theorem F.8 (Security Authentication). *Every GNAP web system fulfills the authentication property and session integrity for authentication.*

Proof. This directly follows from Lemma F.6 and Lemma F.7. □