Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Security Analysis of the Grant Negotiation and Authorization Protocol

Florian Helmschmidt

Course of Study: Informatik

Examiner: Prof. Dr. Ralf Küsters

Supervisor: Pedram Hosseyni, M.Sc.

Commenced: September 23, 2021

Completed: March 23, 2022

Abstract

The Grant Negotiation and Authorization Protocol (GNAP) is a protocol under development by the IETF that allows delegating permissions to third parties. With these permissions, the third party can, for example, access protected APIs or obtain information directly from the issuer of the permissions. The scope of the permissions can be negotiated between the third party and the issuer.

Since this allows the third party to access the resources of the issuer, the security of the protocol is of key importance. For example, only the approved permissions should be delegated to only the authorized third party.

To analyze the security of GNAP, we model the protocol within the *Web Infrastructure Model*, including various interaction modes of GNAP. We define several security properties regarding the authorization of access via the protocol and prove them within our model.

In the course of this work, several attacks and vulnerabilities of GNAP were discovered, which we reported to the editors of GNAP. Together with the editors, we worked out mitigations and security considerations regarding these issues, which were added to the protocol. To be able to prove the security of GNAP, we also implemented them in our model.

Kurzfassung

Das Grant Negotiation and Authorization Protocol (GNAP) ist ein sich in Arbeit befindliches Protokoll der IETF, das es erlaubt, Berechtigungen an eine dritte Partei zu delegieren. Mit diesen Berechtigungen kann die dritte Partei etwa auf geschützte APIs zugreifen oder direkt Informationen vom Aussteller der Berechtigungen erhalten. Der Umfang der Berechtigungen kann dabei zwischen der dritten Partei und dem Aussteller ausgehandelt werden.

Da dies einem Dritten den Zugang zu den Ressourcen des Ausstellers ermöglicht, ist die Sicherheit des Protokolls von zentraler Bedeutung. So sollten etwa nur die genehmigten Berechtigungen an ausschließlich die autorisierte dritte Partei delegiert werden.

Um die Sicherheit von GNAP zu untersuchen, modellieren wir das Protokoll innerhalb des *Web Infrastructure Model* und inkludieren dabei verschiedene Interaktionsmodi von GNAP. Wir stellen mehrere Sicherheitseigenschaften bezüglich der Autorisierung von Zugriffen durch das Protokoll auf und beweisen diese innerhalb unseres Modells.

Im Rahmen der Arbeit wurden dabei verschiedene Angriffe und Schwachstellen von GNAP entdeckt, die wir den Editoren von GNAP gemeldet haben. Zusammen mit den Editoren haben wir Gegenmaßnahmen und Sicherheitserwägungen bezüglich dieser Probleme erarbeitet, die zum Protokoll hinzugefügt wurden. Um die Sicherheit von GNAP beweisen zu können, haben wir diese auch in unserem Modell umgesetzt.

Contents

1 Introduction **17**

2 The Grant Negotiation and Authorization Protocol **19**
2.1 Roles 19
2.2 Message Types 20
2.3 Interaction Modes 23
2.4 Example Flows 25

3 Attacks **31**
3.1 307 Redirect Attack 31
3.2 Cuckoo Token Attack 31
3.3 Client Instance Mix-Up Attack 33

4 Related Work **37**

5 Conclusion and Outlook **39**

Bibliography **41**

A Formal Model of GNAP **45**
A.1 Adjustments to the Web Infrastructure Model 45
A.2 Outline 50
A.3 Modeling Remarks and Limitations 51
A.4 Addresses and Domain Names 55
A.5 Keys and Secrets 55
A.6 Identities and Passwords 56
A.7 Corruption 56
A.8 Network Attackers 56
A.9 Browsers 57
A.10 Helper Functions 57
A.11 Client Instances 60
A.12 Authorization Servers 75
A.13 Resource Servers 87

B Definitions **95**

C Formal Security Properties **97**

D Proofs **99**
D.1 General Properties 99
D.2 Authorization Property for Software-only Authorization 110

D.3 Authorization Property for End Users 113
D.4 Authorization Property 123

List of Figures

2.1 GNAP flow using redirect interaction start mode and redirect interaction finish mode. 26
2.2 GNAP flow using user code interaction start mode and push interaction finish mode. 28

3.1 Cuckoo Token Attack. 32
3.2 Client Instance Mix-Up Attack. 34

List of Tables

A.1 List of scripts in $\mathcal{S}$ and their respective string representations. 51
A.2 List of placeholders used in the client instance algorithms. 62
A.3 List of placeholders used in the AS algorithms. 77
A.4 List of placeholders used in the RS algorithms. 90

List of Algorithms

A.1 Web Browser Model: Execute a script. 46
A.2 Web Browser Model: Process an HTTP response. 48
A.3 Web Browser Model: Main algorithm. 50
A.4 Helper Functions: Signing and sending requests. 58
A.5 Helper Functions: Validating key proofs. 59
A.6 Relation of a Client Instance R^c: Processing HTTPS requests. 65
A.7 Relation of a Client Instance R^c: Processing HTTPS responses. 67
A.8 Relation of a Client Instance R^c: Processing trigger messages. 70
A.9 Relation of a Client Instance R^c: Generating inquired values. 72
A.10 Relation of a Client Instance R^c: Sending a continuation request to finish interaction. 73
A.11 Relation of a Client Instance R^c: Returning resources and service session identifiers to browsers. 74
A.12 Relation of *script_ci_index*. 74
A.13 Relation of an AS R^{as}: Processing HTTPS requests. 79
A.14 Relation of an AS R^{as}: Check signature or MTLS nonce. 83
A.15 Relation of an AS R^{as}: Check login and perform interaction finish mode. 84
A.16 Relation of an AS R^{as}: Send grant response after interaction has finished. 85
A.17 Relation of an AS R^{as}: Creating a grant response. 86
A.18 Relation of an AS R^{as}: Leaking access tokens. 87
A.19 Relation of *script_as_login*. 87
A.20 Relation of an RS R^{rs}: Processing HTTPS requests. 91
A.21 Relation of an RS R^{rs}: Processing HTTPS responses. 92

Acronyms

API Application Programming Interface. 17

AS Authorization Server. 17, 19

CSRF Cross-Site Request Forgery. 37

DNS Domain Name System. 51

DY Dolev-Yao. 46

GNAP Grant Negotiation and Authorization Protocol. 3, 5, 17

IETF Internet Engineering Task Force. 17

JWS JSON Web Signatures. 20

MAC Message Authentication Code. 46

MTLS Mutual TLS. 51

RO Resource Owner. 17, 19

RS Resource Server. 17, 19

SAML Security Assertion Markup Language. 20

SIOP Self-Issued OpenID Provider. 33

SPA Single-Page Application. 20

TLS Transport Layer Security. 20

WIM Web Infrastructure Model. 18

1 Introduction

The Grant Negotiation and Authorization Protocol (GNAP) [26, 27] enables delegated authorization in HTTP-based systems. The delegation can be conveyed to a piece of software, which can thereby obtain user information as well as access to Application Programming Interfaces (APIs). The *negotiation* in the name of the protocol stems from the fact that during the protocol flow, the requested access rights and information can be repeatedly adjusted until an agreement is reached.

The basic idea behind GNAP works as follows: A user (called *end user* by GNAP) of a software (called *client instance*) wants to use the software to access protected resources located at a Resource Server (RS). To obtain this access, the client instance sends a request to an Authorization Server (AS) that manages access to these resources. The AS has the task of assessing the request and determining whether the client instance is granted access or not. In most cases, the AS cannot decide this on its own, but another entity, the Resource Owner (RO), handles this on behalf of the AS. The RO is the subject entity that is authorized to grant access to the requested resources at the RS. In practice, the RO and the end user are often the same natural person. To accept the request, the RO must first authenticate itself to the AS and can then authorize the request. Once the request has been authorized, the AS creates an access token and transmits it to the client instance. The client instance now includes the access token in requests to the RS, which grants access to the requested protected resources, given that the access token is valid. In addition to access tokens, the client instances can also request information about the RO from the AS (such as the RO's e-mail address or telephone number), which is returned directly from the AS to the client instance if the request is granted.

The main advantage of such a delegation process is that the RO does not have to entrust any login credentials to the client instance, but the client instance can still access the RO's protected resources using an access token. In this regard, GNAP fulfills many of the use cases of OAuth 2.0 [12] but is not compatible with it. Rather, GNAP attempts to be a replacement for OAuth 2.0 and many of its extensions while also enabling additional use cases by supporting various so-called *interaction modes* [26].

At the time of writing, GNAP is an Internet Engineering Task Force (IETF) Internet-Draft that is still in progress. The protocol is divided into two documents, with the core document (called *Grant Negotiation and Authorization Protocol* [26]) describing the interaction between client instances and ASs, while another document (called *Grant Negotiation and Authorization Protocol Resource Server Connections* [27]) focuses on the interaction between ASs and RSs. This thesis is based on version 08 of *Grant Negotiation and Authorization Protocol* and version 01 of *Grant Negotiation and Authorization Protocol Resource Server Connections*.

Contributions of this Thesis In this thesis, we present the first formal model of GNAP and perform its first formal security analysis. Our model is based on the Web Infrastructure Model (WIM), which has been used in the past to analyze similar protocols such as OAuth 2.0 [7] and the OpenID Financial-grade API [5], leading to the detection of various attacks.

As part of this work, it was found that two attacks on similar protocols, the *cuckoo token attack*, and the *307 redirect attack*, can also be applied to GNAP. In addition, we contributed to the security considerations regarding another attack called the *client instance mix-up attack* in this work. Details can be found in Chapter 3.

We have also informed the editors of GNAP about a security issue with the derivation of so-called *downstream tokens* by an RS at an AS. Security considerations to fix this issue were still being discussed under this GitHub issue [25] at the time of completion of this work.

Several GitHub pull requests and issues were also created for both the core document and the *GNAP Resource Server Connections* document as part of this work. These are mostly about fixing inconsistencies and ambiguities in the documents. All pull requests and issues created by the author of this thesis regarding the core document can be found under [15]. Pull requests and issues concerning *GNAP Resource Server Connections* can be found under [16].

Structure of this Thesis Chapter 2 provides an overview of GNAP and illustrates its workings with two sample flows. Chapter 3 discusses the attacks on GNAP that were found during this work and how they were addressed. Related work can be found in Chapter 4. We conclude in Chapter 5. The appendix consists of the complete formal model of GNAP in Appendix A, the definitions used in the analysis in Appendix B, the definitions of the proven security properties in Appendix C, and their proofs in Appendix D.

2 The Grant Negotiation and Authorization Protocol

In this chapter, we first consider the different roles of the parties involved in a GNAP flow (Section 2.1). This is followed in Section 2.2 by a description of the different types of messages that are exchanged between the individual participants. Section 2.3 introduces the different interaction modes. These are used when the RO needs to authorize a request. The various interaction modes enable different usage scenarios of GNAP, such as when the client instance cannot perform redirects to arbitrary URIs or when the RO cannot be redirected back to the client instance after its interaction with the AS. Section 2.4 then explains the overall flow of GNAP using two examples in which different interaction modes are used.

2.1 Roles

The parties participating in a flow assume different roles in GNAP, which are presented below based on [26]. A human or an implementation can take on multiple roles. For example, the end user can also be the RO, while RS and AS can be the same piece of software.

Authorization Server (AS) Server that can grant a client instance access to protected resources at an RS by issuing an access token. Can also transmit information about an RO to a client instance. Interacts with ROs, if necessary, to decide whether to grant a request.

Client Instance Application used by an end user to access resources at one or more RSs that requires permission from one or more ASs to do so. Client instances are identified by the ASs and the RSs through their unique keys. GNAP differentiates between the terms *client instance* and *client software*. A client instance is a concrete instance of a client software. Thus, multiple client instances can exist for one client software.

Resource Server (RS) Server where protected resources can be accessed, requiring a valid access token issued by an AS. Depending on the structure of the access token, the RS may need to communicate with the AS that issued the access token to verify its validity. This is called *token introspection*.

Resource Owner (RO) Subject entity that interacts with the AS to grant or deny access to resources that it has authority upon. This does not always have to be a natural person that has to grant a request manually. Depending on the implementation, an AS can also grant a request without human interaction based on organizational rules, for example, if only access to certain resources that are not strongly protected is requested and the client instance is already known to the AS.

End user Natural person operating a client instance. This person can be the RO but does not have to be.

2.2 Message Types

This section describes the different message types used in the interaction between ASs and client instances. In general, all messages in GNAP must be secured using Transport Layer Security (TLS). The client instance must also sign all its requests, whether they are sent to the AS or the RS. The key for checking the signature can either already be known to the AS through a previous registration of the client instance with the AS, or the AS also allows requests from client instances unknown to it and takes the public key from the first request of the client instance. The latter is primarily intended for scenarios in which the client instance only exists temporarily, for example in the case of a Single-Page Application (SPA). Client instance registrations are out of scope for GNAP.

2.2.1 Grant Request

Each GNAP flow starts with a client instance sending a grant request to an AS. This can either be triggered by the end user of the client instance or the client instance does this on its own behalf without any user being involved. The AS to be used by the client instance is either preconfigured or the client instance must determine this based on the requested resources and/or information. For example, the AS to be used can be discovered via the RS on which resources are to be accessed.

A grant request may contain the following fields in its body [cf. 26]:

access_token Optional. If the client instance wants to receive an access token, it must specify in this field the rights and properties to be associated with the access token. The client instance can also request multiple access tokens at once, all of which are specified within this field. For each access token, the client instance can optionally include a `bearer` flag. If the request is granted, the AS issues a bearer access token when the `bearer` flag is used. When using a normal access token with an RS, the request must be signed with a key to which the access token is bound. This is normally the key used by the client instance to sign the grant request. When using a bearer access token, the request to the RS does not have to be signed. The inclusion of the bearer access token alone is sufficient to access the resources.

subject Optional. If the client instance wants to request information about the RO from the AS, the requested information must be specified in this field. Both subject identifier subject types as specified in the *Subject Identifiers for Security Event Tokens* draft [3] and assertion formats can be requested. Possible assertion formats are OpenID Connect ID Tokens [28] and Security Assertion Markup Language (SAML) 2 assertions.

client Required. This field is used by the client instance to identify itself to the AS. The client instance must specify its public key or a key reference. A key reference is a reference to a key that is already known to the AS, for example through a previous registration of the client instance with the AS. When specifying a key, the key proofing method used by the client instance to prove possession of that key must always be specified as well. Currently, GNAP supports the use of HTTP Message Signatures [2], OAuth-mTLS [4], and JSON Web

Signatures (JWS) [19] for this purpose. If a key reference is used, it can also refer to a symmetric key, so that message authentication codes are used as key proofs. Optionally, the client instance can also specify additional information, such as a string identifying the client software used by the client instance. This information can be used by the AS when interacting with the RO to inform the RO about the requesting client instance. Alternatively, if the client instance is already known to the AS, the client instance can specify only an instance identifier (see Section 2.2.2).

user Optional. If the client instance already knows identifiers or assertions of its end user (for example from a previous run), it can specify them in this field. The subject identifiers specified in [3] can be used as identifiers. These identifiers can be used by the AS to identify the RO to be contacted. For example, when interacting with the RO, the AS can directly start a login process for this user, so that the RO does not have to enter its username. The AS must not assume, based on a given subject identifier, that a particular RO is present at the client instance. OpenID Connect ID tokens [28] or SAML 2 assertions can be used as assertions. Valid assertions may be used by the AS to skip the interaction with the RO.

interact Optional. This field specifies the interaction modes that the client instance supports. Interaction modes are used when the RO must interact with the AS to grant a request. GNAP distinguishes between interaction start modes and interaction finish modes. Interaction start modes specify how the client instance can start the interaction, while interaction finish modes specify how the client instance can determine that the interaction with the AS is complete. See Section 2.3 for a description of the different interaction modes. Omitting this field signals to the AS that the client instance does not support any interaction modes for this request, which is the case, for example, when software-only authorization is used.

2.2.2 Grant Response

A grant response is sent from an AS to a client instance in response to a grant request or a continuation request (see Section 2.2.3). It can contain the following fields [cf. 26]:

continue Optional. If the AS allows the client to continue the request through a continuation request, the information required by the client instance for this is specified in this field. The AS must specify a URI to which the client instance must send the continuation request. This can be a stable URI or a different one for each request. The AS must also include an access token that is bound to the key used by the client instance in the grant request. This access token must be presented by the client instance in the continuation request and must not be usable with resources outside the AS. It is recommended to specify a time in seconds that indicates how long the client instance should wait before calling the specified URI.

access_token Optional. This field returns requested access tokens granted to the client instance. In addition to the value of the access token, a management URI can optionally be specified for each access token, which the client instance can use to rotate or revoke the token. If an access token is a bearer access token, the AS must set a `bearer` flag for this access token. The AS may also set a `durable` flag indicating that the access token is still valid after a rotation or after a modification of the underlying request through a continuation request. If an access token is not a bearer access token, the AS can also bind the access token to a different key than the one used by the client in its grant request. In this case, the AS must specify the

key to which the access token is bound. The client instance must be able to dereference or process this key to sign the associated request when using the access token. For example, the AS could generate a new key pair for use with this access token and then include the public and the private key in the grant response. If the client instance has requested multiple access tokens, the AS can issue any subset of them and reject all others.

interact Optional. If the client instance has specified supported interaction modes in its grant request, the AS hereby responds to all interaction modes that it also supports. See Section 2.3 for the exact flow of the different interaction modes.

subject Optional. If the client instance requested subject information about the RO from the AS and the request was granted, the AS returns this information in this field. The AS must only return this information if it is certain that the RO and the end user are the same party, for example by interacting with the RO.

instance_id Optional. An instance identifier is an unguessable string that a client instance can specify instead of the concrete information in the client field of a grant request. This prevents the client instance from having to transmit its full public key and other information in each grant request. The instance_id field allows the AS to assign such an instance identifier to the client instance, which can then use it in future grant requests. Alternatively, an instance identifier can be assigned during pre-registration, as with key references. An instance identifier must be protected as a secret by the client instance.

error Optional. Contains an error message to the client instance. This may indicate, for example, that the RO has rejected the request, or (in the case of a continuation request) the referenced request is not known to the AS.

2.2.3 Continuation Request

A continuation request can be sent from a client instance to an AS if the continue field was used in the last grant response that the client instance received from the AS. If the previous request from the client instance (either a grant request or a continuation request) required interaction with the RO, the AS can only respond to a (further) continuation request once this interaction has been finished. The URI and the access token to be used for the continuation request can be obtained by the client instance from the continue field of the previous grant response. An AS responds to a continuation request with a new grant response.

Continuation requests can be used in the following scenarios:

- If the previous request of the client instance required interaction with the RO and an interaction finish mode was used, the client instance receives a so-called interaction reference through the interaction finish mode (see Section 2.3.2). By using the finish mode, the client instance is signaled when the interaction with the RO is complete. The client instance then sends a continuation request including the interaction reference to the AS to get the result of the interaction. The subsequent grant response may then contain, for example, the requested access token or an error message stating that the RO has rejected the request. Such a continuation request is a POST request.

- If the previous request from the client instance requires interaction with the RO, but no interaction finish mode is used (perhaps because the client instance does not support one), the client instance sends a continuation request with an empty body to the AS to inquire whether the interaction has already finished. The time interval for this can be specified by the AS in the grant response. If the interaction is not yet completed, the AS responds with an error message and the client instance must try again periodically until the interaction is finished. Once the interaction is finished, the grant response of the AS is again dependent on the decision of the RO. Such a continuation request is also a POST request.
- A continuation request can also be used by the client instance to adjust the previous request. This can happen, for example, if the previous request was rejected and the client instance wants to restrict the requested rights and/or information. It is also possible that the client instance has already received an access token in the course of this flow, but now wants to extend the rights associated with it. If this extension is granted, the AS creates a new access token and may revoke the old one unless the durable flag has been set for it. Extending the requested rights and/or information may require a (further) interaction with the RO. To adjust the previous request the client instance sends a PATCH request. In the body of the request, the client instance can specify all fields of a grant request except the client field, which cannot be altered. If a field is specified, its value replaces the previous value. The values of all unspecified fields remain the same. The client instance may also need to specify an interaction reference if the last request involved an interaction with the RO using an interaction finish mode.
- A client instance can cancel an ongoing grant request by sending a DELETE request to the continuation URI. The AS should revoke all associated access tokens in this case.

2.3 Interaction Modes

The interaction modes are used when an interaction with the RO is required to authorize a grant request. There are interaction start modes and interaction finish modes. Interaction start modes specify how the interaction with the RO is initiated, while interaction finish modes describe callback mechanisms to inform the client instance about the completion of the interaction. A client instance specifies the interaction start modes and interaction finish modes it supports in the grant request. When interaction is required, the AS selects from the interaction modes offered by the client instance those that it also supports and includes them along with the required information in the grant response.

2.3.1 Interaction Start Modes

A client instance can indicate support for the following interaction start modes in a grant request:

redirect The redirect mode can be used if the client instance can point the end user to any URL. The AS sends a URL in its grant response to the client instance, where the end user can then interact with the AS via their browser. GNAP does not specify how exactly this URL should

be accessed. Possible options include a browser redirect, starting the user's browser with this URL, or displaying the URL as a QR code that can be scanned by the user. The AS must be able to uniquely associate the URL with the grant request.

user_code The user code mode is intended for scenarios where the client instance cannot easily communicate an arbitrary URL to the end user. In this mode, the AS transmits a short, human-readable code associated with the grant request to the client instance. The client instance in turn communicates this code to the end user. Using a secondary device, the end user then navigates to a static URL where they can interact with the AS. There the end user enters the user code so that the AS can associate the interaction with the grant request. Since the URL is static, it can be recorded in the documentation of the AS and/or preconfigured in the client instance so that it can display the URL to the user in addition to the user code. Nevertheless, it is recommended that the AS also includes the URL in the grant response in addition to the user code.

app The app mode works similarly to the redirect mode. Here, too, the AS sends a URL to the client instance, which it can uniquely associate with the grant request. However, this URL is then used to start an application on the end user's system, which the end user can then use to interact with the AS. How this application is started and how it interacts with the AS is out of scope for GNAP.

If interaction with the RO is required, but there is no interaction start mode supported by both client instance and AS the AS may have the ability to contact the RO asynchronously and ask for authorization of the request. Possible procedures for this are out of scope for GNAP. If the AS cannot contact the RO asynchronously either, the AS must reject the request.

2.3.2 Interaction Finish Modes

Through the interaction finish modes, the client instance specifies how it can be notified by the AS of the completion of the interaction with the RO. GNAP defines two different callback mechanisms for this, one of which the client instance can specify in its grant request:

redirect In this mode, the end user is usually also the RO. The client instance specifies a URI in its grant request to which the RO should be redirected after the interaction is complete. Furthermore, the client instance specifies a nonce (*number once*) in the grant request. If an interaction is required and the AS supports this interaction finish mode, the AS specifies its own nonce within the interact field in its grant response. The AS also generates an unguessable interaction reference that is one-time-use. Then the AS calculates a hash value over the client instance's nonce from the grant request, its own nonce from the grant response, the interaction reference, and the grant endpoint URL that the client instance used for its initial request. Once the interaction between the RO and the AS is complete, the AS redirects the RO to the client instance using the URI specified by the client instance in the grant request. The AS adds the interaction reference and the hash value as query parameters. How exactly the redirect is done is up to the implementation. Besides a browser redirect, it is also possible to start the user's browser with this URI. The client instance validates the hash value and, if the validation is successful, sends a continuation request to the AS, including the interaction reference.

push Using the push interaction finish mode, the client instance also specifies a URI and a nonce in the grant request, and the AS responds with its own nonce if it supports this mode and allows it for this request. Here, however, the AS sends a POST request to the URI specified by the client instance after the interaction is completed. Interaction reference and hash value are generated by the AS as in redirect interaction finish mode and transmitted to the client instance in the body of the POST request. If the hash value is validated successfully, the client also sends a continuation request to the AS, including the interaction reference.

If the client instance does not specify an interaction finish mode or if the specified finish mode is not supported by the AS, the client instance must actively poll the status of the interaction using continuation requests. Since in this case no hash value is generated by the AS that can be checked by the client instance, this is less secure than using an interaction finish mode. The protocol points this out and recommends using an interaction finish mode whenever possible [26].

2.4 Example Flows

This section illustrates how GNAP works using two examples. The first example uses the redirect interaction start mode and the redirect interaction finish mode. The second example shows the user code interaction start mode and the push interaction finish mode. In both examples, all requests sent from the client instance to the AS are secured by a key proofing method. For simplicity, the values contained in the requests for this purpose are not shown in the examples.

2.4.1 Interaction using Redirects

In this example, the redirect interaction start mode and the redirect interaction finish mode are used. The end user is also the RO. This scenario is similar to the OAuth 2.0 Authorization Code grant type. The flow is illustrated in Figure 2.1. The data depicted in the second lines of the arrow labels is either transferred in URI parameters, HTTP headers, or POST bodies.

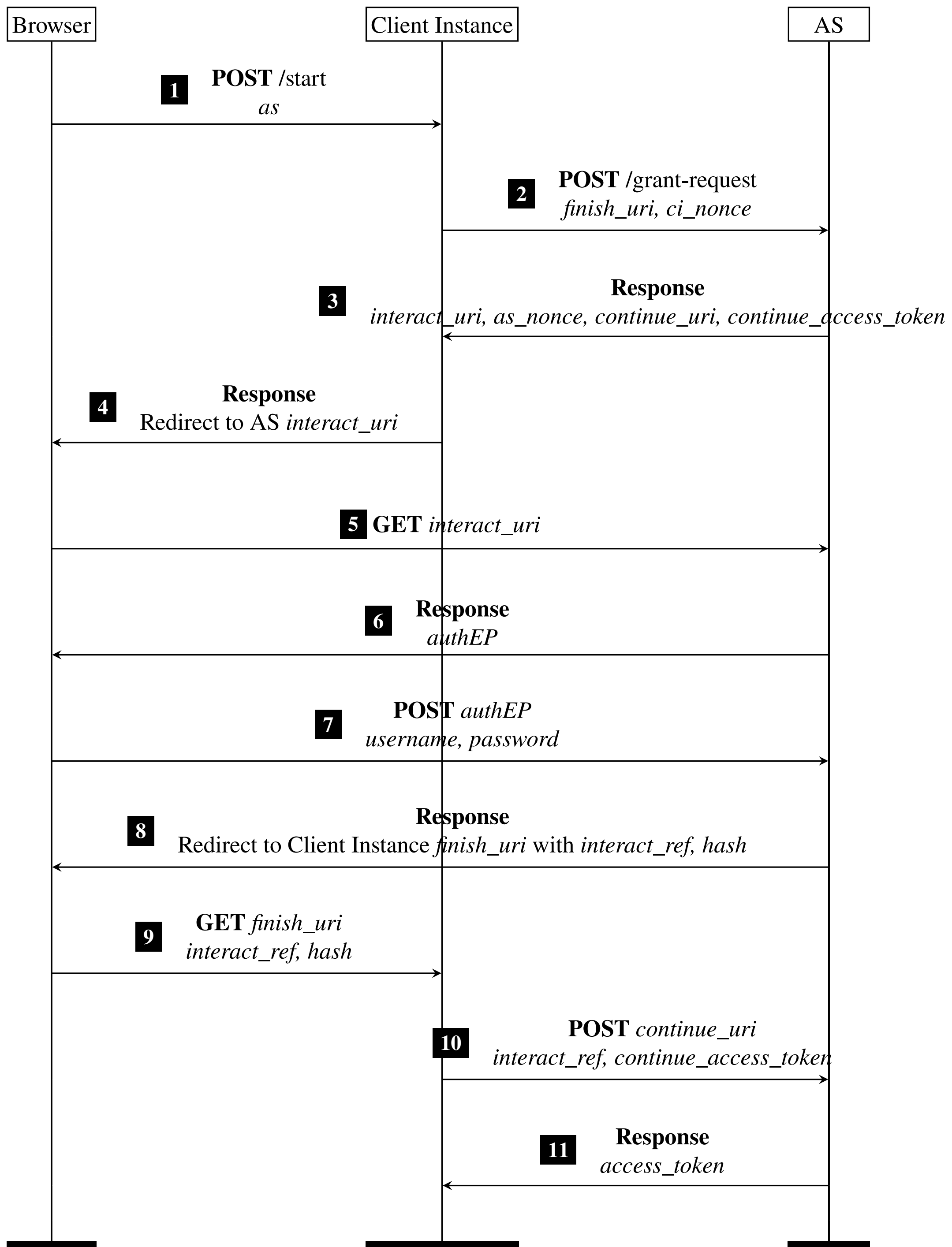

Figure 2.1: GNAP flow using redirect interaction start mode and redirect interaction finish mode.

This flow is triggered by the end user, who e.g. clicks on a button on a web page of the client instance to select a specific AS. In our modeling, a POST request [1] is sent to the client instance for this purpose, in which the selected AS is specified.

The client instance then sends a grant request 2 to the selected AS (if the client instance is configured to use this AS), in which the client instance specifies the URI to which the AS should redirect the end user after the interaction is complete (*finish_uri*). In addition, the AS includes a freshly generated nonce that will be needed later to calculate a hash value (*ci_nonce*).

In the grant response 3, the AS transmits the *interact_uri* to which the client instance should redirect the end user for interaction with the AS. In addition, the AS sends its own nonce (the *as_nonce*), which is also used in the calculation of the hash value, and the *continue_uri*, to which the client instance must later send the continuation request. In the continuation request, the client instance must specify an access token, which the AS sends to the client instance in 3 (*continue_access_token*).

The client instance then redirects the end user/RO to the *interact_uri* of the AS 4 (redirect interaction start mode). There, the RO must authenticate itself to the AS and authorize the grant request of the client instance. In our model, the GET request resulting from the redirect 5 is answered with a script 6 that transfers the username and password of the RO to a login endpoint *authEP* of the AS 7.

After successful login, the RO is redirected to the *finish_uri* of the client instance in 8 and 9 (redirect interaction finish mode). Thereby, the AS inserts two parameters into the *finish_uri*: the *interact_ref* and the previously mentioned hash value *hash*. The *interact_ref* is a value generated by the AS that it can use to map the client instance's subsequent continuation request to the current interaction. The hash value covers the nonce of the client instance from 2 (*ci_nonce*), the nonce of the AS from 3 (*as_nonce*), the *interact_ref*, and the endpoint that the client instance used for the grant request 2. The client instance also calculates this hash value and checks if it matches the hash value from 9. If the values match, the continuation request to complete the flow is sent to the *continue_uri* from 3. This continuation request contains the access token from 3 and the *interact_ref* from 9. The AS checks these values and, if successful, sends one or more access tokens to the client instance, which the client instance can now use to access resources of the RO at the corresponding RSs.

2.4.2 Secondary Device Interaction

In this example, the user code interaction start mode and the push interaction finish mode are used. The flow is illustrated in Figure 2.2. The data depicted in the second lines of the arrow labels is either transferred in URI parameters, HTTP headers, or POST bodies.

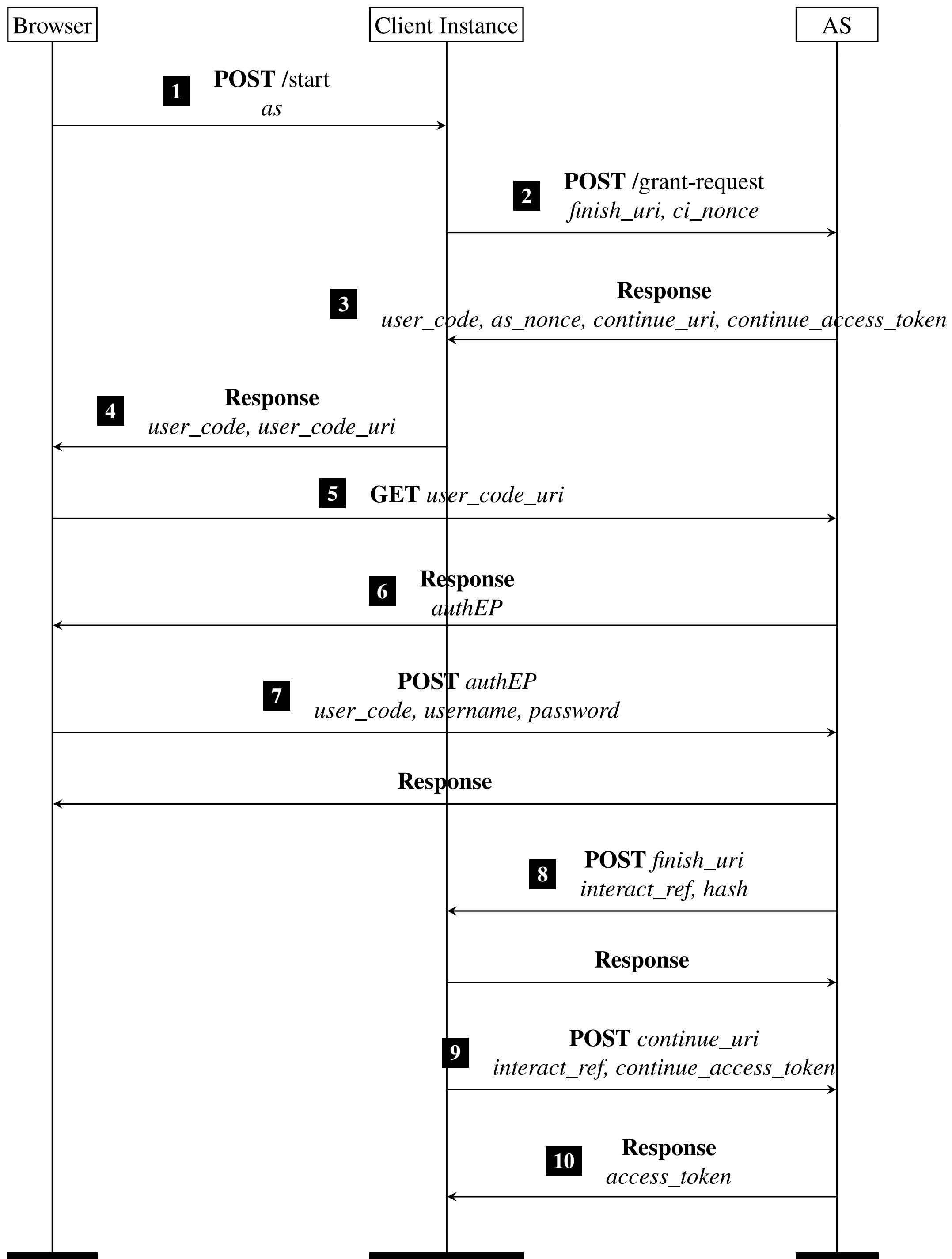

Figure 2.2: GNAP flow using user code interaction start mode and push interaction finish mode.

In our modeling of the user code interaction start mode, steps [1] and [2] proceed as in Section 2.4.1. In the grant response [3], however, the AS does not specify a URI where the RO can log in, but the *user_code* that the RO must enter on a dedicated user code interaction page of the AS. Since this page must be static, the AS does not need to transmit its URI to the client instance. Instead, client instances may have this static URI stored for each AS used and display it to the end user, or the end user may learn the URI elsewhere, such as from the client instance's documentation.

In [4], the client instance now transmits the *user_code* and the user code interaction URI (*user_code_uri*) to the browser. The RO must now navigate to the *user_code_uri* and enter the *user_code* there. In our modeling, this is done by opening the *user_code_uri* in a new window [5]. In [6], a script is then returned that receives the *user_code* as input. This script then simulates the input of the *user_code* and the login credentials by transmitting them to the AS in a POST request [7].

The AS can associate the login with the pending grant request via the user code. If the login is successful, it sends a new interaction reference and the hash value known from Section 2.4.1 via a POST request to the client instance [8] (push interaction finish mode). The client instance checks the hash value and can receive the requested access token(s) [10] via a continuation request [9] as in Section 2.4.1.

3 Attacks

In Section 3.1 and Section 3.2 of this chapter, we explain two attacks on GNAP that we found during the work on this thesis and what was done to mitigate them. Both attacks were reported to the editors of GNAP via email on October 26, 2021, and were confirmed by them to be valid. The attacks were subsequently discussed by the GNAP working group at the IETF 112 meeting on November 11, 2021 [22]. While working on this thesis, another attack was discovered, which we describe in Section 3.3 and for which we have also incorporated mitigations into our model.

3.1 307 Redirect Attack

This attack on OAuth 2.0 was found by Fett et al. [7]. The following assumptions are required for the attack to apply to GNAP:

- An honest RO logs in to an honest AS when interacting with it, and in doing so, the RO's credentials are transmitted to the AS in a POST request.
- After logging in by submitting this POST request, the AS redirects the RO directly back to the client instance due to the use of the redirect interaction finish mode.
- For this redirect the AS uses the HTTP 307 redirect status code.

The problem now is that when using the 307 redirect status code, the RO's user-agent retains the HTTP method used as well as the body of the request. Thus the same POST request including the body is sent to the client instance through the redirect. Since the POST request contains the RO's credentials in the body, this leaks the credentials to the client instance. If the client instance is controlled by an attacker, the attacker can now abuse the credentials, e.g., to impersonate the honest RO at the honest AS.

To prevent this attack, the author of this thesis has proposed a security consideration [13] based on the OAuth 2.0 Security Best Current Practice [21]. This security consideration was merged into the protocol on December 2, 2021, and states that ASs should use the HTTP 303 status code for redirects of requests that potentially contain sensitive data, as only this redirect status code unambiguously requires that the request is rewritten into a GET request and the body is thus dropped.

3.2 Cuckoo Token Attack

This attack on the OpenID Financial-grade API was found by Fett et al. [5]. Under the following assumptions, it can also be applied to GNAP:

- An honest client instance is configured to use multiple ASs, one of which is controlled by the attacker.
- The honest client instance uses the same key and key proofing method for the AS controlled by the attacker as well as at least one of the honest ASs the client instance is configured to use.
- The attacker obtained a leaked access token issued by one of the ASs for which the honest client instance uses the same key and key proofing method as for the AS controlled by the attacker. This access token can be used to access the resources of an honest RO at an honest RS, and it is bound to the key and key proofing method of the honest client instance.

If these assumptions are given, the attacker can now use the attack to bypass the token binding of the access token he obtained, as shown in Figure 3.1.

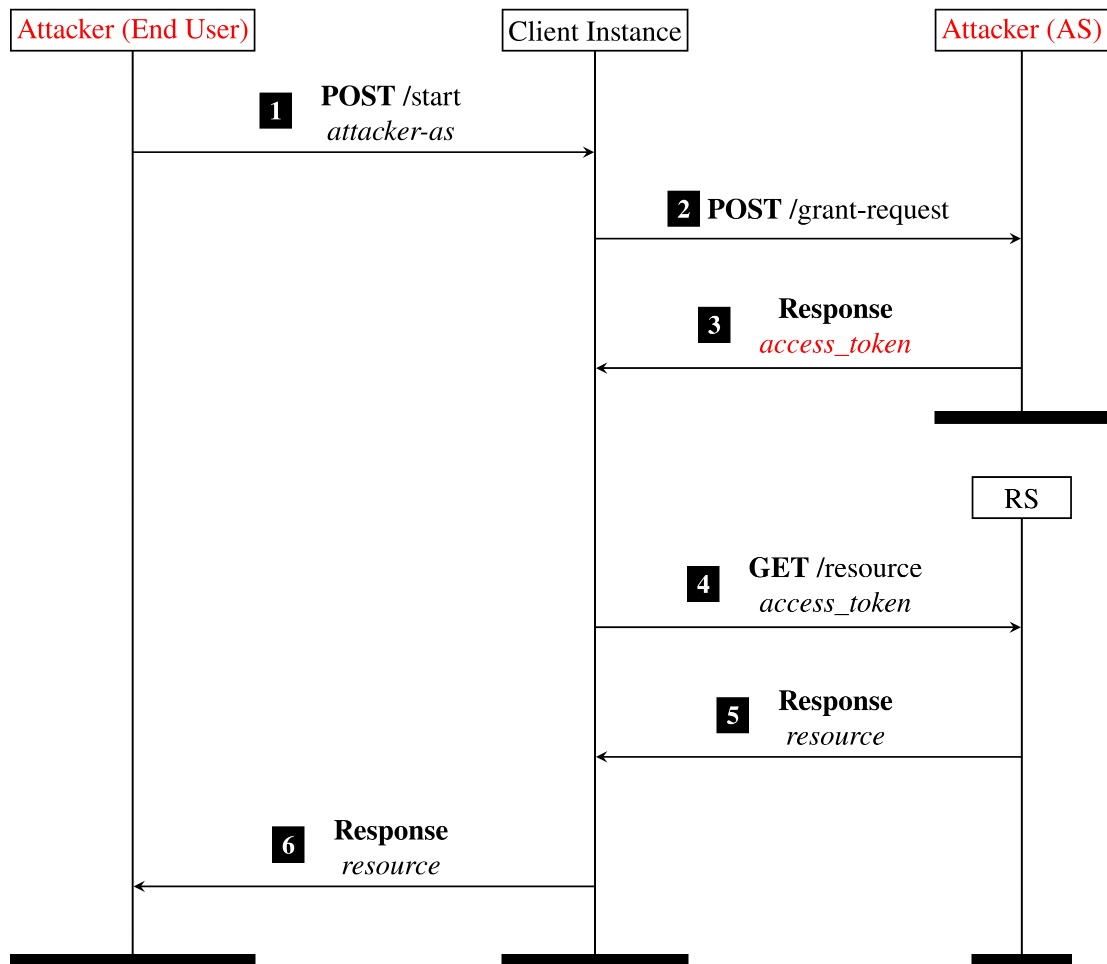

Figure 3.1: Cuckoo Token Attack.

In the role of an end user, the attacker starts a flow with the honest client instance for which the access token obtained by the attacker was issued [1]. Thereby, the attacker lets the client instance use the AS it controls. The client instance then sends a grant request to the attacker-controlled AS

requesting an access token [2]. The attacker responds to this grant request with a grant response that contains the leaked access token obtained by the attacker [3]. The attacker also binds the returned access token to the key and key proofing method used by the client instance (instead of binding the access token to a newly generated key). The client instance now thinks that this access token is a new access token issued to it since access tokens are opaque to client instances in GNAP. If the client instance now requests resources from an honest RS [4], the RS will return the resources of the honest RO for whom the access token was actually issued [5]. The access token can be used at the RS both in the session of the client instance with the honest RO and in the session with the attacker since the access token was bound to the key and the key proofing method of the client instance in both sessions and the client instance uses the same key and the same key proofing method with both ASs. Thus, the attacker now has access to the honest RO's resources via the honest client instance [6].

To prevent this attack, the editors of GNAP added a new security consideration to the protocol that proposed two different approaches to mitigate the attack [24]. This security consideration was merged into the protocol on December 23, 2021. The first mitigation is that the client instance uses a different key for each AS it is configured to use. This invalidates the second assumption. The RS will reject the request for the resource [4] because the client instance will use the key it uses for the AS controlled by the attacker, but the access token is bound to the key for the honest AS. The second mitigation is that the client instance maintains a strong association between the RS and a particular AS that is allowed to issue access tokens for that RS. Thus, the attacker cannot access the honest RO's resources because there is no RS that the client instance accesses with both access tokens it received from the honest AS and access tokens it received from the attacker-controlled AS.

3.3 Client Instance Mix-Up Attack

In December 2021, it was discovered that a known attack on the cross-device flow of the Self-Issued OpenID Provider (SIOP) protocol can also be applied to GNAP under certain conditions. The attack on SIOP was presented and discussed at the OAuth Security Workshop 2021 [20].

In the following, we explain how the attack on GNAP works when the user code interaction start mode and the push interaction finish mode are used. More details and an explanation of why the attack does not work when the redirect interaction start mode and the redirect interaction finish mode are used can be found in [14].

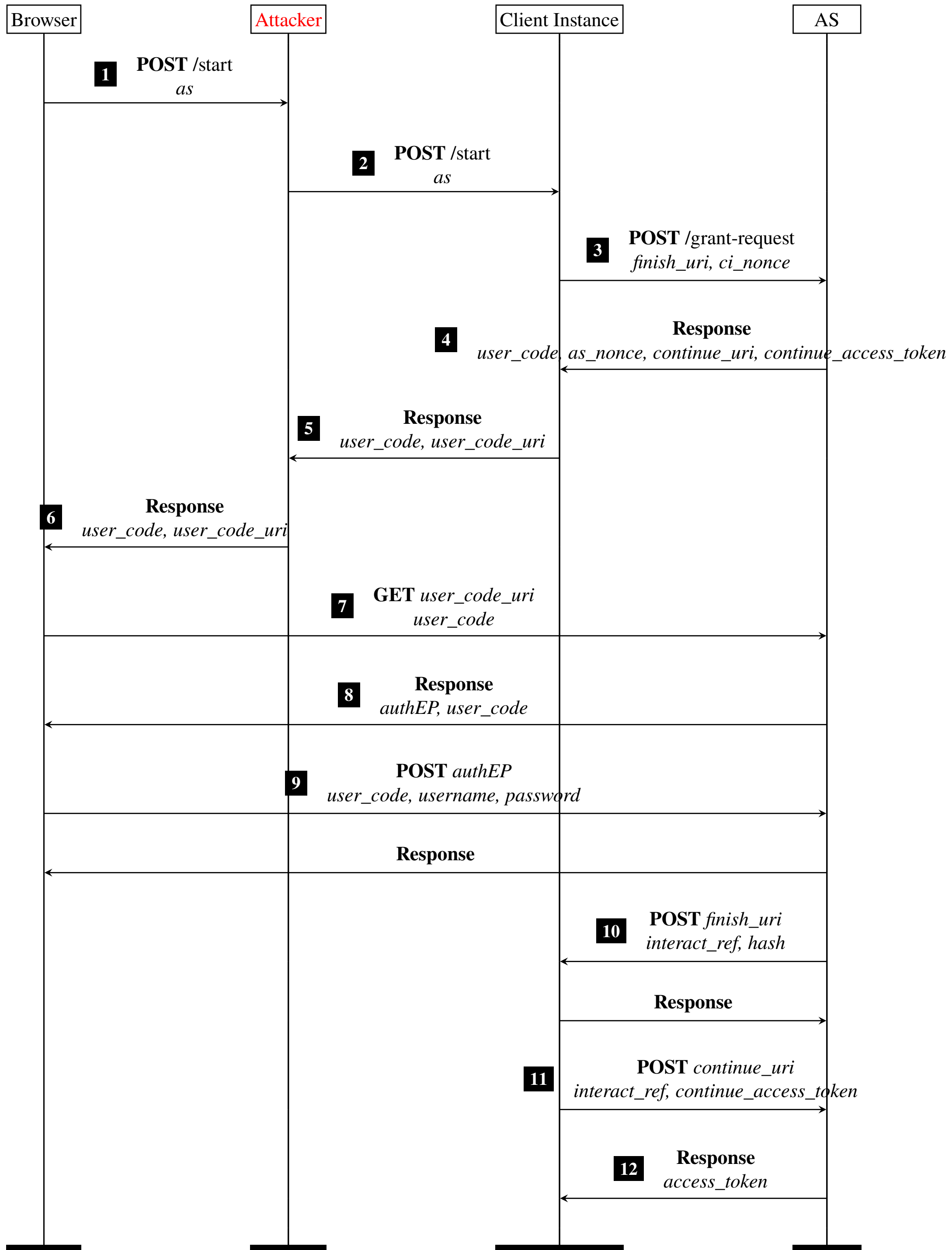

Figure 3.2: Client Instance Mix-Up Attack.

The attack is illustrated in Figure 3.2. In this attack, the attacker takes on two different roles. In the communication with the end user/browser, the attacker assumes the role of a client instance while acting as an end user/browser towards an honest client instance. The attack begins with an honest browser starting a GNAP flow with the attacker as the client instance [1]. The attacker replays this request (in the role of a browser) by also starting a GNAP flow with an honest client instance, specifying the same AS as the honest browser [2]. The client instance then sends a grant request to the chosen honest AS [3] and receives a user code [4] due to the use of the user code interaction start mode, which it then transmits to the attacker [5]. The attacker passes this user code back to the honest browser [6]. The honest browser then navigates to the user code interaction URI of the AS [7] and transmits the user code to the AS [9]. Since the honest browser interacts with the AS selected by itself in [1], the browser cannot detect the attack at this point. In theory, the honest end user can detect the attack based on the information about the client instance provided by the AS, since the AS interacted with a different client instance than the end user. However, this requires corresponding knowledge on the part of the end user, which the end user often does not have in practice. If the end user authorizes the grant request of the honest client instance due to this lack of knowledge, the flow between the honest client instance and the AS is completed normally ([10]-[12]), and the attacker can access the resources of the honest end user with the privileges of the honest client instance after the access token has been transmitted to the honest client instance [12].

The possible scenarios in which the combination of interaction start mode and interaction finish mode enables this attack were addressed by the editors of GNAP in this pull request [23], which has been merged into the protocol. A section has been added to this pull request by the author of this thesis explaining the attack in more detail. The associated commit is available at `https://github.com/ietf-wg-gnap/gnap-core-protocol/pull/390/commits/b028a1e363e90ad1c2711bd8244ea76e3957f935`.

Appendix A.3 describes how we dealt with this attack in our model.

4 Related Work

We focus on other security analyses of GNAP as well as other WIM-based formal security analyses of comparable standards such as OAuth 2.0.

Fett et al. [7] performed the first extensive formal analysis of the OAuth 2.0 standard using the WIM. They were able to find four attacks for which they proposed fixes to prove the security properties they had defined. One of these attacks is the 307 redirect attack, for which we found, as detailed in Section 3.1, that it can also be applied to GNAP. Fett et al. [5] performed an analysis of the OpenID Financial-grade API, which found, among other vulnerabilities, the cuckoo token attack, which can also be applied to GNAP, as seen in Section 3.2. Furthermore, Fett et al. [10] conducted a WIM-based analysis of the OpenID Connect standard, which is built on OAuth 2.0.

Axeland and Oueidat [1] performed a security analysis of attack surfaces on GNAP. In doing so, they looked at legacy attacks on OAuth 2.0 and evaluated whether they could be applied to GNAP as well. They applied the IdP mix-up attack found by Fett et al. [7] to GNAP and found that GNAP is also vulnerable to it. This attack has since been fixed in GNAP when using an interaction finish mode, but the attack can still be carried out when not using an interaction finish mode [26]. They also tested common attacks such as redirect attacks and Cross-Site Request Forgery (CSRF), whereby they did not find any vulnerability. Compared to this work, they have only evaluated specific known attacks in the context of GNAP, but have not subjected GNAP to a formal analysis through proving security properties. They have also focused on interaction using the redirect interaction modes, as these are closest to OAuth 2.0, while this work also considers the user code interaction start mode and the push interaction finish mode.

5 Conclusion and Outlook

In this thesis, we have modeled and analyzed the Grant Negotiation and Authorization Protocol in the Web Infrastructure Model. Our model covers all roles involved, namely the client instance, the authorization server, the resource server, and the end user/resource owner. Furthermore, our model allows arbitrary combinations of the redirect interaction start mode and the user code interaction start mode with the redirect interaction finish mode and the push interaction finish mode. In addition, we have covered token introspection, requesting subject identifiers, and software-only authorization in our model.

For modeling the user code interaction start mode, we extended the browser model of the Web Infrastructure Model by simulating entering a user code at an authorization server.

As part of our work, we found that two attacks on similar protocols, the 307 redirect attack, and the cuckoo token attack, can also be applied to GNAP. For those attacks, as well as the client instance mix-up attack which was discovered during the work on this thesis, we have worked out security considerations together with the editors of GNAP to address and prevent these attacks. In addition, we pointed out to the working group a security problem in deriving so-called *downstream tokens* and several insufficiently specified particulars, such as the use of symmetric keys together with token introspection.

In our analysis, we proved that if the aforementioned security considerations are taken into account, our authorization property for end user-owned resources and our authorization property for software-only authorization hold.

Outlook

In addition to the authorization property we have shown for end user-owned resources, the question arises whether session integrity also applies to authorization. Since GNAP can also be used as an identity protocol, it should also be examined whether authentication and session integrity for authentication are fulfilled when a user logs in to a client instance using GNAP. As our model already contains such a login functionality, it can be used as a starting point for such an analysis.

Once the security problem regarding the derivation of downstream tokens is solved and this feature is further detailed, it would be worthwhile to extend our model with this functionality and investigate whether it results in new attacks. The transmission of assertions between client instances and authorization servers, which we have left out of our model, could be another interesting extension.

Bibliography

[1] Å. Axeland, O. Oueidat. “Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol”. MA thesis. Chalmers University of Technology and University of Gothenburg, 2021. URL: `https://hdl.handle.net/20.500.12380/304105` (cit. on p. 37).

[2] A. Backman, J. Richer, M. Sporny. *HTTP Message Signatures*. Internet-Draft draft-ietf-httpbis-message-signatures-06. Work in Progress. Internet Engineering Task Force, Aug. 13, 2021. 59 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-message-signatures-06` (cit. on pp. 20, 57).

[3] A. Backman, M. Scurtescu. *Subject Identifiers for Security Event Tokens*. Internet-Draft draft-ietf-secevent-subject-identifiers-08. Work in Progress. Internet Engineering Task Force, May 24, 2021. 22 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-secevent-subject-identifiers-08` (cit. on pp. 20, 21, 56).

[4] B. Campbell, J. Bradley, N. Sakimura, T. Lodderstedt. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. RFC 8705. Feb. 2020. DOI: `10.17487/RFC8705`. URL: `https://rfc-editor.org/rfc/rfc8705.txt` (cit. on p. 20).

[5] D. Fett, P. Hosseyni, R. Küsters. “An Extensive Formal Security Analysis of the OpenID Financial-Grade API”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 453–471. DOI: `10.1109/SP.2019.00067` (cit. on pp. 18, 31, 37).

[6] D. Fett, P. Hosseyni, R. Küsters. *An Extensive Formal Security Analysis of the OpenID Financial-grade API*. 2019. arXiv: `1901.11520 [cs.CR]` (cit. on pp. 51, 58, 76, 99, 102).

[7] D. Fett, R. Küsters, G. Schmitz. “A Comprehensive Formal Security Analysis of OAuth 2.0”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1204–1215. ISBN: 9781450341394. DOI: `10.1145/2976749.2978385`. URL: `https://doi.org/10.1145/2976749.2978385` (cit. on pp. 18, 31, 37).

[8] D. Fett, R. Küsters, G. Schmitz. “A Comprehensive Formal Security Analysis of OAuth 2.0”. In: *CoRR* abs/1601.01229 (2016). arXiv: `1601.01229`. URL: `http://arxiv.org/abs/1601.01229` (cit. on pp. 51, 55, 56, 113).

[9] D. Fett, R. Küsters, G. Schmitz. *The Web Infrastructure Model (WIM)*. Tech. rep. Version 1.0. SEC, University of Stuttgart, Germany, 2022. URL: `https://www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf` (cit. on pp. 46, 50, 56, 57, 62, 77, 90, 95, 99).

[10] D. Fett, R. Küsters, G. Schmitz. “The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 2017, pp. 189–202. DOI: `10.1109/CSF.2017.20` (cit. on p. 37).

[11] D. Fett, R. Küsters, G. Schmitz. *The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines*. 2017. arXiv: `1704.08539 [cs.CR]` (cit. on p. 57).

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

[13] F. Helmschmidt. *Add redirection status code security considerations*. GitHub Pull Request. Nov. 2021. URL: https://github.com/ietf-wg-gnap/gnap-core-protocol/pull/351 (cit. on p. 31).

[14] F. Helmschmidt. *End user/client instance mix-up attack*. GitHub Issue. Dec. 2021. URL: https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/364 (cit. on p. 33).

[15] F. Helmschmidt. *GitHub contributions to the gnap-core-protocol repository*. 2021. URL: https://github.com/ietf-wg-gnap/gnap-core-protocol/issues?q=author:pq2 (cit. on p. 18).

[16] F. Helmschmidt. *GitHub contributions to the gnap-resource-servers repository*. 2021. URL: https://github.com/ietf-wg-gnap/gnap-resource-servers/issues?q=author:pq2 (cit. on p. 18).

[17] F. Helmschmidt. *How does token introspection handle symmetric keys?* GitHub Issue. Nov. 2021. URL: https://github.com/ietf-wg-gnap/gnap-resource-servers/issues/47 (cit. on p. 78).

[18] M. Jones. *JSON Web Key (JWK)*. RFC 7517. May 2015. DOI: 10.17487/RFC7517. URL: https://rfc-editor.org/rfc/rfc7517.txt (cit. on pp. 57, 58).

[19] M. Jones, J. Bradley, N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: https://rfc-editor.org/rfc/rfc7515.txt (cit. on p. 21).

[20] P. Kasselmann, D. Fett. *Risk Mitigation for Cross Device Flows*. Presentation at the OAuth Security Workshop 2021. Dec. 2021. URL: https://www.youtube.com/watch?v=6tpmyLf88pU (cit. on p. 33).

[21] T. Lodderstedt, J. Bradley, A. Labunets, D. Fett. *OAuth 2.0 Security Best Current Practice*. Internet-Draft draft-ietf-oauth-security-topics-19. Work in Progress. Internet Engineering Task Force, Dec. 2021. 52 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-19 (cit. on p. 31).

[22] *Recording of the GNAP meeting session at IETF 112*. YouTube Video. Nov. 2021. URL: https://www.youtube.com/watch?v=E6jRFCmivSo (cit. on p. 31).

[23] J. Richer. *Clarified user presence on interaction finish methods*. GitHub Pull Request. Feb. 2022. URL: https://github.com/ietf-wg-gnap/gnap-core-protocol/pull/390 (cit. on p. 35).

[24] J. Richer. *Mitigate Cuckoo Token Attack*. GitHub Pull Request. Dec. 2021. URL: https://github.com/ietf-wg-gnap/gnap-core-protocol/pull/360 (cit. on p. 33).

[25] J. Richer. *Security Consideration: Derivation of Stolen Tokens*. GitHub Issue. Dec. 2021. URL: https://github.com/ietf-wg-gnap/gnap-resource-servers/issues/48 (cit. on p. 18).

[26] J. Richer, A. Parecki, F. Imbault. *Grant Negotiation and Authorization Protocol*. Internet-Draft draft-ietf-gnap-core-protocol-08. Work in Progress. Internet Engineering Task Force, Oct. 2021. 157 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-gnap-core-protocol-08 (cit. on pp. 17, 19–21, 25, 37, 52, 58).

[27] J. Richer, A. Parecki, F. Imbault. *Grant Negotiation and Authorization Protocol Resource Server Connections*. Internet-Draft draft-ietf-gnap-resource-servers-01. Work in Progress. Internet Engineering Task Force, July 12, 2021. 16 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-gnap-resource-servers-01 (cit. on p. 17).

[28] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, C. Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Foundation. Nov. 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html (cit. on pp. 20, 21).

All links were last followed on March 18, 2022.

A Formal Model of GNAP

This appendix contains our formal model of GNAP, which is used to prove the security properties defined in Appendix C. 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

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

with $n \in \mathcal{N}$ and $\mathit{scheme} \in \{\mathtt{GNAP}, \mathtt{Bearer}\}$. 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:

- $\langle \mathtt{Digest}, t \rangle$ 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., $\mathsf{hash}(m.\mathtt{body})$. This header is used together with the `Signature-Input` header and the `Signature` header to simulate HTTP Message Signature key proofs.
- $\langle \mathtt{Signature\text{-}Input}, t \rangle$ 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 Message Authentication Code (MAC).

A.1.2 Browser Model

To use the user code interaction start mode we have adapted the browser model of the WIM [9]. 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 Dolev-Yao (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
- $\mathit{usedCIs} \in \left[\mathcal{N} \times \mathsf{Doms}\right]$ 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 $\langle \texttt{STARTGRANT}, \mathit{url}, \mathit{as}\rangle$. 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.

```
1: function RUNSCRIPT(w̄, d̄, s′)
       ⋮
18:    switch command do
19:        case ⟨STARTGRANT, url, as⟩
20:            let reference := ⟨START, s′.w̄.nonce⟩
21:            let req := ⟨HTTPReq, ν4, POST, url.host, url.path, url.parameters, ⟨⟩, as⟩
22:            let s′ := CANCELNAV(reference, s′)
23:            call HTTP_SEND(reference, req, url, docorigin, referrer, referrerPolicy, s′)
24:        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 38), and the user code together with the domain of the client instance used for the request in *usedCIs* (Line 39).

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

```
1:  function PROCESSRESPONSE(response, reference, request, requestUrl, key, f, s')
        ⋮
26:             let referrerPolicy := response.headers[ReferrerPolicy]
27:             if π1(reference) ≡ START then
28:                 let reference := ⟨REQ, π2(reference)⟩    → Redirect interaction start mode is used
29:             call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, s')
            else
30:             stop ⟨⟩, s'
31:     switch π1(reference) do
32:         case START
33:             if userCode ∉ response ∨ userCodeUrl ∉ response then
34:                 stop ⟨⟩, s'     → The response must contain a user code and the URI of the user
                                      code interaction page
35:             let userCode := response[userCode]
36:             let userCodeUrl := response[userCodeUrl]
37:             let domainCI := requestUrl.host
38:             let s'.pendingInteractions := s'.pendingInteractions
                 ↪ +⟨⟩ ⟨userCode, userCodeUrl⟩
39:             let s'.usedCIs := s'.usedCIs +⟨⟩ ⟨userCode, domainCI⟩
40:         case UCL
41:             let w̄ ← Subwindows(s') such that s'.w̄.nonce ≡ π2(reference) if possible;
                 ↪ otherwise stop
42:             if response.body ≁ ⟨∗, ∗⟩ then
43:                 stop ⟨⟩, s'
44:             let script := π1(response.body)
45:             let domainCI := π2(response.body)
46:             let userCode := requestUrl.parameters[user-code]
47:             let domainUsedCI := s'.usedCIs[userCode]
48:             if domainCI ≡ domainUsedCI then
49:                 let scriptinputs := [userCode:userCode]
50:             else
51:                 let scriptinputs := ⟨⟩
52:             let s'.pendingInteractions := s'.pendingInteractions − userCode
53:             let s'.usedCIs := s'.usedCIs − userCode
54:             let d := ⟨ν7, requestUrl, response.headers, referrer, script, ⟨⟩, scriptinputs, ⟨⟩, ⊤⟩
55:             if s'.w̄.documents ≡ ⟨⟩ then
56:                 let s'.w̄.documents := ⟨d⟩
57:             else
58:                 let ī ← ℕ such that s'.w̄.documents.ī.active ≡ ⊤
59:                 let s'.w̄.documents.ī.active := ⊥
60:                 remove s'.w̄.documents.(ī + 1) and all following documents
                     ↪ from s'.w̄.documents
61:                 let s'.w̄.documents := s'.w̄.documents +⟨⟩ d
62:             stop ⟨⟩, s'
63:         case REQ
        ⋮
```

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 3.3). 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 $\langle\rangle$ 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 48ff.). In Lines 52f. the used entries from *pendingInteractions* and *usedCIs* are removed so that they are not used again.

Algorithm A.3 Web Browser Model: Main algorithm.

```
Input: ⟨a, f, m⟩, s
        ⋮
 7: if m ≡ TRIGGER then
 8:     let switch ← {script, login, urlbar, reload, forward, back}
 9:     if switch ≡ script then
            ⋮
12:         call RUNSCRIPT(w̄, d̄, s′)
13:     else if switch ≡ login then     → Perform login using user code
14:         if s′.pendingInteractions ≡ ⟨⟩ then
15:             stop     → No user code has been received yet or all received user codes have been used
16:         let newwindow ← {⊤, ⊥}
17:         if newwindow ≡ ⊤ then     → Create a new window
18:             let windownonce := ν1
19:             let w′ := ⟨windownonce, ⟨⟩, ⊥⟩
20:             let s′.windows := s′.windows +⟨⟩ w′
21:         else     → Use existing top-level window
22:             let windownonce := s′.tlw.nonce
23:         let ⟨userCode, userCodeUrl⟩ ← s′.pendingInteractions
24:         let req := ⟨HTTPReq, ν2, GET, userCodeUrl.host, userCodeUrl.path,
             ↪ [user-code:userCode], ⟨⟩, ⟨⟩⟩
25:         call HTTP_SEND(⟨UCL, windownonce⟩, req, url, ⊥, ⊥, ⊥, s′)
26:     else if switch ≡ urlbar then
        ⋮
59: else if m ≡ CLOSECORRUPT then
60:     let s′.pendingInteractions := ⟨⟩
61:     let s′.usedCIs := ⟨⟩
62:     let s′.secrets := ⟨⟩
        ⋮
```

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 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 [9] to describe states.

A.2 Outline

We model GNAP as a web system in the WIM as defined in [9].

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 Fett et al. [8], 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 Domain Name System (DNS) servers, as they are subsumed by the network attacker. Table A.1 shows the set of scripts $\mathsf{Scripts}$ 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 A.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.

Modeling of Mutual TLS (MTLS) GNAP uses MTLS as one of its key proofing methods. MTLS has already been modeled within the WIM by Fett et al. [6], 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. In Figure 2.2, these can be seen between steps 7 and 8 and steps 8 and 9. The model does not include these responses because an attacker cannot extract any information from them. We do not model error messages.

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. 26]. 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 MTLS As recommended by GNAP in response to the cuckoo token attack described in Section 3.2, we use each 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.

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, but not the app mode, which we consider out of scope for this work.

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/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 As mentioned in Section 2.3.2, 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.

Assertions We do not model the ability to transmit assertions from an AS to a client instance or vice versa, as this optional GNAP feature is beyond the scope of this thesis.

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 security problem mentioned in Chapter 1.

Transfer of Subject Identifiers from Client Instance 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.

Handling the Client Instance Mix-Up Attack The following describes how we dealt with the client instance mix-up attack presented in Section 3.3 in our model.

When using the redirect interaction finish mode, the attack cannot be performed in our model because the AS would redirect the browser to the honest client instance after the interaction. The honest client instance would then detect the attack because the browser would not transmit a session identifier to the client instance, since the browser has previously talked to the client instance controlled by the attacker and the attacker cannot set a session identifier for the honest client instance because it uses a different domain.

If the redirect interaction start mode is used together with the push interaction finish mode, the attack is detected by the AS after the redirect by the attacker-controlled client instance. In our model, the AS checks the `Referer` header after the redirect. If the domain specified in the header does not match the domain of the client instance that sent the grant request to the AS, the AS rejects the interaction with the browser. Note that this fix is only possible if redirects are used in the redirect interaction start mode (which, contrary to the name of this mode, is not necessarily the case). For example, if only a browser with the interaction URL is started from a client instance installed locally on the end user's device, no `Referer` header is sent, so this fix is not applicable in this case.

If the user code interaction start mode is used together with the push interaction finish mode, the attack is prevented by our adjustments to the browser model described in Appendix A.1.2. These simulate that the end user detects the attack due to the information provided by the AS during the interaction and therefore does not enter the user code so that the grant request is not authorized. For this purpose, the browser remembers the domain of the client instance used by the browser and the AS transmits the domain of the client instance that sent the grant request. Only if these two domains match, the script sent by the AS receives the user code as input via the *scriptinputs*. If the domains do not match, the AS does not receive any user code, so the grant request cannot get authorized.

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 [8], 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 [8], the set $\mathcal{N}$ of nonces is partitioned into six sets, the infinite sequence N and finite sets K_{TLS}, K_{KP}, KeyIDs, $\mathsf{Passwords}$, and $\mathsf{ProtectedResources}$. 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 $tlskeys^p = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(p)\}\rangle$.
- 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 $\mathsf{Passwords}$ 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 $\mathsf{ProtectedResources}$ 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 [8], 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 2
An *identity* i is a term of the form $\langle name, domain\rangle$ with $name \in \mathbb{S}$ and $domain \in \mathsf{Doms}$.

Let ID be the set of identities. By ID^y we denote the set $\{\langle name, domain\rangle \in \mathsf{ID} \mid domain \in \mathsf{dom}(y)\}$.

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

The governor of an ID will usually be an AS, but could also be the attacker. Besides $\mathsf{governor}$, 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 [3].

A.7 Corruption

Similar to [8], 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 $\bot$, 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 [9]. As in [8], we allow it to listen to/spoof all available IP addresses, and hence, define $I^{na} = \mathsf{IPs}$. The initial state is $s_0^{na} = \langle attdoms, tlskeys, keyproofkeys\rangle$, 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 [9], with $I^b := \mathsf{addr}(b)$ being its addresses.

We define the initial state similar to [11]. First let $\mathsf{ID}_b := \mathsf{ownerOfID}^{-1}(b)$ be the set of all IDs of 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

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

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 [9]. The algorithm simulates the HTTP Message Signature (`httpsig`) key proofing method [2]. 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 [18]. *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 $\perp$. *reference*, s', and a are required as input parameters for HTTPS_SIMPLE_SEND.

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

```
1: function SIGN_AND_SEND(HTTPMethod, url, keyID, key, keyProof, authHeader, body,
   ↪ reference, s', a)
2:     let sigInput := [method:HTTPMethod, targetURI:url]
3:     let sigParams := [covered:⟨method, targetURI⟩, keyID:keyID, nonce:ν_n3]
4:     if body ≢ ⟨⟩ then    → If the message contains a body, its digest must be signed
5:         let sigInput[contentDigest] := hash(body)
6:         let sigParams[covered] := sigParams[covered] +⟨⟩ contentDigest
7:     if authHeader ≢ ⊥ then    → If present, the AuthZ header must be covered by the signature
8:         let sigInput := sigInput +⟨⟩ authHeader
9:         let sigParams[covered] := sigParams[covered] +⟨⟩ authorization
10:    let sigInput[sigParams] := sigParams
11:    if keyProof ≡ sign then
12:        let signature := sig(siginput, key)
13:    else
14:        let signature := mac(siginput, key)
15:    let headers := [Signature-Input:sigParams, Signature:signature]
16:    if body ≢ ⟨⟩ then
17:        let headers[Digest] := hash(body)
18:    if authHeader ≢ ⊥ then
19:        let headers := headers +⟨⟩ authHeader
20:    let req := ⟨HTTPReq, ν_n4, HTTPMethod, url.host, url.path, ⟨⟩, headers, body⟩
21:    call HTTPS_SIMPLE_SEND(reference, req, s', a)
```

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* ≡ `sign`, this algorithm simulates the validation of an HTTP Message Signature key proof using the public key *key*. If *method* ≡ `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 [18]. If *method* ≡ `mtls`, this algorithm finishes an MTLS key proof by verifying the MTLS nonce sent by the requester[1] 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 [26].

[1] See [6] for an explanation of how MTLS is modeled within the WIM.

Algorithm A.5 Helper Functions: Validating key proofs.

```
1:  function VALIDATE_KEY_PROOF(method, m, keyID, key, s′)
2:      if method ≡ sign ∨ method ≡ mac then    → HTTP Message Signature
3:          if m.body ≢ ⟨⟩ then
4:              let digest := m.headers[Digest]
5:              if digest ≢ hash(m.body) then
6:                  stop
7:          let sigParams := m.headers[Signature-Input]
8:          let signature := m.headers[Signature]
9:          if keyID ≢ sigParams[keyID] then
10:             stop
11:         let covered := sigParams[covered]
12:         if method ∉⟨⟩ covered ∨ targetURI ∉⟨⟩ covered
            ↪ ∨ (m.body ≢ ⟨⟩ ∧ contentDigest ∉⟨⟩ covered)
            ↪ ∨ (Authorization ∈ m.headers ∧ authorization ∉⟨⟩ covered) then
13:             stop    → The signature does not cover all required parts of the message
14:         if nonce ∉ sigParams ∨ sigParams[nonce] ∈⟨⟩ s′.sigNonces then
15:             stop    → Replay protection
16:         let controlURL := ⟨URL, S, m.host, m.path, m.parameters, ⟨⟩⟩
17:         let controlInput := [method:m.method, targetURI:controlURL]
18:         if m.body ≢ ⟨⟩ then
19:             let controlInput[contentDigest] := hash(m.body)
20:         if Authorization ∈ m.headers then
21:             let controlInput := controlInput +⟨⟩ ⟨Authorization, m.headers[Authorization]⟩
22:         let controlInput[sigParams] := sigParams
23:         if controlInput ≢ extractmsg(signature) then
24:             stop    → The signature was not created for the correct input
25:         if method ≡ sign then
26:             if checksig(signature, key) ≢ ⊤ then
27:                 stop    → Invalid signature
28:         else    → method ≡ mac
29:             if checkmac(signature, key) ≢ ⊤ then
30:                 stop    → Invalid MAC
31:         let s′.sigNonces := s′.sigNonces +⟨⟩ sigParams[nonce]
32:     else if method ≡ mtls then
33:         let mtlsNonce := m.body[mtlsNonce]
34:         let mtlsInfo such that mtlsInfo ∈⟨⟩ s′.mtlsRequests
            ↪ ∧ mtlsInfo.1 ≡ mtlsNonce if possible; otherwise stop
35:         let s′.mtlsRequests := s′.mtlsRequests − mtlsInfo.1
36:         if mtlsInfo.2 ≢ key then
37:             stop    → Key used for MTLS does not match the key of the sender
38:     else
39:         stop    → Unsupported method
40:     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 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 $\mathit{instanceID} \not\equiv \bot$, *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 $r.\mathtt{method} \equiv \mathtt{mac}$, *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 4
A *state* $s \in Z^c$ *of client instance* c is a term of the form $\langle$*DNSaddress*, *pendingDNS*, *corrupt*, *pendingRequests*, *keyMapping*, *tlskeys*, *keyRecords*, *authServers*, *resourceServers*, *sessions*, *grants*, *receivedValues*, *browserRequests*$\rangle$ with $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{pendingRequests} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{keyMapping} \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$, $\mathit{keyRecords} \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{authServers} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{resourceServers} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{sessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{grants} \in [\mathcal{N} \times [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]]$, $\mathit{receivedValues} \in [\mathcal{N} \times [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]]$, and $\mathit{browserRequests} \in [\mathcal{N} \times [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]]$,

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 $as \in \mathsf{AS}$ with $d \in \mathsf{dom}(as)$.

$s_0^c.\mathtt{resourceServers}$ 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 $rs \in \mathsf{RS}$ with $d \in \mathsf{dom}(rs)$.

$s_0^c.\mathtt{keyRecords}$ 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 $as = \mathsf{dom}^{-1}(d)$ 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 $d \in^{\langle\rangle} s_0^c.\mathtt{authServers}$. Let $R := \left\langle \bigcup_{d \in s_0^c.\mathtt{keyRecords}} \bigcup_{r \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[d]} r \right\rangle$ be a sequence containing all key records in $s_0^c.\mathtt{keyRecords}$. 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., $d \in \mathsf{dom}(as) \wedge d' \in \mathsf{dom}(as)$), 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 $r.\mathtt{key}$ of each key record $r \in^{\langle\rangle} R$ with $r.\mathtt{method} \in \{\mathtt{sign}, \mathtt{mtls}\}$ appears only as a public key in s_0^p. If $r.\mathtt{method} \equiv \mathtt{mac}$, $r.\mathtt{key}$ must only be initially stored in c, $\mathsf{dom}^{-1}(r.\mathtt{rs})$, and the AS as that has the domain d in $\mathsf{dom}(as)$ for which $r \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[d]$.

We now specify the relation R^c: This relation is based on the generic HTTPS server model defined in [9]. 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 A.2 shows a list of all placeholders used.

Placeholder	Usage
ν_1	new grant ID
ν_2	new nonce to generate a unique interaction finish URL
ν_3	new 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 A.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 [9].

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 $m.\mathtt{path} \equiv /$. 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'.`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'.`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 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.

```
1: 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.
2:   if m.path ≡ / ∧ m.method ≡ GET then   → Serve index page
3:     let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨script_ci_index, ⟨⟩⟩⟩, k)
       → Send script_ci_index in HTTP response.
4:     stop ⟨⟨f, a, m'⟩⟩, s'
5:   else if m.path ≡ /startGrantRequest ∧ m.method ≡ POST then   → Start a new grant request
6:     let domainAS := m.body   → Domain of the AS to send the grant request to
7:     if domainAS ∉⟨⟩ s'.authServers then
8:       stop   → c is not configured to use this AS
9:     let endpoint := ⟨URL, S, domainAS, /requestGrant, ⟨⟩⟩   → Endpoint for the grant request
10:    let grantID := ν1   → Identifier for this grant request
11:    let inquiredValues := GENERATE_INQUIRED_VALUES(⊤)
12:    let finishMode ← {redirect, push}   → Non-det. select the used interaction finish mode
13:    if finishMode ≡ redirect then
14:      let finishURL := ⟨URL, S, m.host, /finish, [request:ν2]⟩
15:    else
16:      let finishURL := ⟨URL, S, m.host, /push, [request:ν2]⟩
17:    let finish := [finishMode:finishMode, finishURL:finishURL, nonce:ν3]
18:    let grantRequest := [inquiredValues:inquiredValues, finish:finish, interaction:⊤]
       → interaction:⊤ signals the AS that interaction with the RO is possible
19:    let s'.browserRequests[grantID][startRequest] := ⟨k, a, f, m.nonce⟩
20:    if ⟨__Host, sessionID⟩ ∈ m.headers[Cookie] then   → The browser sent a session ID
21:      let sessionID := m.headers[Cookie][⟨__Host, sessionID⟩]
22:      if domainAS ∈ s'.sessions[sessionID] then   → Check if a subject identifier is stored
                                                       for this session ID and this AS
23:        let grantRequest[user] := s'.sessions[sessionID][domainAS]
       → Include previously received subject identifier in grant request
24:    else
25:      let sessionID := ν4
26:    let keyRecord ← s'.keyRecords[domainAS]   → Non-det. select key record for this AS
27:    let s'.grants[grantID] := [AS:domainAS, sessionID:sessionID, keyRecord:keyRecord,
     ↪ finishURLnonce:ν2, CIfinishNonce:ν3, requested:inquiredValues]
28:    if keyRecord.instanceID ≢ ⊥ then   → key is registered at the AS used
29:      let grantRequest[instanceID] := keyRecord.instanceID
30:      let s'.grants[grantID][request] := grantRequest
31:      if keyRecord.method ≢ mtls then
32:        let reference := [responseTo:grantResponse, grantID:grantID]
33:        call SIGN_AND_SEND(POST, endpoint, keyRecord.keyID, keyRecord.key,
         ↪ keyRecord.method, ⊥, grantRequest, reference, s', a)
34:      else
35:        let body := [instanceID:keyRecord.instanceID]
36:        let message := ⟨HTTPReq, ν5, POST, domainAS, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
37:        let reference := [responseTo:MTLS_GR, grantID:grantID]
38:        call HTTPS_SIMPLE_SEND(reference, message, s', a)
```

```
39:         else    → c is not registered with the AS used
40:             let key := keyRecord.key
41:             if keyRecord.method ≡ sign then
42:                 let keyID := keyRecord.keyID
43:                 let grantRequest[client] := [keyID:keyID, key:pub(key), method:sign]
44:                 let s′.grants[grantID][request] := grantRequest
45:                 let reference := [responseTo:grantResponse, grantID:grantID]
46:                 call SIGN_AND_SEND(POST, endpoint, keyID, key, sign, ⊥, grantRequest,
                     ↪ reference, s′, a)
47:             else    → MTLS is used as key proofing method
48:                 let grantRequest[client] := [key:pub(key), method:mtls]
49:                 let s′.grants[grantID][request] := grantRequest
50:                 let body := [publicKey:pub(key)]
51:                 let message := ⟨HTTPReq, v5, POST, domainAS, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
52:                 let reference := [responseTo:MTLS_GR, grantID:grantID]
53:                 call HTTPS_SIMPLE_SEND(reference, message, s′, a)
54:     else if m.path ≡ /finish ∧ m.method ≡ GET then    → Redirect interaction finish mode
55:         let finishURLnonce := m.parameters[request]
56:         let grantID such that s′.grants[grantID][finishURLnonce] ≡ finishURLnonce
         ↪ if possible; otherwise stop
57:         if s′.grants[grantID][request][finish][finishMode] ≢ redirect then
58:             stop    → Wrong interaction finish mode was used
59:         if m.headers[Cookie][⟨__Host, sessionID⟩]
         ↪ ≢ s′.grants[grantID][sessionID] then
60:             stop    → Browsers session identifier does not match the one from the grant request
61:         let s′.browserRequests[grantID][finishRequest] := ⟨k, a, f, m.nonce⟩
62:         let interactRef := m.parameters[interactRef]
63:         let hash := m.parameters[hash]
64:         call SEND_CONTINUATION_REQUEST(grantID, interactRef, hash, s′, a)
65:     else if m.path ≡ /push ∧ m.method ≡ POST then    → Push interaction finish mode
66:         let finishURLnonce := m.parameters[request]
67:         let grantID such that s′.grants[grantID][finishURLnonce] ≡ finishURLnonce
         ↪ if possible; otherwise stop
68:         if s′.grants[grantID][request][finish][finishMode] ≢ push then
69:             stop    → Wrong interaction finish mode was used
70:         let interactRef := m.body[interactRef]
71:         let hash := m.body[hash]
72:         call SEND_CONTINUATION_REQUEST(grantID, interactRef, hash, s′, a)
73:     else if m.path ≡ /getData ∧ m.method ≡ GET then
74:         let finishURLnonce := m.parameters[request]
75:         let grantID such that s′.grants[grantID][finishURLnonce] ≡ finishURLnonce
         ↪ if possible; otherwise stop
76:         if s′.grants[grantID][request][finish][finishMode] ≢ push then
77:             stop    → This endpoint is only used when the push interaction finish mode is used
78:         if m.headers[Cookie][⟨__Host, sessionID⟩]
         ↪ ≢ s′.grant[grantID][sessionID] then
79:             stop    → Browsers session identifier does not match the one from the grant request
```

```
80:         let s'.browserRequests[grantID][finishRequest] := ⟨k, a, f, m.nonce⟩
81:         if domainFirstRS ∈ s'.grants[grantID] ∧ s'.grants[grantID][domainFirstRS] ∈
             ↪ s'.grants[grantID][resources] then    → Resource has already been received
                                                        from the RS
82:             call SEND_RESPONSE_TO_BROWSER(grantID, s')
83:         else
84:             stop ⟨⟩, s'
85:     else if m.path ≡ /logout ∧ m.method ≡ POST then
            → Set a new session ID so that the browser instance can log in with a
              different identity at already used ASs
86:         if ⟨__Host, sessionID⟩ ∈ m.headers[Cookie] then    → The browser sent a session ID
87:             let oldSessionID := m.headers[Cookie][⟨__Host, sessionID⟩]
88:             if oldSessionID ∈ s'.sessions then
89:                 let s'.sessions := s'.sessions − oldSessionID    → Delete data of the old session
90:         let headers := [Set-Cookie:⟨⟨⟨__Host, sessionID⟩, ⟨v4, ⊤, ⊤, ⊤⟩⟩⟩]
91:         let m' := enc_s(HTTPResp, m.nonce, 200, headers, ⟨script_ci_index, ⟨⟩⟩, k)
92:         stop ⟨⟨f, a, m'⟩⟩, s'
93:     else
94:         stop    → Unsupported operation
```

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

```
 1: function PROCESS_HTTPS_RESPONSE(m, reference, request, a, f, s')
 2:     let grantID := reference[grantID]
 3:     let grantRequest := s'.grants[grantID]
 4:     let domainAS := grantRequest[AS]
 5:     let sessionID := grantRequest[sessionID]
 6:     let keyRecord := grantRequest[keyRecord]
 7:     if reference[responseTo] ≡ grantResponse then
 8:         let grantResponse := m.body
 9:         if instanceID ∈ grantResponse then    → The AS has registered c
10:             if keyRecord.instanceID ≡ ⊥ then    → c was not already registered
11:                 let ī ← ℕ such that s'.keyRecords[domainAS].ī ≡ keyRecord
12:                 let s'.keyRecords[domainAS].ī.instanceID := grantResponse[instanceID]
13:         if subjectID ∈ grantResponse then
14:             if subjectID ∉⟨⟩ grantRequest[requested] then
15:                 stop    → AS returned subject identifier that was not requested
16:             let subjectID := grantResponse[subjectID]
17:             let s'.sessions[sessionID][domainAS] := subjectID    → Store subject identifier of this
                                                                      end user at this AS
18:             let s'.receivedValues[grantID][subjectID] := subjectID
19:         if accessToken ∈ grantResponse then
20:             if accessToken ∉⟨⟩ grantRequest[requested]
                 ↪ ∨ bearerToken ∉⟨⟩ grantRequest[requested] then
21:                 stop    → No (bearer) access token was requested
22:             if accessToken ∈⟨⟩ grantRequest[requested]
                 ↪ ∧ grantResponse[accessToken][flags] ≡ bearer then
23:                 stop    → Access token was requested, but bearer token was issued
24:             if bearerToken ∈⟨⟩ grantRequest[requested]
                 ↪ ∧ grantResponse[accessToken][flags] ≢ bearer then
25:                 stop    → Bearer token was requested, but access token was issued
26:             let s'.receivedValues[grantID][accessToken] := grantResponse[accessToken]
```

```
27:     if interact ∈ grantResponse then    → Interaction is required
28:         if finishedInteraction ∈ grantRequest then
29:             stop    → Interaction has already been completed
30:         if interaction ∉ grantRequest[request] then
31:             stop    → Interaction is required, but c did not indicate support for interaction
32:         if continue ∉ grantResponse then
33:             stop    → c needs to be allowed to continue once the interaction is finished
34:         let s'.grants[grantID][continueAT] := grantResponse[continue][accessToken]
35:         let s'.grants[grantID][continueURL] := grantResponse[continue][url]
36:         let s'.grants[grantID][ASfinishNonce] := grantResponse[interact][finish]
37:         let ⟨key, receiver, sender, nonce⟩ := s'.browserRequests[grantID][startRequest]
38:         let cookies := ⟨⟨⟨__Host, sessionID⟩, ⟨sessionID, ⊤, ⊤, ⊤⟩⟩⟩
39:         let startMode ← {redirect, userCode}
40:         if startMode ≡ redirect then
41:             let redirectURL := grantResponse[interact][redirect]
42:             let s'.grants[grantID][redirectNonce] := redirectURL.parameters[request]
43:             let m' := enc_s(⟨HTTPResp, nonce, 303,
                ↪ [Location:redirectURL, Set-Cookie:cookies], ⟨⟩⟩, key)
44:             stop ⟨⟨sender, receiver, m'⟩⟩, s'
45:         else    → user code interaction start mode
46:             let url := ⟨URL, S, domainAS, /interactUC, ⟨⟩⟩
47:             let userCode := grantResponse[interact][userCode]
48:             let s'.grants[grantID][userCode] := userCode
49:             let m' := enc_s(⟨HTTPResp, nonce, 200, [Set-Cookie:cookies],
                ↪ [userCodeUrl:url, userCode:userCode]⟩, key)
50:             stop ⟨⟨sender, receiver, m'⟩⟩, s'
51:     else if continue ∈ grantResponse then    → c can continue and no interaction is required
52:         let continue ← {⊤, ⊥}    → Non-det. decide whether to continue
53:         if continue ≡ ⊤ then    → Request values again using a PATCH request
54:             if interaction ∈ grantRequest[request] then
55:                 let inquiredValues := GENERATE_INQUIRED_VALUES(⊤)
56:             else
57:                 let inquiredValues := GENERATE_INQUIRED_VALUES(⊥)
58:             let s'.grants[grantID][requested] := inquiredValues
59:             let continueAT := grantResponse[continue][accessToken]
60:             let continueURL := grantResponse[continue][url]
61:             let authHeader := ⟨Authorization, ⟨GNAP, continueAT⟩⟩
62:             let body := [inquiredValues:inquiredValues]
63:             if keyRecord.method ≢ mtls then
64:                 let ref := [responseTo:grantResponse, grantID:grantID]
65:                 call SIGN_AND_SEND(PATCH, continueURL, keyRecord.keyID, keyRecord.key,
                    ↪ keyRecord.method, authHeader, body, ref, s', a)
```

```
66:             else   → MTLS is used as key proofing method
67:                 let s'.grants[grantID][patchRequest] := ⟨authHeader,
                     ↪ body, continueURL⟩
68:                 if keyRecord.instanceID ≢ ⊥ then
69:                     let body := [instanceID:keyRecord.instanceID]
70:                 else
71:                     let body := [publicKey:pub(keyRecord.key)]
72:                 let message := ⟨HTTPReq, ν6, POST, continueURL.host, /MTLS-prepare, ⟨⟩,
                     ↪ ⟨⟩, body⟩
73:                 let ref := [responseTo:MTLS_PR, grantID:grantID]
74:                 call HTTPS_SIMPLE_SEND(ref, message, s', a)
75:         else
76:             stop   → AS rejected request without possibility to continue or grant response is invalid
77:     else if reference[responseTo] ≡ resourceResponse then
78:         let domainRS := reference[domainRS]
79:         let s'.grants[grantID][resources][domainRS] := m.body
80:         if finishRequest ∈ s'.browserRequests[grantID] ∧ domainRS ≡
             ↪ s'.grants[grantID][domainFirstRS] then
         → Browser awaits response and the resource of the first used RS was obtained
81:             call SEND_RESPONSE_TO_BROWSER(grantID, s')
82:         else
83:             stop ⟨⟩, s'
84:     else if reference[responseTo] ≡ MTLS_GR then   → A new grant request is to be sent
85:         let m_dec := dec_a(m.body, keyRecord.key)
86:         let mtlsNonce, pubKey such that ⟨mtlsNonce, pubKey⟩ ≡ m_dec if possible; otherwise stop
87:         if pubKey ≡ s'.keyMapping[request.host] then
88:             let body := grantRequest[request]
89:             let body[mtlsNonce] := mtlsNonce
90:             let req := ⟨HTTPReq, ν6, POST, domainAS, /requestGrant, ⟨⟩, ⟨⟩, body⟩
91:             let ref := [responseTo:grantResponse, grantID:grantID]
92:             call HTTPS_SIMPLE_SEND(ref, req, s', a)
93:         else
94:             stop   → Send nonce only to the process that created it
95:     else if reference[responseTo] ≡ MTLS_CR then   → A new continuation request is to be sent
96:         let m_dec := dec_a(m.body, keyRecord.key)
97:         let mtlsNonce, pubKey such that ⟨mtlsNonce, pubKey⟩ ≡ m_dec if possible; otherwise stop
98:         if pubKey ≡ s'.keyMapping[request.host] then
99:             let authHeader := ⟨Authorization, ⟨GNAP, s'.grants[grantID][continueAT]⟩⟩
100:            let interactRef := s'.grants[grantID][interactRef]
101:            let url := s'.grants[grantID][continueURL]
102:            let ref := [responseTo:grantResponse, grantID:grantID]
103:            if adjustedInquiredValues ∉ s'.grants[grantID] then
104:                let body := [interactRef:interactRef, mtlsNonce:mtlsNonce]
105:                let req := ⟨HTTPReq, ν6, POST, url.host, url.path, ⟨⟩, ⟨authHeader⟩, body⟩
106:            else
107:                let inquiredValues := s'.grants[grantID][requested]
108:                let body := [interactRef:interactRef, inquiredValues:inquiredValues,
                    ↪ mtlsNonce:mtlsNonce]
109:                let req := ⟨HTTPReq, ν6, PATCH, url.host, url.path, ⟨⟩, ⟨authHeader⟩, body⟩
110:            call HTTPS_SIMPLE_SEND(ref, req, s', a)
```

```
111:         else
112:             stop    → Send nonce only to the process that created it
113:     else if reference[responseTo] ≡ MTLS_PR then    → c modifies its grant request
114:         let m_dec := dec_a(m.body, keyRecord.key)
115:         let mtlsNonce, pubKey such that ⟨mtlsNonce, pubKey⟩ ≡ m_dec if possible; otherwise stop
116:         if pubKey ≡ s'.keyMapping[request.host] then
117:             let ⟨authHeader, body, url⟩ := s'.grants[grantID][patchRequest]
118:             let body[mtlsNonce] := mtlsNonce
119:             let req := ⟨HTTPReq, ν_6, PATCH, url.host, url.path, ⟨⟩, ⟨authHeader⟩, body⟩
120:             let ref := [responseTo:grantResponse, grantID:grantID]
121:             call HTTPS_SIMPLE_SEND(ref, req, s', a)
122:         else
123:             stop    → Send nonce only to the process that created it
124:     else if reference[responseTo] ≡ MTLS_RR then    → A new resource request is to be sent
125:         if key ∈ reference then    → Access token is bound to its own key
126:             let m_dec := dec_a(m.body, reference[key])
127:         else    → Access token is bound to key from key record
128:             let m_dec := dec_a(m.body, keyRecord.key)
129:         let mtlsNonce, pubKey such that ⟨mtlsNonce, pubKey⟩ ≡ m_dec if possible; otherwise stop
130:         if pubKey ≡ s'.keyMapping[request.host] then
131:             let ref := reference[reference]
132:             let req := reference[request]
133:             let req.body := [mtlsNonce:mtlsNonce]
134:             call HTTPS_SIMPLE_SEND(ref, req, s', a)
135:         else
136:             stop    → Send nonce only to the process that created it
```

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

```
1: function PROCESS_TRIGGER(s')
2:     let startGrantRequest ← {⊤, ⊥}
3:     if startGrantRequest ≡ ⊤ then    → Start software-only authorization
4:         let domainAS, ī such that s'.keyRecords[domainAS].ī.instanceID ≢ ⊥
           ↪ if possible; otherwise stop
5:         let keyRecord := s'.keyRecords[domainAS].ī
6:         let inquiredValues := GENERATE_INQUIRED_VALUES(⊥)
7:         let grantID := ν_7    → Identifier for this grant request
8:         let instanceID := keyRecord.instanceID
9:         let grantRequest := [inquiredValues:inquiredValues, instanceID:instanceID]
10:        let s'.grants[grantID] := [request:grantRequest, AS:domainAS,
           ↪ keyRecord:keyRecord, requested:inquiredValues]
11:        if keyRecord.method ≢ mtls then
12:            let endpoint := ⟨URL, S, domainAS, /requestGrant, ⟨⟩⟩
13:            let reference := [responseTo:grantResponse, grantID:grantID]
14:            call SIGN_AND_SEND(POST, endpoint, keyRecord.keyID, keyRecord.key,
               ↪ keyRecord.method, ⊥, grantRequest, reference, s', a)
15:        else
16:            let body := [instanceID:instanceID]
17:            let message := ⟨HTTPReq, ν_8, POST, domainAS, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
18:            let reference := [responseTo:MTLS_GR, grantID:grantID]
19:            call HTTPS_SIMPLE_SEND(reference, message, s', a)
```

```
20:     else   → Use received access token and/or subject identifier
21:         let grantID such that grantID ∈ s'.receivedValues if possible; otherwise stop
22:         let originallyInqValues := s'.grants[grantID][request][inquiredValues]
23:         if accessToken ∉⟨⟩ originallyInqValues ∧ bearerToken ∉⟨⟩ originallyInqValues
             ↪ ∧ finishRequest ∈ s'.browserRequests[grantID] then
            → Originally only a subject identifier was requested and browser is not yet logged in
24:             call SEND_RESPONSE_TO_BROWSER(grantID, s')
25:         else   → An access token was requested. If applicable, the response
                     to the browser is sent once the resource is received.
26:             let domainRS ← s'.resourceServers   → Non-det. choose an RS
27:             if (accessToken ∈⟨⟩ originallyInqValues ∨ bearerToken ∈⟨⟩ originallyInqValues)
                 ↪ ∧ domainFirstRS ∉ s'.grants[grantID] then
28:                 let s'.grants[grantID][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
29:             let accessToken := s'.receivedValues[grantID][accessToken]
30:             let value := accessToken[value]
31:             let reference := [responseTo:resourceResponse, grantID:grantID,
                 ↪ domainRS:domainRS]
32:             if accessToken[flags] ≡ bearer then   → Access token is a bearer token
33:                 let s'.grants[grantID][bearerRSs] := s'.grants[grantID][bearerRSs]+⟨⟩
                     ↪ domainRS   → Store the RSs to which bearer tokens were sent
34:                 let authHeader := ⟨Authorization, ⟨Bearer, value⟩⟩
35:                 let request := ⟨HTTPReq, ν8, GET, domainRS, /resource, ⟨⟩, ⟨authHeader⟩, ⟨⟩⟩
36:                 call HTTPS_SIMPLE_SEND(reference, request, s', a)
37:             else   → Access token is key-bound
38:                 let url := ⟨URL, S, domainRS, /resource, ⟨⟩⟩
39:                 let authHeader := ⟨Authorization, ⟨GNAP, value⟩⟩
40:                 if key ∈ accessToken then   → Access token is bound to its own key
41:                     let keyData := accessToken[key]
42:                     let method := keyData[method]
43:                     let privateKey := keyData[privateKey]
44:                     if method ≡ sign then
45:                         let keyID := keyData[keyID]
46:                         call SIGN_AND_SEND(GET, url, keyID, privateKey, sign, authHeader, ⟨⟩,
                             ↪ reference, s', a)
47:                     else if method ≡ mtls then
48:                         let request := ⟨HTTPReq, ν8, GET, domainRS, /resource, ⟨⟩, ⟨authHeader⟩, ⟨⟩⟩
49:                         let body := [publicKey:pub(privateKey)]
50:                         let message := ⟨HTTPReq, ν9, POST, domainRS, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
51:                         let ref := [responseTo:MTLS_RR, grantID:grantID, key:privateKey,
                             ↪ reference:reference, request:request]
            → Store privateKey in reference since it is not from a key record
52:                         call HTTPS_SIMPLE_SEND(ref, message, s', a)
53:                     else
54:                         stop   → Unsupported method
```

55: **else** → Access token is bound to client instances key
56: **let** $keyRecord$:= s'.grants[$grantID$][keyRecord]
57: **let** key := $keyRecord$.key
58: **if** $keyRecord$.method ≡ sign **then**
59: **let** $keyID$:= $keyRecord$.keyID
60: **call** SIGN_AND_SEND(GET, url, $keyID$, key, sign, $authHeader$, ⟨⟩,
↪ $reference$, s', a)
61: **else if** $keyRecord$.method ≡ mac **then**
62: **let** $keyID$:= $keyRecord$.keyID
63: **let** url' := ⟨URL, S, $keyRecord$.rs, /resource, ⟨⟩⟩
→ We have to use the RS with which this symmetric key is shared
64: **call** SIGN_AND_SEND(GET, url', $keyID$, key, mac, $authHeader$, ⟨⟩,
↪ $reference$, s', a)
65: **else** → $keyRecord$.method ≡ mtls
66: **let** $request$:= ⟨HTTPReq, ν_8, GET, $domainRS$, /resource, ⟨⟩, ⟨$authHeader$⟩, ⟨⟩⟩
67: **let** $body$:= [publicKey:pub(key)]
68: **let** $message$:= ⟨HTTPReq, ν_9, POST, $domainRS$, /MTLS-prepare, ⟨⟩, ⟨⟩, $body$⟩
69: **let** ref := [responseTo:MTLS_RR, grantID:$grantID$,
↪ reference:$reference$, request:$request$]
70: **call** HTTPS_SIMPLE_SEND(ref, $message$, s', a)

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

1: **function** GENERATE_INQUIRED_VALUES($ROpresent$)
2: **let** $inquiredValues$:= ⟨⟩
3: **let** $requestAccessToken$ ← {⊤, ⊥}
4: **if** $requestAccessToken$ ≡ ⊤ ∨ $ROpresent$ ≡ ⊥ **then**
5: **let** $setBearerFlag$ ← {⊤, ⊥}
6: **if** $setBearerFlag$ ≡ ⊤ **then**
7: **let** $inquiredValues$:= $inquiredValues$ $+^{\langle\rangle}$ bearerToken
→ Requested access token is a bearer token
8: **else**
9: **let** $inquiredValues$:= $inquiredValues$ $+^{\langle\rangle}$ accessToken
→ Requested access token is bound to a key
10: **if** $ROpresent$ ≡ ⊤ **then**
→ A subject identifier can only be requested when an RO is present, as otherwise we use software-only authorization and client instances have no subject identifiers
11: **let** $requestSubjectID$ ← {⊤, ⊥}
12: **if** $requestSubjectID$ ≡ ⊤ ∨ $requestAccessToken$ ≡ ⊥ **then**
13: **let** $inquiredValues$:= $inquiredValues$ $+^{\langle\rangle}$ subjectID
14: **return** $inquiredValues$

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

```
1:  function SEND_CONTINUATION_REQUEST(grantID, interactRef, hash, s', a)
2:      let CIfinishNonce := s'.grants[grantID][CIfinishNonce]
3:      let ASfinishNonce := s'.grants[grantID][ASfinishNonce]
4:      let domainAS := s'.grants[grantID][AS]
5:      let grantEndpoint := ⟨URL, S, domainAS, /requestGrant, ⟨⟩⟩
6:      let controlHash := hash(⟨CIfinishNonce, ASfinishNonce, interactRef, grantEndpoint⟩)
7:      if hash ≢ controlHash then
8:          stop
9:      let continueURL := s'.grants[grantID][continueURL]
10:     let s'.grants[grantID][finishedInteraction] := ⊤
            → Save the information that the interaction is complete
11:     let adjustInquiredValues ← {⊤, ⊥}
12:     if adjustInquiredValues ≡ ⊤ then
13:         let inquiredValues := GENERATE_INQUIRED_VALUES(⊤)
14:         let s'.grants[grantID][requested] := inquiredValues
15:     let keyRecord := s'.grants[grantID][keyRecord]
16:     if keyRecord.method ≢ mtls then
17:         let authHeader := ⟨Authorization, ⟨GNAP, s'.grants[grantID][continueAT]⟩⟩
18:         let reference := [responseTo:grantResponse, grantID:grantID]
19:         if adjustInquiredValues ≡ ⊥ then
20:             let body := [interactRef:interactRef]
21:             call SIGN_AND_SEND(POST, continueURL, keyRecord.keyID, keyRecord.key,
                 ↪ keyRecord.method, authHeader, body, reference, s', a)
22:         else
23:             let body := [interactRef:interactRef, inquiredValues:inquiredValues]
24:             call SIGN_AND_SEND(PATCH, continueURL, keyRecord.keyID, keyRecord.key,
                 ↪ keyRecord.method, authHeader, body, reference, s', a)
25:     else    → MTLS was used for grant request
26:         let s'.grants[grantID][interactRef] := interactRef
27:         if adjustInquiredValues ≡ ⊤ then
28:             let s'.grants[grantID][adjustedInquiredValues] := ⊤
29:         if keyRecord.instanceID ≢ ⊥ then
30:             let body := [instanceID:keyRecord.instanceID]
31:         else
32:             let body := [publicKey:pub(keyRecord.key)]
33:         let message := ⟨HTTPReq, ν10, POST, continueURL.host, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
34:         let reference := [responseTo:MTLS_CR, grantID:grantID]
35:         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.

1: **function** SEND_RESPONSE_TO_BROWSER($grantID$, s')
2: **if** $\mathtt{subjectID} \in s'.\mathtt{receivedValues}[grantID]$ **then**
3: **let** $subjectID := s'.\mathtt{receivedValues}[grantID][\mathtt{subjectID}]$
4: **let** $sessionID := s'.\mathtt{grants}[grantID][\mathtt{sessionID}]$
5: **let** $domainAS := s'.\mathtt{grants}[grantID][\mathtt{AS}]$
6: **let** $s'.\mathtt{sessions}[sessionID][\mathtt{loggedInAs}] := \langle domainAS, subjectID \rangle$
7: **let** $s'.\mathtt{sessions}[sessionID][\mathtt{serviceSessionID}] := \nu_{11}$
8: **let** $headers := \langle \mathtt{Set\text{-}Cookie}, \langle\langle\langle \mathtt{__Host}, \mathtt{serviceSessionID}\rangle, \langle \nu_{11}, \top, \top, \top \rangle\rangle\rangle\rangle$
9: **else**
10: **let** $headers := \langle\rangle$
11: **if** $\mathtt{domainFirstRS} \in s'.\mathtt{grants}[grantID]$ **then**
12: **let** $domainFirstRS := s'.\mathtt{grants}[grantID][\mathtt{domainFirstRS}]$
13: **let** $body := s'.\mathtt{grants}[grantID][\mathtt{resources}][domainFirstRS]$
14: **else**
15: **let** $body := \mathsf{ok}$
16: **let** $\langle key, receiver, sender, nonce \rangle := s'.\mathtt{browserRequests}[grantID][\mathtt{finishRequest}]$
17: **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, nonce, \mathtt{200}, headers, body \rangle, key)$
18: **let** $s'.\mathtt{browserRequests}[grantID] := s'.\mathtt{browserRequests}[grantID] - \mathtt{finishRequest}$
→ Remove browser request to avoid sending another response in case of another grant response in response to a continuation request
19: **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
1: **let** $switch \leftarrow \{\mathtt{start}, \mathtt{logout}, \mathtt{link}\}$ → Non-deterministically decide whether to start a grant request or to follow some link
2: **if** $switch \equiv \mathtt{start}$ **then** → Start grant request
3: **let** $url := \mathsf{GETURL}(tree, docnonce)$
4: **let** $\langle username, domain \rangle \leftarrow ids$ → Non-det. select identity to specify its domain as AS
5: **let** $url' := \langle \mathtt{URL}, \mathtt{S}, url.\mathtt{host}, \mathtt{/startGrantRequest}, \langle\rangle \rangle$
6: **let** $command := \langle \mathtt{STARTGRANT}, url', domain \rangle$
7: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$
8: **else if** $switch \equiv \mathtt{logout}$ **then** → Log out from the client instance to get a new session ID
9: **let** $url := \mathsf{GETURL}(tree, docnonce)$
10: **let** $url' := \langle \mathtt{URL}, \mathtt{S}, url.\mathtt{host}, \mathtt{/logout}, \langle\rangle \rangle$
11: **let** $command := \langle \mathtt{FORM}, url', \mathtt{POST}, \langle\rangle, \bot \rangle$
12: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$
13: **else** → Follow link
14: **let** $protocol \leftarrow \{\mathtt{P}, \mathtt{S}\}$ → Non-det. select protocol (HTTP or HTTPS)
15: **let** $host \leftarrow \mathsf{Doms}$ → Non-det. select host
16: **let** $path \leftarrow \mathbb{S}$ → Non-det. select path
17: **let** $fragment \leftarrow \mathbb{S}$ → Non-det. select fragment part
18: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$ → Non-det. select parameters
19: **let** $url := \langle \mathtt{URL}, protocol, host, path, parameters, fragment \rangle$ → Assemble URL
20: **let** $command := \langle \mathtt{HREF}, url, \bot, \bot \rangle$ → Follow link to the selected URL
21: **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 $I^{as} := \mathsf{addr}(as)$.

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

Definition 5
A *client registration record* is a term of one of the following forms

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

with $instanceID \in \mathbb{S}$, $keyID \in \mathsf{KeyIDs}$, $key \in K_{\text{KP}}$, and $publicKey \in \mathcal{T}_{\mathcal{N}}$.

instanceID is the instance identifier that the registered client instance will use. *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 client registration record *r* we will use $r.\mathtt{keyData}$ as notation for $r.2$. We will use $r.\mathtt{keyData.method}$ as notation for $r.2.1$.

Definition 6
A *user record* is a term of the form

$$\langle identity, password\rangle$$

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

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 7
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 $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $pendingRequests \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $keyMapping \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $tlskeys \in [\mathsf{Doms} \times K_{\text{TLS}}]$, $registrations \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, $users \in [\mathsf{ID} \times \mathcal{N}]$, $mtlsRequests \in [\mathcal{N} \times \mathcal{N}]$, $sigNonces \in \mathcal{T}_{\mathcal{N}}$, $grantRequests \in [\mathcal{N} \times [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]]$, and $tokenBindings \in [\mathcal{N} \times [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]]$.

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, $s_0^{as}.\mathtt{tlskeys} \equiv tlskeys^{as}$, 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

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

For each domain $d \in \mathsf{dom}(as)$ and each key record $r' \in^{\langle\rangle} s_0^{c}.\mathtt{keyRecords}[d]$ of any client instance $c \in \mathsf{CI}$, there must be a client registration record $r \in^{\langle\rangle} s_0^{as}.\mathtt{registrations}$ such that the conditions defined above are fulfilled.

$s_0^{as}.\mathtt{users}$ is a dictionary containing user records. For each user record $u \in^{\langle\rangle} s_0^{as}.\mathtt{users}$ it must hold that $u.\mathtt{identity.domain} \in \mathsf{dom}(as)$. We also require that $\forall u, u' \in^{\langle\rangle} s_0^{as}.\mathtt{users}, u \neq u' \colon u.\mathtt{identity} \not\equiv u'.\mathtt{identity}$.

For each $b \in \mathsf{B}, i \in^{\langle\rangle} s_0^{b}.\mathtt{ids}$ there must be an AS *as* that contains a user record $u \in^{\langle\rangle} s_0^{as}.\mathtt{users}$ such that $i \equiv u.\mathtt{identity}$ 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 [6]. 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 [9]. Table A.3 shows a list of all placeholders used in the algorithms.

Placeholder	Usage
ν_1	new grant ID
ν_2	new continuation access token
ν_3	new nonce to generate a unique redirect URL
ν_4	new user code
ν_5	new nonce for the calculation of the interaction finish hash
ν_6	new 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 private key
ν_{12}	new key ID
ν_{13}	new continuation access token

Table A.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 newly generated key. If the access token is bound to its own key, it is decided non-deterministically whether MTLS or signatures are used as the key proofing method. We do not allow the use of MACs here, since the newly generated symmetric key would then have to be transmitted to the resource servers for MAC validation. Since symmetric keys cannot be transmitted via token introspection, it is unclear at the time of this work how this could be implemented. The author of this work has created a GitHub issue regarding this that was still being discussed at the time this work was completed [17].
- 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.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:   if m.path ≡ /requestGrant ∧ m.method ≡ POST then    → New grant request
3:     let grantID := ν_1
4:     let s' := PERFORM_KEY_PROOF(m, grantID, s')
5:     let grantRequest := m.body
6:     let grantResponse := []
7:     let continueAT := ν_2
8:     let continueURL := ⟨URL, S, m.host, /continue, ⟨⟩⟩
9:     let s'.grantRequests[grantID] := [inquiredValues:grantRequest[inquiredValues]]
10:    if grantRequest[inquiredValues] ≡ ⟨⟩ then    → Client instance requested nothing
11:      let allowContinuation ← {⊤, ⊥}
12:      if allowContinuation ≡ ⊤ then
13:        let grantResponse[continue] := [accessToken:continueAT, url:continueURL]
14:        let s'.grantRequests[grantID][continueAT] := continueAT
15:    else if grantRequest[interaction] ≡ ⊤ then    → Interaction with RO is possible
16:      let grantResponse[continue] := [accessToken:continueAT, url:continueURL]
17:      let redirectURL := ⟨URL, S, m.host, /interact, [request:ν_3]⟩
18:      let grantResponse[interact] := [redirect:redirectURL, userCode:ν_4, finish:ν_5]
19:      let s'.grantRequests[grantID][continueAT] := continueAT
20:      let s'.grantRequests[grantID][redirectNonce] := ν_3
21:      let s'.grantRequests[grantID][userCode] := ν_4
22:      let s'.grantRequests[grantID][ASfinishNonce] := ν_5
23:      let s'.grantRequests[grantID][CIfinishNonce] := grantRequest[finish][nonce]
24:      let s'.grantRequests[grantID][finishMode] := grantRequest[finish][finishMode]
25:      let s'.grantRequests[grantID][finishURL] := grantRequest[finish][finishURL]
26:      let s'.grantRequests[grantID][grantEndpoint] := ⟨URL, S, m.host, m.path, ⟨⟩⟩
27:      if user ∈ grantRequest then    → Client instance included subject identifier
28:        let s'.grantRequests[grantID][user] := grantRequest[user]
29:    else    → No end user is present at the client instance
30:      if instanceID ∉ grantRequest then
31:        stop    → Only a registered client instance can get an access token for its resources
32:      let ⟨grantResponse, s'⟩ := CREATE_GRANT_RESPONSE(grantID, CI,
         ↪ grantRequest[inquiredValues], ⊥, m.host, s')
33:    if client ∈ grantRequest then    → Client instance is not registered
34:      let registerCI ← {⊤, ⊥}    → Non-det. decide whether to register the client instance
35:      if registerCI ≡ ⊤ then
36:        let instanceID ← 𝕊 such that instanceID ∉ s'.registrations
37:        if grantRequest[client][method] ≡ sign then
38:          let keyID := grantRequest[client][keyID]
39:          let publicKey := grantRequest[client][key]
40:          let s'.registrations[instanceID] := ⟨sign, keyID, publicKey⟩
41:        else    → grantRequest[client][method] ≡ mtls
42:          let publicKey := grantRequest[client][key]
43:          let s'.registrations[instanceID] := ⟨mtls, publicKey⟩
44:        let grantResponse[instanceID] := instanceID
45:    let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, grantResponse⟩, k)
46:    call STOP_WITH_LEAKS(f, a, m', grantResponse, s')
```

```
47:     else if m.path ≡ /continue ∧ m.method ≡ POST then
            → Continuation request after interaction completed
48:         if m.headers[Authorization].1 ≢ GNAP then
49:             stop    → Wrong Authentication scheme was used
50:         let continueAT := m.headers[Authorization].2
51:         if continueAT ≡ ⟨⟩ then
52:             stop    → Access token for continuation must be a nonce
53:         let grantID such that s'.grantRequests[grantID][continueAT] ≡ continueAT
             ↪  if possible; otherwise stop
54:         call PERFORM_KEY_PROOF(m, grantID, s')
55:         if interactRef ∉ s'.grantRequests[grantID] then
56:             stop    → An interaction reference must exist when using this endpoint
57:         if interactRef ∉ m.body then
58:             stop    → The client instance must specify an interaction reference
59:         let interactRef := m.body[interactRef]
60:         let inquiredValues := s'.grantRequests[grantID][inquiredValues]
61:         call SEND_GRANT_RESPONSE(grantID, interactRef, inquiredValues, m, k, a, f, s')
62:     else if m.path ≡ /continue ∧ m.method ≡ PATCH then    → Grant request modification
63:         if m.headers[Authorization].1 ≢ GNAP then
64:             stop    → Wrong Authentication scheme was used
65:         let continueAT := m.headers[Authorization].2
66:         if continueAT ≡ ⟨⟩ then
67:             stop    → Access token for continuation must be a nonce
68:         let grantID such that s'.grantRequests[grantID][continueAT] ≡ continueAT
             ↪  if possible; otherwise stop
69:         call PERFORM_KEY_PROOF(m, grantID, s')
70:         let inquiredValues := m.body[inquiredValues]
71:         if interactRef ∈ s'.grantRequests[grantID] then    → Interaction is not yet complete
72:             if interactRef ∉ m.body then
73:                 stop    → The interaction reference is required to finish the interaction
74:             let interactRef := m.body[interactRef]
75:             call SEND_GRANT_RESPONSE(grantID, interactRef, inquiredValues, m, k, a, f, s')
76:         else    → Interaction has already been completed or software-only authorization
77:             if interactRef ∈ m.body then
78:                 stop    → Request is not allowed to contain an interaction reference
79:             let s'.grantRequests[grantID] := s'.grantRequests[grantID] − continueAT
80:             if finishMode ∈ s'.grantRequests[grantID] then    → End user is present
81:                 let ⟨grantResponse, s'⟩ := CREATE_GRANT_RESPONSE(grantID, endUser,
                     ↪  inquiredValues, continueAT, m.host, s')
82:             else    → Software-only authorization
83:                 let ⟨grantResponse, s'⟩ := CREATE_GRANT_RESPONSE(grantID, CI,
                     ↪  inquiredValues, continueAT, m.host, s')
84:             let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, grantResponse⟩, k)
85:             call STOP_WITH_LEAKS(f, a, m', grantResponse, s')
```

```
86:   else if m.path ≡ /interact ∧ m.method ≡ GET then      → Interaction using redirect
87:     if request ∈ m.parameters then
88:       let headers := [ReferrerPolicy:origin]
89:       let request := m.parameters[request]
90:       let referrer := m.headers[Referer]
91:       let m′ := enc_s(⟨HTTPResp, m.nonce, 200, headers,
            ↪ ⟨script_as_login, [request:request, referrer:referrer]⟩⟩, k)
92:       stop ⟨⟨f, a, m′⟩⟩, s′
93:     else
94:       stop      → The parameters need to contain a request identifier
95:   else if m.path ≡ /interactUC ∧ m.method ≡ GET then      → Interaction using user code
96:     if user-code ∉ m.parameters then
97:       stop      → The parameters need to contain a user code
98:     let headers := [ReferrerPolicy:origin]
99:     let userCode := m.parameters[user-code]
100:    let grantID such that s′.grantRequests[grantID][userCode] ≡ userCode
          ↪ if possible; otherwise stop
101:    let domainCI := s′.grantRequests[grantID][finishURL].host
102:    let m′ := enc_s(⟨HTTPResp, m.nonce, 200, headers,
          ↪ ⟨script_as_login, domainCI⟩⟩, k)
103:    stop ⟨⟨f, a, m′⟩⟩, s′
104:  else if m.path ≡ /redirectLogin ∧ m.method ≡ POST
        ↪ ∧ m.headers[Origin] ≡ ⟨m.host, S⟩ then
105:    let redirectNonce := m.body[request]
106:    let grantID such that s′.grantRequests[grantID][redirectNonce] ≡ redirectNonce
          ↪ if possible; otherwise stop
107:    if s′.grantRequests[grantID][finishMode] ≡ push
          ↪ ∧ m.body[referrer].host ≢ s′.grantRequests[grantID][finishURL].host then
108:      stop      → If the push interaction finish mode is used, we verify that the browser
                      was redirected by the client instance that sent the grant request to prevent
                      the client instance mix-up attack
109:    call FINISH_INTERACTION(grantID, m, k, a, f, s′)
110:  else if m.path ≡ /userCodeLogin ∧ m.method ≡ POST
        ↪ ∧ m.headers[Origin] ≡ ⟨m.host, S⟩ then
111:    let userCode := m.body[userCode]
112:    let grantID such that s′.grantRequests[grantID][userCode] ≡ userCode
          ↪ if possible; otherwise stop
113:    call FINISH_INTERACTION(grantID, m, k, a, f, s′)
114:  else if m.path ≡ /introspect ∧ m.method ≡ POST then      → Token introspection
115:    let method := m.body[RS][method]
116:    let key := m.body[RS][key]
117:    if method ≡ sign then
118:      let keyID := m.body[RS][keyID]
119:      let s′ := VALIDATE_KEY_PROOF(method, m, keyID, key, s′)
120:    else if method ≡ mtls then
121:      let s′ := VALIDATE_KEY_PROOF(method, m, ⊥, key, s′)
122:    else
123:      stop      → Unsupported method
124:    let accessToken := m.body[accessToken]
125:    if accessToken ∉ s′.tokenBindings then
126:      let body := [active:⊥]      → Unknown access token
```

```
127:        else
128:            let body := [active:⊤]
129:            let binding := s'.tokenBindings[accessToken]
130:            let type := binding[type]
131:            let grantID := binding[grantID]
132:            let grantRequest := s'.grantRequests[grantID]
133:            if type ≡ newSign then
134:                let body[key] := [keyID:binding[keyID], key:binding[publicKey], method:sign]
135:            else if type ≡ newMTLS then
136:                let body[key] := [key:binding[publicKey], method:mtls]
137:            else if type ≡ CIKey then
138:                let method := grantRequest[method]
139:                if method ≡ sign then
140:                    let body[key] := [keyID:grantRequest[keyID], key:grantRequest[publicKey],
                    ↪ method:sign]
141:                else if method ≡ mac then    → RS must already know the key since
                                                 a symmetric key is used
142:                    let body[instanceID] := grantRequest[instanceID]
143:                else    → method ≡ mtls
144:                    let body[key] := [key:grantRequest[clientKey], method:mtls]
145:            else    → type ≡ bearer
146:                let body[flags] := bearer
147:            if binding[for] ≡ endUser then    → Access token is used to access resources of
                                                  an end user
148:                let body[access] := [identity:grantRequest[subjectID]]
149:            else    → Access token is used to access resources of a client instance
150:                let body[access] := [instanceID:grantRequest[instanceID]]
151:        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
152:        stop ⟨⟨f, a, m'⟩⟩, s'
153:    else if m.path ≡ /MTLS-prepare ∧ m.method ≡ POST then
154:        let mtlsNonce := ν_6
155:        if instanceID ∈ m.body then    → Client instance is registered
156:            let instanceID := m.body[instanceID]
157:            if instanceID ∉ s'.registrations then
158:                stop    → as does not know this instance identifier
159:            if s'.registrations[instanceID].method ≢ mtls then
160:                stop    → This client instance does not use MTLS
161:            let clientKey := s'.registrations[instanceID].publicKey
162:        else if publicKey ∈ m.body then    → Client instance is not registered
163:            let clientKey := m.body[publicKey]
164:        else
165:            stop    → Information to determine clientKey is missing
166:        let s'.mtlsRequests := s'.mtlsRequests +^⟨⟩ ⟨mtlsNonce, clientKey⟩
167:        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩,
        ↪ enc_a(⟨mtlsNonce, s'.keyMapping[m.host]⟩, clientKey)⟩, k)
168:        stop ⟨⟨f, a, m'⟩⟩, s'
169:    else
170:        stop    → Unsupported operation
```

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

```
1: function PERFORM_KEY_PROOF(m, grantID, s′)
2:     if s′.grantRequests[grantID] ≡ ⟨⟩ then    → New grant request
3:         let grantRequest := m.body
4:         if instanceID ∈ grantRequest ∧ client ∉ grantRequest then
5:             let instanceID := grantRequest[instanceID]
6:             if instanceID ∉ s′.registrations then
7:                 stop    → Instance identifier is unknown to as
8:             let s′.grantRequests[grantID][instanceID] := instanceID
9:             let keyData := s′.registrations[instanceID]
10:            let method := keyData.method
11:            if method ≡ sign then
12:                let keyID := keyData.keyID
13:                let publicKey := keyData.publicKey
14:            else if method ≡ mac then
15:                let keyID := keyData.keyID
16:                let key := keyData.key
17:            else    → MTLS
18:                let clientKey := keyData.publicKey
19:        else if instanceID ∉ grantRequest ∧ client ∈ grantRequest then
20:            let method := grantRequest[client][method]
21:            if method ≡ sign then
22:                let keyID := grantRequest[client][keyID]
23:                let publicKey := grantRequest[client][key]
24:            else if method ≡ mtls then
25:                let clientKey := grantRequest[client][key]
26:            else
27:                stop    → Invalid method or no method specified
28:        else
29:            stop    → client or instanceID must be specified, but not both
30:        let s′.grantRequests[grantID][method] := method
31:        if method ≡ sign then
32:            let s′.grantRequests[grantID][keyID] := keyID
33:            let s′.grantRequests[grantID][publicKey] := publicKey
34:        else if method ≡ mac then
35:            let s′.grantRequests[grantID][keyID] := keyID
36:            let s′.grantRequests[grantID][key] := key
37:        else    → MTLS
38:            let s′.grantRequests[grantID][clientKey] := clientKey
39:    else    → Continuation request
40:        let method := s′.grantRequests[grantID][method]
41:        if method ≡ sign then
42:            let keyID := s′.grantRequests[grantID][keyID]
43:            let publicKey := s′.grantRequests[grantID][publicKey]
44:        else if method ≡ mac then
45:            let keyID := s′.grantRequests[grantID][keyID]
46:            let key := s′.grantRequests[grantID][key]
47:        else    → MTLS
48:            let clientKey := s′.grantRequests[grantID][clientKey]
```

```
49:     if method ≡ sign then
50:         return VALIDATE_KEY_PROOF(method, m, keyID, publicKey, s′)
51:     else if method ≡ mac then
52:         return VALIDATE_KEY_PROOF(method, m, keyID, key, s′)
53:     else    → MTLS
54:         return VALIDATE_KEY_PROOF(method, m, ⊥, clientKey, s′)
```

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

```
 1: function FINISH_INTERACTION(grantID, m, k, a, f, s′)
 2:     if subjectID ∈ s′.grantRequests[grantID] then
 3:         stop    → Interaction has already been completed for this request
 4:     let identity := m.body[identity]
 5:     let password := m.body[password]
 6:     if identity ∉ s′.users then
 7:         stop    → Identity is not registered at this AS
 8:     if user ∈ s′.grantRequests[grantID] then
 9:         if identity ≢ s′.grantRequests[grantID][user] then
10:             stop    → Identity that logged in does not match identity specified in grant request
11:     if password ≢ s′.users[identity] then
12:         stop    → Incorrect password was provided
13:     let interactRef := ν7
14:     let CIfinishNonce := s′.grantRequests[grantID][CIfinishNonce]
15:     let ASfinishNonce := s′.grantRequests[grantID][ASfinishNonce]
16:     let finishMode := s′.grantRequests[grantID][finishMode]
17:     let finishURL := s′.grantRequests[grantID][finishURL]
18:     let grantEndpoint := s′.grantRequests[grantID][grantEndpoint]
19:     let hash := hash(⟨CIfinishNonce, ASfinishNonce, interactRef, grantEndpoint⟩)
20:     let s′.grantRequests[grantID][interactRef] := interactRef
21:     let s′.grantRequests[grantID][subjectID] := identity
22:     if finishMode ≡ redirect then
23:         let finishURL.parameters[interactRef] := interactRef
24:         let finishURL.parameters[hash] := hash
25:         let m′ := enc_s(⟨HTTPResp, m.nonce, 303, [Location:finishURL], ⟨⟩⟩, k)
26:         stop ⟨⟨f, a, m′⟩⟩, s′
27:     else if finishMode ≡ push then
28:         let body := [interactRef:interactRef, hash:hash]
29:         let message := ⟨HTTPReq, ν8, POST, finishURL.host, finishURL.path,
              ↪ finishURL.parameters, ⟨⟩, body⟩
30:         let s′.pendingDNS[ν9] := ⟨⊥, message⟩
            → Simulate HTTPS_SIMPLE_SEND because we have to emit two events
31:         let dataURL := ⟨URL, S, finishURL.host, /getData, finishURL.parameters⟩
32:         let m′ := enc_s(⟨HTTPResp, m.nonce, 303, [Location:dataURL], ⟨⟩⟩, k)
            → Redirect browser to client instance in order to be able to send data to browser
33:         stop ⟨⟨s′.DNSaddress, a, ⟨DNSResolve, message.host, ν9⟩⟩, ⟨f, a, m′⟩⟩, s′
34:     else
35:         stop    → Invalid interaction finish mode
```

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

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

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

```
1:  function CREATE_GRANT_RESPONSE(grantID, for, inquiredValues, oldContinueAT, host, s′)
2:      if subjectID ∈⟨⟩ inquiredValues ∧ for ≡ CI then
3:          stop    → Subject identifiers cannot be requested when using software-only authorization
4:      let grantResponse := []
5:      if accessToken ∈⟨⟩ inquiredValues then
6:          let accessToken := ν10
7:          let grantResponse[accessToken] := [value:accessToken]
8:          let bindToNewKey ← {⊤, ⊥}
9:          if bindToNewKey ≡ ⊤ then    → Access token is bound to its own key
10:             let privateKey := ν11
11:             let method ← {sign, mtls}    → Non-det. select key proofing method
12:             if method ≡ sign then
13:                 let keyID := ν12
14:                 let s′.tokenBindings[accessToken] := [grantID:grantID, for:for, type:newSign,
                    ↪ keyID:keyID, publicKey:pub(privateKey)]
15:                 let grantResponse[accessToken][key] := [method:sign, keyID:keyID,
                    ↪ privateKey:privateKey]
16:             else
17:                 let s′.tokenBindings[accessToken] := [grantID:grantID, for:for, type:newMTLS,
                    ↪ publicKey:pub(privateKey)]
18:                 let grantResponse[accessToken][key] := [method:mtls, privateKey:privateKey]
19:         else    → Access token is bound to client instances key
20:             let s′.tokenBindings[accessToken] := [grantID:grantID, for:for, type:CIKey]
21:     else if bearerToken ∈⟨⟩ inquiredValues then
22:         let bearerToken := ν10
23:         let s′.tokenBindings[bearerToken] := [grantID:grantID, for:for, type:bearer]
24:         let grantResponse[accessToken] := [value:bearerToken, flags:bearer]
25:     if subjectID ∈⟨⟩ inquiredValues then
26:         let grantResponse[subjectID] := s′.grantRequests[grantID][subjectID]
27:     let allowContinuation ← {⊤, ⊥}
28:     if allowContinuation ≡ ⊤ then
29:         let keepOldAT ← {⊤, ⊥}
30:         if keepOldAT ≡ ⊤ ∧ oldContinueAT ≢ ⊥ then
31:             let continueAT := oldContinueAT    → Continue access token does not change
32:         else
33:             let continueAT := ν13
34:         let continueURL := ⟨URL, S, host, /continue, ⟨⟩⟩
35:         let grantResponse[continue] := [accessToken:continueAT, url:continueURL]
36:         let s′.grantRequests[grantID][continueAT] := continueAT
37:     return ⟨grantResponse, s′⟩
```

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

1: **function** STOP_WITH_LEAKS(f, a, m', $grantResponse$, s')
2: **let** $events := \langle\langle f, a, m'\rangle\rangle$
3: **if** `continue` $\in grantResponse$ **then** → Leakage of continuation access token
4: **let** $events := events +^{\langle\rangle} \langle leak, a, \langle \texttt{LEAK}, grantResponse[\texttt{continue}][\texttt{accessToken}]\rangle\rangle$
5: **if** `accessToken` $\in grantResponse \wedge grantResponse[\texttt{accessToken}][\texttt{type}] \not\equiv$ `bearer` **then** → Leakage of key-bound access token
6: **let** $events := events +^{\langle\rangle} \langle leak, a, \langle \texttt{LEAK}, grantResponse[\texttt{accessToken}][\texttt{value}]\rangle\rangle$
7: **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
1: **let** $switch \leftarrow \{\texttt{login}, \texttt{link}\}$ → Non-deterministically decide whether to log in or to follow some link
2: **if** $switch \equiv$ `login` **then** → Log in to the AS
3: **let** $url :=$ GETURL($tree$, $docnonce$)
4: **if** `request` $\in scriptstate$ **then** → redirect interaction finish mode is used
5: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/redirectLogin}, \langle\rangle\rangle$
6: **let** $formData := scriptstate$ → Contains redirect request identifier + referrer
7: **else** → user code interaction start mode is used
8: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/userCodeLogin}, \langle\rangle\rangle$
9: **let** $formData := scriptinputs$ → Contains user code if client instance domain matched
10: **let** $identity \leftarrow ids$
11: **let** $secret \leftarrow secrets$
12: **let** $formData[\texttt{identity}] := identity$
13: **let** $formData[\texttt{password}] := secret$
14: **let** $command := \langle \texttt{FORM}, url', \texttt{POST}, formData, \bot\rangle$
15: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$
16: **else** → Follow link
17: **let** $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$ → Non-det. select protocol (HTTP or HTTPS)
18: **let** $host \leftarrow$ Doms → Non-det. select host
19: **let** $path \leftarrow \mathbb{S}$ → Non-det. select path
20: **let** $fragment \leftarrow \mathbb{S}$ → Non-det. select fragment part
21: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$ → Non-det. select parameters
22: **let** $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$ → Assemble URL
23: **let** $command := \langle \texttt{HREF}, url, \bot, \bot\rangle$ → Follow link to the selected URL
24: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$

A.13 Resource Servers

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

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

Definition 8
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 9
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 $s_0^{rs}.\mathtt{signingKey}$.

$s_0^{rs}.\mathtt{signingKey}$ 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 $\mathsf{ProtectedResources}$. All nonces in $\mathsf{ProtectedResources}$ that are subterms

of s_0^{rs}.clientResources must only be stored in rs initially. For each $d \in s_0^{rs}$.instanceIDs and each $i \in^{\langle\rangle} s_0^{rs}$.instanceIDs$[d]$ there must be a nonce $n \in \mathsf{ProtectedResources}$ such that s_0^{rs}.clientResources$[d][i] \equiv n$. For each $d \in s_0^{rs}$.clientResources and each $i \in s_0^{rs}$.clientResources$[d]$ it must hold that $i \in^{\langle\rangle} s_0^{rs}$.instanceIDs$[d]$. For each client instance to have its own resource, the following must apply: For all $d \in s_0^{rs}$.clientResources and all $i, i' \in s_0^{rs}$.clientResources$[d]$, $i \not\equiv i'$ we require that s_0^{rs}.clientResources$[d][i] \not\equiv s_0^{rs}$.clientResources$[d][i']$. Furthermore, for all $d, d' \in s_0^{rs}$.clientResources, $d \not\equiv d'$, all $i \in s_0^{rs}$.clientResources$[d]$, and all $i' \in s_0^{rs}$.clientResources$[d']$ it must hold that s_0^{rs}.clientResources$[d][i] \not\equiv s_0^{rs}$.clientResources$[d'][i']$.

There must not be a nonce $n \in \mathsf{ProtectedResources}$ that is a subterm of both s_0^{rs}.userResources and s_0^{rs}.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}.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 10
Given a nonce n and a domain d,

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

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 [9]. Table A.4 shows a list of all placeholders used in the algorithms.

Placeholder	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 A.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.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:   if m.path ≡ /resource ∧ m.method ≡ GET then   → Client instance wants to receive a resource
3:     let type, accessToken such that ⟨type, accessToken⟩ ≡ m.headers[Authorization]
         ↪ if possible; otherwise stop
4:     let requestID := v_1
5:     let domainAS ← s'.authServers such that is_issuer(accessToken, domainAS) ≡ ⊤
         ↪ if possible; otherwise stop
6:     let introspectionEndpoint := ⟨URL, S, domainAS, /introspect, ⟨⟩, ⊥⟩
7:     let s'.resourceRequests[requestID] := [request:m, key:k, receiver:a,
         ↪ sender:f, type:type, AS:domainAS]
8:     let method ← {sign, mtls}   → Key proofing method for introspection request
9:     if method ≡ sign then
10:      let keyEntry := [keyID:s'.signingKeyID, key:pub(s'.signingKey), method:sign]
11:      let body := [accessToken:accessToken, RS:keyEntry]
12:      let reference := [responseTo:introspection, requestID:requestID]
13:      call SIGN_AND_SEND(POST, introspectionEndpoint, s'.signingKeyID,
           ↪ s'.signingKey, ⊥, ⊥, body, reference, s', a)
14:    else
15:      let keyEntry := [key:pub(s'.mtlsKey), method:mtls]
16:      let body := [accessToken:accessToken, RS:keyEntry]
17:      let s'.resourceRequests[requestID][body] := body
18:      let request := ⟨HTTPReq, v_2, POST, domainAS, /MTLS-prepare, ⟨⟩, ⟨⟩,
           ↪ [publicKey:pub(s'.mtlsKey)]⟩
19:      let reference := [responseTo:MTLS, requestID:requestID]
20:      call HTTPS_SIMPLE_SEND(reference, request, s', a)
21:  else if m.path ≡ /MTLS-prepare ∧ m.method ≡ POST then
22:    let mtlsNonce := v_3
23:    if publicKey ∈ m.body then
24:      let clientKey := m.body[publicKey]
25:    else
26:      stop   → clientKey is missing
27:    let s'.mtlsRequests := s'.mtlsRequests +⟨⟩ ⟨mtlsNonce, clientKey⟩
28:    let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩,
         ↪ enc_a(⟨mtlsNonce, s'.keyMapping[m.host]⟩, clientKey)⟩, k)
29:    stop ⟨⟨f, a, m'⟩⟩, s'
30:  else
31:    stop   → Unsupported operation
```

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

```
1: function PROCESS_HTTPS_RESPONSE(m, reference, request, a, f, s')
2:   let requestID := reference[requestID]
3:   if reference[responseTo] ≡ introspection then
4:     let response := m.body
5:     if response[active] ≡ ⊥ then
6:       stop  → Access token is invalid
7:     else
8:       let resourceReq := s'.resourceRequests[requestID][request]
9:       let type := s'.resourceRequests[requestID][type]
10:      let domainAS := s'.resourceRequests[requestID][AS]
11:      if response[flags] ≢ bearer then  → Access token is bound to a specific key
12:        if type ≢ GNAP then
13:          stop  → Wrong authentication scheme was used
14:        if instanceID ∈ response then  → A MAC must be validated
15:          if response[instanceID] ∈ s'.symKeys[domainAS] then
16:            let ⟨keyID, key⟩ := s'.symKeys[domainAS][response[instanceID]]
17:            let s' := VALIDATE_KEY_PROOF(mac, resourceReq, keyID, key, s')
18:          else
19:            stop  → rs does not know the symmetric key
20:        else
21:          let method := response[key][method]
22:          let key := response[key][key]
23:          if method ≡ sign then
24:            let keyID := response[key][keyID]
25:            let s' := VALIDATE_KEY_PROOF(sign, resourceReq, keyID, key, s')
26:          else if method ≡ mtls then
27:            let s' := VALIDATE_KEY_PROOF(mtls, resourceReq, ⊥, key, s')
28:          else
29:            stop  → Unsupported method
30:      else  → Access token is a bearer token
31:        if type ≢ Bearer then
32:          stop  → Wrong Authentication scheme was used
33:        if key ∈ response then
34:          stop  → For a bearer token no key may be included
35:    if identity ∈ response[access] then
36:      let identity := response[access][identity]  → Identity of the RO
37:      if identity ∉⟨⟩ s'.identities[domainAS] then
38:        stop  → rs does not store resources for this RO or identity is not managed by this AS
39:      let resource := s'.userResources[identity]
40:    else if instanceID ∈ response[access] then
41:      let instanceID := response[access][instanceID]
42:      if instanceID ∉⟨⟩ s'.instanceIDs[domainAS] then  → rs does not yet store resources
                                                           for this client instance
43:        let resource := v_4
44:        let s'.instanceIDs[domainAS] := s'.instanceIDs[domainAS] +⟨⟩ instanceID
45:        let s'.clientResources[domainAS][instanceID] := resource
46:      else  → rs already stores a resource for this client instance
47:        let resource := s'.clientResources[domainAS][instanceID]
48:    else
49:      stop  → Invalid response
```

50: **let** $\mathit{responseKey} := s'.\texttt{resourceRequests}[\mathit{requestID}][\texttt{key}]$
51: **let** $\mathit{sender} := s'.\texttt{resourceRequests}[\mathit{requestID}][\texttt{sender}]$
52: **let** $\mathit{receiver} := s'.\texttt{resourceRequests}[\mathit{requestID}][\texttt{receiver}]$
53: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, \mathit{resourceReq}.\texttt{nonce}, \texttt{200}, \langle\rangle, \mathit{resource}\rangle, \mathit{responseKey})$
54: **stop** $\langle\langle \mathit{sender}, \mathit{receiver}, m'\rangle\rangle, s'$
55: **else if** $\mathit{reference}[\texttt{responseTo}] \equiv \texttt{MTLS}$ **then**
56: **let** $m_{\text{dec}} := \mathsf{dec_a}(m.\texttt{body}, s'.\texttt{mtlsKey})$
57: **let** $\mathit{mtlsNonce}, \mathit{pubKey}$ **such that** $\langle \mathit{mtlsNonce}, \mathit{pubKey}\rangle \equiv m_{\text{dec}}$ **if possible; otherwise stop**
58: **if** $\mathit{pubKey} \equiv s'.\texttt{keyMapping}[\mathit{request}.\texttt{host}]$ **then** → Send nonce only to the process that created it
59: **let** $\mathit{domainAS} := s'.\texttt{resourceRequests}[\mathit{requestID}][\texttt{AS}]$
60: **let** $\mathit{body} := s'.\texttt{resourceRequests}[\mathit{requestID}][\texttt{body}]$
61: **let** $\mathit{body}[\texttt{mtlsNonce}] := \mathit{mtlsNonce}$
62: **let** $\mathit{introspectionRequest} := \langle \texttt{HTTPReq}, \nu_5, \texttt{POST}, \mathit{domainAS}, \texttt{/introspect}, \langle\rangle, \langle\rangle, \mathit{body}\rangle$
63: **let** $\mathit{ref} := [\texttt{responseTo}{:}\texttt{introspection}, \texttt{requestID}{:}\mathit{requestID}]$
64: **call** $\mathsf{HTTPS_SIMPLE_SEND}(\mathit{ref}, \mathit{introspectionRequest}, s', a)$
65: **else**
66: **stop**

B Definitions

This appendix contains definitions that we will use to define our security properties in Appendix C and for our proofs in Appendix D. In addition to the definitions listed here, we also adopt the definitions of the formulations "emitting events" from Definition 44, "leaking a term" from Definition 46, and "knowing a term" from Definition 49 of the WIM [9].

In addition, we define the following formulation:

Definition 11 (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 $k' \in \mathcal{T}_{\mathcal{N}}$.

We will use the function $\mathsf{ownerOfResource}$ to determine the owner of a protected resource stored at an honest resource server.

Definition 12 (ownerOfResource)
Given a GNAP web system $\mathcal{GWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$, a run ρ of $\mathcal{GWS}$, a configuration (S, E, N) in ρ, an RS $rs \in \mathsf{RS}$ that is honest in S, and a nonce $n \in \mathcal{N}$ that is either a subterm of $S(rs).\mathtt{clientResources}$ or a subterm of $S(rs).\mathtt{userResources}$, $\mathsf{ownerOfResource}\colon \mathcal{N} \to \mathcal{W}$ is defined as follows:

- If $n \equiv S(rs).\mathtt{clientResources}[d][i]$ for a domain d and an instance identifier i, then $\mathsf{ownerOfResource}(n)$ depends on whether $i \in s_0^{rs}.\mathtt{clientResources}[d]$. If $i \in s_0^{rs}.\mathtt{clientResources}[d]$, $\mathsf{ownerOfResource}(n)$ is defined to be the client instance c for which there is a key record $r \in^{\langle\rangle} s_0^c.\mathtt{keyRecords}[d']$ for some $d' \in \mathsf{dom}(\mathsf{dom}^{-1}(d))$ such that $r.\mathtt{instanceID} \equiv i$. Otherwise, n must have been stored in $S(rs).\mathtt{clientResources}[d][i]$ in Line 45 of Algorithm A.21. In this case, $\mathsf{ownerOfResource}(n)$ is the process to which rs returned the newly created n in Line 54 of Algorithm A.21, i.e., $\mathsf{ownerOfResource}(n) = \mathsf{addr}^{-1}(\mathit{sender})$ for the address sender from that line.
- If $n \equiv S(rs).\mathtt{userResources}[u]$ for an identity u, $\mathsf{ownerOfResource}(n)$ is defined to be $\mathsf{ownerOfID}(u)$.

The following definition describes the process of an RO authenticating to an AS and the resulting authorization of a grant request.

Definition 13 (End user 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 authenticated to an authorization server as using an identity u in a GNAP flow identified by a nonce gid at the client instance c if there is a processing step Q in ρ with

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

(for some S, S', E, E', N, N') in which the browser b was triggered, selected a document loaded from an origin of 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 $\mathtt{request}$, let $s \equiv scriptstate[\mathtt{request}]$. Otherwise, let $s' \equiv scriptinputs[\mathtt{userCode}]$. With $grantID$ as the grant ID of as, for which $S(as).\mathtt{grantRequests}[grantID][\mathtt{redirectNonce}] \equiv s$ respectively $S(as).\mathtt{grantRequests}[grantID][\mathtt{userCode}] \equiv s'$, c is the client instance that sent the request m that led to the creation of $grantID$ in Line 3 of Algorithm A.13. gid is the grant ID of c that was created in Line 10 of Algorithm A.6 in the processing step in which m was sent by c. We then write $\mathsf{authenticated}_\rho^Q(b, c, u, as, gid)$.

C Formal Security Properties

In this appendix, we formally define our security properties for GNAP, which focus on authorization. Intuitively, authorization for $\mathcal{GWS}$ means that an attacker should not be able to obtain a protected resource that is stored at an honest RS, is protected by an honest AS, and is owned by an honest end user or an honest client instance. If the resource is owned by an honest end user, this property cannot be satisfied if the end user has granted access to its resources to a corrupted client instance.

Definition 14 (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 $s_0^c.\mathtt{keyRecords}[dmnAS']$ (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 15 (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 $n \equiv S^j(rs).\mathtt{userResources}[u]$ is derivable from the attacker's knowledge in S^j (i.e., $n \in d_{\emptyset}(S^j(na))$), it follows that

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

(2) the browser $b = \mathsf{ownerOfResource}(n)$ that owns this resource is fully corrupted in S^j, or

(3) there exist a client instance c that is honest in S^j and a key record $k \in s_0^c.\mathtt{keyRecords}[dmnAS]$ (for some domain $dmnAS \in \mathsf{dom}(\mathsf{governor}(u))$) such that $k.\mathtt{method} \equiv \mathtt{mac}$ and $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is corrupted in S^j (an honest client instance shares a symmetric key with $\mathsf{governor}(u)$ and a corrupted RS), or

(4) there exist a client instance c, a grant ID gid, and a processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$, such that $i < j$, $\mathsf{authenticated}_{\rho}^{Q}(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), gid)$, 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).

Definition 16 (Authorization Property)
Let $\mathcal{GWS}$ be a GNAP web system. We say that $\mathcal{GWS}$ *is secure w.r.t. authorization* iff $\mathcal{GWS}$ fulfills the authorization property for software-only authorization and the authorization property for end users.

D Proofs

This appendix contains the proofs of the security properties we formulated, for which we will prove various lemmas. The overall goal is to prove the authorization property.

In the proofs, we argue at various points that certain statements hold because HTTPS is used to send requests and responses between two DY processes. In doing so, we refer to the HTTPS lemmas for the browser model and the WIM's generic HTTPS server model from [9]. The lemma for the browser model is applicable because our adjustments to the browser model for the user code interaction start mode do not affect the proof of the lemma or the lemma itself. The HTTPS lemma for the WIM's generic HTTPS server model is applicable because all of our server instances (client instances, ASs, and RSs) satisfy the preconditions of the lemma. The exact properties satisfied by the WIM's modeling of HTTPS, as well as their proofs, can be found in [9].

D.1 General Properties

The following lemma was adapted from [6].

Lemma 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 $\mathsf{PROCESS_HTTPS_REQUEST}(m_{\text{dec}}, k, a, f, s')$ in Algorithm 18 of the WIM [9], then $m_{\text{dec}}.\mathtt{host} \in \mathsf{dom}(p)$, for all values of k, a, f, and s'.

Proof. For the proof we refer to [6] as it is the same. ∎

Lemma 2 (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 $r.\mathtt{method}$ or the value of $r.\mathtt{key}$ 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 $r.\mathtt{key}$ 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 $r.\mathtt{key}$ in S, there must have been a processing step in which c leaked $r.\mathtt{key}$ to another process. Whenever $r.\mathtt{key}$ 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 $r.\mathtt{key}$, $r.\mathtt{key} \notin d_\emptyset(S(p))$ must hold for all $p \in \mathcal{W} \setminus \{c\}$. ∎

Lemma 3 (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 $s_0^{rs}.\mathtt{signingKey}$ or the value of $s_0^{rs}.\mathtt{mtlsKey}$ could change. Thus, it must hold that these keys are unchanged since the initial state (i.e. $s_0^{rs}.\mathtt{signingKey} \equiv S(rs).\mathtt{signingKey} \wedge s_0^{rs}.\mathtt{mtlsKey} \equiv S(rs).\mathtt{mtlsKey}$). 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 it holds true that $s_0^{rs}.\mathtt{signingKey} \notin d_\emptyset(s_0^p) \wedge s_0^{rs}.\mathtt{mtlsKey} \notin d_\emptyset(s_0^p)$ 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 4 (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, it must hold that $r.\mathtt{key} \equiv S(c).\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key} \equiv s_0^c.\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key}$. By definition, $s_0^c.\mathtt{keyRecords}[dmn].\bar{i}.\mathtt{key}$ is only stored in c, as, and rs initially. 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 $r.\mathtt{key}$ in S, there must have been a processing step in which c, as, or rs leaked $r.\mathtt{key}$ 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 5 (Keys generated 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 grant ID $grantID$ for which $grantID \in S(c).\mathtt{receivedValues}$, every private key $k \equiv S(c).\mathtt{receivedValues}[grantID][\mathtt{accessToken}][\mathtt{key}][\mathtt{privateKey}]$ that has been stored by c in Line 26 of Algorithm A.7 in a previous state S' in response to a request sent to an AS $as \in \mathsf{AS}$ that was honest in S', and every process $p \neq c$ it holds true that $k \notin d_\emptyset(S(p))$.

Proof. When the honest client instance c stores the nonce k (a received private key) under $S'(c).\mathtt{receivedValues}[grantID][\mathtt{accessToken}][\mathtt{key}][\mathtt{privateKey}]$ in Line 26 of Algorithm A.7, this value is equal to $grantResponse[\mathtt{accessToken}][\mathtt{key}][\mathtt{privateKey}] \equiv m.\mathsf{body}[\mathtt{accessToken}][\mathtt{key}][\mathtt{privateKey}]$ (Line 8). m is the response to the request sent by c to as, which must have been an HTTPS request (since all honest processes only use HTTPS requests). Since as is honest in S', it must have included k in the response in Line 15 or Line 18 of Algorithm A.17 since only in these lines keys for an access token are returned. In both cases the value for the dictionary key $\mathtt{privateKey}$ is $privateKey \equiv \nu_{11}$ (Line 10). Since ν_{11} is a placeholder for a new nonce that is k, k cannot be derived by any other process than as at this point. Notice that as only stores $\mathsf{pub}(privateKey)$ in Line 14 or Line 17 but not the private key itself. So after as returns $privateKey \equiv k$ to c, as cannot derive it anymore. Since c used HTTPS for its request, only c is able to decrypt the response m from as containing k. Thus, it holds that $k \notin d_{\emptyset}(S'(p))$ for all $p \neq c$. $S'(c).\mathtt{receivedValues}[grantID][\mathtt{accessToken}][\mathtt{key}][\mathtt{privateKey}]$ can only be included as a subterm in an event emitted by c in Line 49 of Algorithm A.8. However, in this line the associated public key $\mathsf{pub}(privateKey) \equiv \mathsf{pub}(k)$ is included, from which the private key k cannot be derived due to the equational theory. Thus, there cannot be a processing step in which c leaks k, so $k \notin d_{\emptyset}(S(p))$ holds. ∎

Lemma 6 (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 $c \in \mathsf{CI}$ that is honest in S, every AS $as \in \mathsf{AS}$ 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 $domainAS \equiv S'(c).\mathtt{grants}[grantID][\mathtt{AS}]$ (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. Thus (and due to the use of HTTPS), the grant response processed by c must originate from as.

as only uses the $\mathtt{instanceID}$ key in a response in Line 44 of Algorithm A.13. Under the returned $instanceID$, as stores the value $publicKey$ in the $registrations$ subterm (Line 40 resp. Line 43), which is why it must hold that

$$
\begin{aligned}
& S'(as).\mathtt{registrations}[instanceID].\mathtt{publicKey} \\
\equiv\ & publicKey && \text{(L. 40/43)} \\
\equiv\ & grantRequest[\mathtt{client}][\mathtt{key}] && \text{(L. 39/42)} \\
\equiv\ & m.\mathsf{body}[\mathtt{client}][\mathtt{key}] && \text{(Line 5)}
\end{aligned}
$$

Note that Line 36 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 $S'(as).\mathtt{registrations}[instanceID] \equiv S(as).\mathtt{registrations}[instanceID]$. This means that $S(as).\mathtt{registrations}[instanceID].\mathtt{publicKey}$ 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 43 and Line 48 of Algorithm A.6. In both cases, the public key

$$\begin{aligned} & \mathsf{pub}(\mathit{key}) \\ \equiv\ & \mathsf{pub}(\mathit{keyRecord}.\mathtt{key}) && \text{(Line 40)} \\ \equiv\ & \mathsf{pub}(S''(c).\mathtt{keyRecords}[\mathit{domainAS}].\bar{i}.\mathtt{key}) && \text{(Line 26)} \end{aligned}$$

(for a previous state S'' and some $\bar{i} \in \mathbb{N}$) is transmitted in the grant request. The key record used in Line 26 is stored in $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{keyRecord}]$ in Line 27 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 45 of Algorithm A.6 resp. in Line 52 of Algorithm A.6 and then again in Line 91 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 26 of Algorithm A.6 ($S''(c).\mathtt{keyRecords}[\mathit{domainAS}].\bar{i}$), since

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

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

The following lemma and its proof are based on Lemma 7 from [6].

Lemma 7 (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 154 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 $\mathit{mtlsNonce} \notin d_{\emptyset}(S(p'))$.

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 154 of Algorithm A.13 only in Line 168, where it is asymmetrically encrypted with either the public key

$$\begin{aligned} & \mathit{clientKey} \\ \equiv\ & S(p).\mathtt{registrations}[\mathit{instanceID}].\mathtt{publicKey} && \text{(Line 161)} \\ \equiv\ & S(p).\mathtt{registrations}[m.\mathtt{body}[\mathtt{instanceID}]].\mathtt{publicKey} && \text{(Line 156)} \end{aligned}$$

or the public key $m.\mathtt{body}[\mathtt{publicKey}]$ (Line 163).

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 50 of Algorithm A.6:

$$\begin{aligned}
&\mathsf{pub}(key) \\
\equiv\ &\mathsf{pub}(keyRecord.\mathtt{key}) && \text{(Line 40)} \\
\equiv\ &\mathsf{pub}(S'(c).\mathtt{keyRecords}[domainAS].\bar{i}.\mathtt{key}) && \text{(Line 26)}
\end{aligned}$$

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:

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

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 27 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 26 of Algorithm A.6 resp. Line 5 of Algorithm A.8).

3. In Line 67 of Algorithm A.8:

$$\begin{aligned}
&\mathsf{pub}(key) \\
\equiv\ &\mathsf{pub}(keyRecord.\mathtt{key}) && \text{(Line 57)} \\
\equiv\ &\mathsf{pub}(S'(c).\mathtt{grants}[grantID][\mathtt{keyRecord}].\mathtt{key}) && \text{(Line 56)}
\end{aligned}$$

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

4. In Line 32 of Algorithm A.10:

$$\begin{aligned}
&\mathsf{pub}(keyRecord.\mathtt{key}) \\
\equiv\ &\mathsf{pub}(S'(c).\mathtt{grants}[grantID][\mathtt{keyRecord}].\mathtt{key}) && \text{(Line 15)}
\end{aligned}$$

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 $S''(c).\mathtt{keyRecords}[domainAS]$.

5. In Line 49 of Algorithm A.8:

$$\begin{aligned}
&\mathsf{pub}(privateKey) \\
\equiv\ &\mathsf{pub}(keyData[\mathtt{privateKey}]) && \text{(Line 43)} \\
\equiv\ &\mathsf{pub}(accessToken[\mathtt{key}][\mathtt{privateKey}]) && \text{(Line 41)} \\
\equiv\ &\mathsf{pub}(S'(c).\mathtt{receivedValues}[grantID][\mathtt{accessToken}][\mathtt{key}][\mathtt{privateKey}]) && \text{(Line 29)}
\end{aligned}$$

for a previous state S' and a grant ID $grantID \in \mathcal{N}$.

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

In cases 1 to 4, m.body[publicKey] equals the public key of a private key from a key record in $S'(c)$.keyRecords[$domainAS$] for a previous state S' and a domain $domainAS \in \mathsf{Doms}$. By Lemma 2, this private key can only be known to the honest process c. In case 5, due to Lemma 5, only c can know the private key associated with the public key. In case 6, the private key is the key used for MTLS by the RS c, which can only be known to c due to Lemma 3.

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 p that contains the instanceID key. Thereby the following instance identifiers are sent:

1. In Line 35 of Algorithm A.6:

$$\begin{aligned} & keyRecord.\texttt{instanceID} \\ \equiv\ & S'(c).\texttt{keyRecords}[domainAS].\bar{i}.\texttt{instanceID} && \text{(Line 26)} \end{aligned}$$

   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:

$$\begin{aligned} & instanceID \\ \equiv\ & keyRecord.\texttt{instanceID} && \text{(Line 8)} \\ \equiv\ & S'(c).\texttt{keyRecords}[domainAS].\bar{i}.\texttt{instanceID} && \text{(Line 5)} \end{aligned}$$

   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 $r \in^{\langle\rangle} S'(c)$.keyRecords[$domainAS$]. Since $domainAS$ is used as the host of c's request in both cases, it must hold that $domainAS \in \mathsf{dom}(p)$.

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

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

r.key can only be known to c due to Lemma 2.

If $r \notin^{\langle\rangle} s_0^c$.keyRecords[$domainAS$], the instance identifier r.instanceID must have been set in Line 12 of Algorithm A.7. Since $domainAS \in \mathsf{dom}(p)$ and because of Lemma 6, it must hold that $S(p)$.registrations[m.body[instanceID]].publicKey $\equiv \mathsf{pub}(r.\texttt{key})$. Again, r.key can only be known to c due to Lemma 2.

If p is a resource server, it sends an $mtlsNonce$ created in Line 22 of Algorithm A.20 only in Line 29, where it is asymmetrically encrypted with the public key m.body[publicKey] (Line 24). Thus, using the same reasoning as in the case where p is an AS, it must hold that the private key to this public key is known only to c in S.

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 84 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 52 and 37 of Algorithm A.6 and Line 18 of Algorithm A.8.

   In Line 52 and 37 of Algorithm A.6, the request goes to *domainAS*, which is stored in Line 27 in $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{AS}]$. 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 95 of Algorithm A.7:

   Here, the *mtlsNonce* is sent to $\mathit{url}.\mathtt{host} \equiv S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{continueURL}].\mathtt{host}$ (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 34 of Algorithm A.10. In this section, the request goes to $\mathit{continueURL}.\mathtt{host} \equiv S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{continueURL}].\mathtt{host}$. Here, S'' is a state before S' and *grantID* is the grant ID used in the reference.

   It must hold true that the URL $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{continueURL}]$ is equivalent to $S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{continueURL}]$ since this value 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 113 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 in $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{patchRequest}].3$ 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 URL $S''(c).\mathtt{grants}[\mathit{grantID}][\mathtt{patchRequest}]$ is equivalent to $S'(c).\mathtt{grants}[\mathit{grantID}][\mathtt{patchRequest}]$ since this 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 124 of Algorithm A.7:

   In this code section, the request into which the *mtlsNonce* is inserted is loaded from the reference in Line 132. The request must have been inserted into the reference in Line 51 of Algorithm A.8 or in Line 69 of Algorithm A.8. In both cases, the host of the inserted request is *domainRS* (Line 48 resp. Line 66 of Algorithm A.8), which is also used as the host for the request to the `/MTLS-prepare` path in Line 50 resp. Line 68. Thus, the *mtlsNonce* is sent to the same domain as the request to the `/MTLS-prepare` path.

5. Line 55 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, S'' is a state before S' and *requestID* is the request ID from the reference.

   It must hold true that the URL $S''(c).\mathtt{resourceRequests}[\mathit{requestID}][\mathtt{AS}]$ is equivalent to $S'(c).\mathtt{resourceRequests}[\mathit{requestID}][\mathtt{AS}]$ since this 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 $\mathit{mtlsNonce} \notin d_{\emptyset}(S(p'))$. ∎

Lemma 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 $\mathsf{VALIDATE_KEY_PROOF}(\mathit{method}, m, \mathit{keyID}, \mathit{key}, s')$ for some $\mathit{method} \in \{\mathtt{sign}, \mathtt{mac}, \mathtt{mtls}\}$, an HTTP request $m \in \mathsf{HTTPRequests}$, some $\mathit{keyID} \in \mathcal{N}$, some state s', and some *key* with $\mathit{key} \equiv \mathsf{pub}(S(c).\mathtt{keyRecords}[\mathit{domainAS}].\bar{i}.\mathtt{key})$ (if $\mathit{method} \in \{\mathtt{sign}, \mathtt{mtls}\}$) or $\mathit{key} \equiv S(c).\mathtt{keyRecords}[\mathit{domainAS}].\bar{i}.\mathtt{key}$ (if $\mathit{method} \equiv \mathtt{mac}$) for some process $c \in \mathsf{CI}$ that is honest in S, some $\mathit{domainAS} \in \mathsf{Doms}$, and some $\bar{i} \in \mathbb{N}$, and $\mathsf{VALIDATE_KEY_PROOF}$ returns (i.e. it does not stop), then c previously sent an HTTP request m' (using $\mathsf{HTTPS_SIMPLE_SEND}$) 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 $\mathit{method} \equiv \mathtt{mac}$ we additionally require that the AS $as = \mathsf{dom}^{-1}(\mathit{domainAS})$ and the RS $rs = \mathsf{dom}^{-1}(S(c).\mathtt{keyRecords}[\mathit{domainAS}].\bar{i}.\mathtt{rs})$ are honest in S (this is already given for as or rs if $p = as$ respectively $p = rs$).

PROOF. VALIDATE_KEY_PROOF (Algorithm A.5) only returns in Line 40. Line 40 is reached due to Line 39 only if $method \equiv \texttt{mtls}$ (Line 32) or if $method \in \{\texttt{sign}, \texttt{mac}\}$ (Line 2).

For the algorithm to return in the former case, it must hold that there exists an $mtlsInfo \in^{\langle\rangle} S(p).\texttt{mtlsRequests}$ such that $mtlsInfo.1 \equiv m.\mathsf{body}[\texttt{mtlsNonce}] \wedge mtlsInfo.2 \equiv key$. This *mtlsInfo* must have been stored in Line 166 of Algorithm A.13 (in case of an AS) or in Line 27 of Algorithm A.20 (in case of an RS). In both cases, the $mtlsNonce \equiv mtlsInfo.1 \equiv m.\mathsf{body}[\texttt{mtlsNonce}]$ is sent only encrypted with the $clientKey \equiv mtlsInfo.2 \equiv key$. Thus, due to Lemma 2, only c can decrypt the *mtlsNonce*, since only c can know the matching private key. Since $mtlsNonce \equiv m.\mathsf{body}[\texttt{mtlsNonce}]$, it must hold for the sender p' of m that $mtlsNonce \in d_{\emptyset}(S(p'))$. Thus, due to Lemma 7, p' must be c or *as*. Since the honest AS *as* does not send HTTP requests, $p' = c$ must hold. Thus, the message m must have been sent by c, which means that in the context of this lemma, the stronger statement that $m \equiv m'$ holds for the MTLS case.

In the case that $method \in \{\texttt{sign}, \texttt{mac}\}$, it must hold that $\mathsf{checksig}(m.\texttt{headers}[\texttt{signature}], key) \equiv \top$ (if $method \equiv \texttt{sign}$) or that $\mathsf{checkmac}(m.\texttt{headers}[\texttt{signature}], key) \equiv \top$ (if $method \equiv \texttt{mac}$). It also must hold that $controlInput \equiv \mathsf{extractmsg}(m.\texttt{headers}[\texttt{signature}])$. If we have that $\mathsf{checksig}(m.\texttt{headers}[\texttt{signature}], key) \equiv \top$, the signature must have been created by c due to Lemma 2. If $\mathsf{checkmac}(m.\texttt{headers}[\texttt{signature}], key) \equiv \top$, the MAC must have been created by c due to Lemma 4 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 (SIGN_AND_SEND) and $controlInput \equiv \mathsf{extractmsg}(m.\texttt{headers}[\texttt{signature}])$, c must have sent an HTTP request m' using SIGN_AND_SEND for which the following holds true:

1. $m'.\texttt{method} \equiv m.\texttt{method}$, since the HTTP method is always covered by the *controlInput* (Line 17 of Algorithm A.5),
2. $m'.\mathsf{body} \equiv m.\mathsf{body}$, since if $m.\mathsf{body} \not\equiv \langle\rangle$ the body is covered by the *controlInput* using a hash of it (Line 19 of Algorithm A.5) and if $m.\mathsf{body} \equiv \langle\rangle$ it must hold that $m'.\mathsf{body} \equiv \langle\rangle$ 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 be equivalent to $\mathsf{extractmsg}(m.\texttt{headers}[\texttt{signature}])$,
3. $m'.\texttt{host} \equiv m.\texttt{host}$, since the host is always covered by the *controlInput* via the *controlURL* (Line 16 of Algorithm A.5),
4. $m'.\texttt{path} \equiv m.\texttt{path}$, since the path is always covered by the *controlInput* via the *controlURL* (Line 16 of Algorithm A.5),
5. $m'.\texttt{parameters} \equiv m.\texttt{parameters}$, since the parameters are always covered by the *controlinput* via the *controlURL* (Line 16 of Algorithm A.5), and
6. if $\texttt{Authorization} \in m.\texttt{headers}$, it must hold that $m'.\texttt{headers}[\texttt{Authorization}] \equiv m.\texttt{headers}[\texttt{Authorization}]$ since when using the 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. ∎

Lemma 9 (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 $\mathsf{PERFORM_KEY_PROOF}(m, grantID, s')$ (for some m, $grantID$, s') in Line 54 of Algorithm A.13 or in Line 69 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 $r.\mathtt{method} \equiv \mathtt{mac}$ for this grant request, we additionally require that $rs = \mathsf{dom}^{-1}(r.\mathtt{rs})$ 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 68 respectively, that $grantID \in S(as).\mathtt{grantRequests}$. $grantID$ must have been stored in $S'(as).\mathtt{grantRequests}$ 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 $S'(as).\mathtt{grantRequests}[grantID] \equiv \langle\rangle$, 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 30-38 of Algorithm A.14 in $S'(as).\mathtt{grantRequests}[grantID]$. Since this is the only code section where these subterms are written to and this write operation implies that the condition $S(as).\mathtt{grantRequests}[grantID] \equiv \langle\rangle$ 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 $S(as).\mathtt{grantRequests}[grantID]$ are still the same.

In the calls to PERFORM_KEY_PROOF in Line 54 of Algorithm A.13 and in Line 69 of Algorithm A.13 it thus must still hold that $S(as).\mathtt{grantRequests}[grantID] \not\equiv \langle\rangle$, so now the previously stored key proofing method and the used key are loaded again from $S(as).\mathtt{grantRequests}[grantID]$ in Lines 40-48 of Algorithm A.14. So PERFORM_KEY_PROOF calls VALIDATE_KEY_PROOF in Lines 50-54 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 8 that m was sent by c.[1] 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 the request is detected in Line 15, since the replayed request must contain the same nonce in *sigParams* as the original request, this nonce was already stored in Line 31 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 10 (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 $accessToken \in S(as).\mathtt{tokenBindings}$ with $S(as).\mathtt{tokenBindings}[accessToken][\mathtt{type}] \equiv \mathtt{bearer}$ it holds true that $accessToken \notin d_{\emptyset}(S(p))$ for any process p as long as the following conditions hold true:

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

(1) the client instance c that sent the grant request that led to the creation of the grant ID $grantID \equiv S(as).\mathtt{tokenBindings}[accessToken][\mathtt{grantID}]$ in Line 3 of Algorithm A.13 is honest in S,

(2) if c used a key record r with $r.\mathtt{method} \equiv \mathtt{mac}$ for this grant request, it holds true that $\mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S,

(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, and

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

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 37 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 46: grant response in response to a continuation request
2. Line 85: adjustment of the requested values via a PATCH request
3. Line 75 (CREATE_GRANT_RESPONSE is called via Algorithm A.16): adjustment of the requested values via a PATCH request including interaction finish
4. Line 61 (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 9 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 $S(as).\mathtt{tokenBindings}[accessToken][\mathtt{type}] \equiv \mathtt{bearer}$, *as* must have set the value of $grantResponse[\mathtt{accessToken}][\mathtt{flags}]$ to $\mathtt{bearer}$ in Line 24 of Algorithm A.17 (since $S(as).\mathtt{tokenBindings}[accessToken][\mathtt{type}]$ 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 $S'(c).\mathtt{grants}[grantID][\mathtt{bearerRSs}]$ in Line 33 for some state S'. Since a domain can never be removed from $S'(c).\mathtt{grants}[grantID][\mathtt{bearerRSs}]$, it must hold that c sends *accessToken* only to RSs in $\left\{rs \in \mathsf{RS} \,\middle|\, \exists dmnRS \in^{\langle\rangle} S(c).\mathtt{grants}[grantID][\mathtt{bearerRSs}]\colon dmnRS \in \mathsf{dom}(rs)\right\}$ (using HTTPS).

Since all RSs 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 129 of Algorithm A.13 and then discards it.

Thus, in total, only processes in

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

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. ∎

D.2 Authorization Property for Software-only Authorization

Lemma 11 (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 $dmn \in S(rs).\mathtt{clientResources}$, and every $instanceID \in S(rs).\mathtt{clientResources}[dmn]$ it holds true that if $S(rs).\mathtt{clientResources}[dmn][instanceID]$ is included as *resource* in the response m' in Line 53 of Algorithm A.21 by *rs*, then m' is a response to an HTTP request m sent by $c = \mathsf{ownerOfResource}(resource)$ 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 $dmnAS \in s_0^c.\mathtt{keyRecords}$ and all key records $r \in s_0^c.\mathtt{keyRecords}[dmnAS]$ with $r.\mathtt{method} \equiv \mathtt{mac}$ it holds true that $\mathsf{dom}^{-1}(r.\mathtt{rs})$ is honest in S, and

(4) for every grant ID gid in $S(c).\mathtt{grants}$ for which it holds that $\mathtt{sessionID} \notin S(c).\mathtt{grants}[gid]$ (software-only authorization is used) and that $S(c).\mathtt{grants}[gid][\mathtt{AS}] \in \mathsf{dom}(\mathsf{dom}^{-1}(dmn))$, and every $dmnRS \in S(c).\mathtt{grants}[gid][\mathtt{bearerRSs}]$ it holds true that $\mathsf{dom}^{-1}(dmnRS)$ is honest in S.

Proof. If $resource \equiv S(rs).\mathtt{clientResources}[dmn][instanceID]$ in Line 53 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 $[\mathtt{access}][\mathtt{instanceID}]$ (Line 41). If Line 53 of Algorithm A.21 is executed, it must hold that in the introspection response the bearer flag has been set (Line 30) 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 17, 25 or 27 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 146 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 145 `bearer` is the only remaining type used). The sender of m must know the bearer token $\mathit{accessToken}$ in S, otherwise it could not have included it inside the `Authorization` header. By Lemma 10, 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 131 of Algorithm A.13 and then used in Line 132 to load the grant request. The stored instance identifier is then loaded from the grant request in Line 150, which is then sent to rs under $[\texttt{access}][\texttt{instanceID}]$, so this is the $\mathit{instanceID}$ used by rs. This instance identifier must have been stored in Line 8 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 $\mathit{instanceID}$ 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 $\mathsf{ownerOfResource}(\mathit{resource})$.

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 17: In this line, VALIDATE_KEY_PROOF validates a MAC. The key for this key proof is loaded by rs in Line 16 from the $\mathit{symKeys}$ subterm. The instance identifier used was returned by as in the introspection response in the $[\texttt{instanceID}]$ entry (using HTTPS). This must be the same instance identifier that was returned by as under $[\texttt{access}][\texttt{instanceID}]$ ($\mathit{instanceID}$), since as uses $\mathit{grantRequest}[\texttt{instanceID}]$ for both (Line 142 and Line 150 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 8, that m was sent by $\mathsf{ownerOfResource}(\mathit{resource})$ (headers other than the `Authorization` header may differ, but are irrelevant).
- Line 25: In this line, VALIDATE_KEY_PROOF validates a signature. The key for this key proof is returned by as in the introspection response under $[\texttt{key}][\texttt{key}]$ (Line 22). This can be either the key used by the client instance in the grant request (returned by as in Line 140 of Algorithm A.13 whereby the returned value was stored in Line 33 of Algorithm A.14) or a key generated by AS only for binding to this access token (returned in Line 134 of Algorithm A.13). If the key is the key from the grant request, this must be the key associated with $\mathit{instanceID}$, and since VALIDATE_KEY_PROOF returned using this key, it must hold according to Lemma 8 that m was sent by $\mathsf{ownerOfResource}(\mathit{resource})$ (again, headers other than the `Authorization` header may differ, but are irrelevant). If the key is one generated by as for this access token, it must have been loaded in Line 134 of Algorithm A.13 from $S'(as).\texttt{tokenBindings}[\mathit{accessToken}][\texttt{publicKey}]$ for a previous state S'. By Lemma 5, only the client instance to which as sent the associated private key after generating it knows that private key. Since VALIDATE_KEY_PROOF returned, the signature validation in Line 26 of the algorithm must have been successful. As seen in the proof of Lemma 8, m must thus have been sent by this client instance (again, headers other than the `Authorization` header may

differ, but are irrelevant). This client instance must be $\mathsf{ownerOfResource}(\mathit{resource})$, since the grant response in which the client instance received the private key from *as* is sent by *as* only after calling VALIDATE_KEY_PROOF with the client instance's public key associated with *instanceID* (since, as with the bearer tokens, *instanceID* was stored during the call to PERFORM_KEY_PROOF when the grant request was processed and exactly this instance identifier is returned to *rs* in the introspection response under $[\mathtt{access}][\mathtt{instanceID}]$).

- Line 27: 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. ∎

Lemma 12 (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 $\mathit{dmnAS} \in S(rs).\mathtt{clientResources}$, and every instance identifier $i \in S(rs).\mathtt{clientResources}[\mathit{dmnAS}]$ it holds true that if $n \equiv S(rs).\mathtt{clientResources}[\mathit{dmnAS}][i]$ 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 $s_0^c.\mathtt{keyRecords}[\mathit{dmnAS}']$ (for some domain $\mathit{dmnAS}' \in \mathsf{dom}(\mathsf{dom}^{-1}(\mathit{dmnAS}))$) 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 $y \in^{\langle\rangle} S(c).\mathtt{grants}[\mathit{gid}][\mathtt{bearerRSs}]$ such that $\mathtt{sessionID} \notin S(c).\mathtt{grants}[\mathit{gid}]$ (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) $\mathsf{dom}^{-1}(\mathit{dmnAS})$ is honest in S,

(2) $c = \mathsf{ownerOfResource}(n)$ is honest in S,

(3) for all domains $\mathit{dmnAS}' \in \mathsf{dom}(\mathsf{dom}^{-1}(\mathit{dmnAS}))$ with $\mathit{dmnAS}' \in s_0^c.\mathtt{keyRecords}$ and all key records $k \in s_0^c.\mathtt{keyRecords}[\mathit{dmnAS}']$ 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 $y \in^{\langle\rangle} S(c).\mathtt{grants}[\mathit{gid}][\mathtt{bearerRSs}]$ such that $\mathtt{sessionID} \notin S(c).\mathtt{grants}[\mathit{gid}]$ 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 43 of Algorithm A.21 and then stored in the state of *rs* in Line 45 (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 53 of Algorithm A.21. Since all the conditions for Lemma 11 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. ∎

D.3 Authorization Property for End Users

Lemma 13 (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 [8]. 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 11 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.

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 11 of Algorithm A.15 and then discards it. $\mathsf{ownerOfID}(u)$ is then redirected to the client instance by *as* in Line 25 or Line 32. 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 $p \notin \{as, \mathsf{ownerOfID}(u)\}$, which proves the lemma. ∎

Lemma 14 (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 $rs \in \mathsf{RS}$ that is honest in S, and every identity $u \in S(rs).\mathtt{userResources}$ it holds true that if $S(rs).\mathtt{userResources}[u]$ is included as *resource* in the response m' in Line 53 of Algorithm A.21 by *rs*, then m' is a response to an HTTP request m for which the following holds true: with $as = \mathsf{governor}(u)$ and $gid \equiv S(as).\mathtt{tokenBindings}[m.\mathtt{headers}[\mathtt{Authorization}].2][\mathtt{grantID}]$ we have that $S(as).\mathtt{grantRequests}[gid][\mathtt{subjectID}] \equiv u$ as long as *as* is honest in S.

Proof. If $S(rs).\mathtt{userResources}[u]$ is included as *resource* in the response m' in Line 53 of Algorithm A.21 by *rs*, it must hold that *resource* was loaded in Line 39, since this is the only line where *rs* loads a user resource. The identity u used in this line is retrieved from

$response$[access][identity] (Line 36), where $response$ is the introspection response received by rs. The corresponding introspection request must have been sent by rs in Line 13 of Algorithm A.20 or Line 64 of Algorithm A.21, since these are the only lines where introspection requests are sent. This introspection request must have been sent to as, otherwise the check in Line 37 of Algorithm A.21 would have been successful (the $identities$ subterm cannot change) and therefore Line 39 would not have been executed. Since as is honest by precondition, as must have included the identity u in the introspection response in Line 148 of Algorithm A.13. The value used here for u is $S'(as)$.grantRequests[$grantID$][subjectID], where $grantID \equiv S'(as)$.tokenBindings[$accessToken$][grantID] (Line 131) for some previous state S' and the $accessToken$ that was transmitted to as by rs. As seen in Line 11 resp. Line 16 and Line 3 of Algorithm A.20, this access token passed from rs to as is taken from m.headers[Authorization].2. This proves the lemma, since the values of $S'(as)$.grantRequests[$grantID$][subjectID] and $S'(as)$.tokenBindings[$accessToken$][grantID] cannot be overwritten and thus must still be the same in S. ■

Lemma 15 (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 $u \in S^j(rs)$.userResources it holds true that if $S^j(rs)$.userResources[u] is included as $resource$ in the response m' in Line 53 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 processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$, such that $i < j$ and $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), gid)$ (with gid being the grant ID of the grant in whose context c sent m) as long as

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

(2) $\mathsf{ownerOfResource}(resource)$ is not fully corrupted in S^j,

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

(4) there do not exist c', gid', and $Q' = (S^{i'}, E^{i'}, N^{i'}) \rightarrow (S^{i'+1}, E^{i'+1}, N^{i'+1})$, such that $i' < j$, $\mathsf{authenticated}_\rho^{Q'}(\mathsf{ownerOfID}(u), c', u, \mathsf{governor}(u), gid')$, and
   (a) c' is corrupted in S^j, or
   (b) there exists a domain $y \in^{\langle\rangle} S^j(c')$.grants[$gid'$][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 $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), gid)$ holds for some previous processing step Q and some grant ID gid. (III) ensures that gid is the grant ID of the grant in whose context c sent m.[2] If $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), gid)$ 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 $as = \mathsf{governor}(u)$.

We start with showing (I). By Lemma 14, m must have contained an access token *accessToken* in the Authorization header, such that $S^j(as).\mathtt{grantRequests}[grantID][\mathtt{subjectID}] \equiv u$ with $grantID \equiv S^j(as).\mathtt{tokenBindings}[accessToken][\mathtt{grantID}]$. The [subjectID] entry must have been written in Line 21 of Algorithm A.15 since this is the only line where this entry is written. Since u is written in Line 21, it must have hold in Line 11 that $password \equiv s_0^c.\mathtt{users}[u] \equiv \mathsf{secretOfID}(u)$ (the entries of the *users* subterm of an honest AS never change). The value of *password* is taken from the parameter m (Line 5). m must be a request received by as at the /redirectLogin path or at the /userCodeLogin path, since only in these sections Algorithm A.15 is called. According to Lemma 13, 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 /redirectLogin path or the /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 21 of Algorithm A.15, and u was taken from the [identity] subterm of m (Line 4), 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 14, we know that the request m must contain an access token *accessToken* in the Authorization header, such that for the grant ID $grantID \equiv S^j(as).\mathtt{tokenBindings}[accessToken][\mathtt{grantID}]$, it holds true that $S^j(as).\mathtt{grantRequests}[grantID][\mathtt{subjectID}] \equiv u$. 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 11, 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 23 of Algorithm A.17 in $S^{i'}(as).\mathtt{tokenBindings}$ for some previous state $S^{i'}$. Since the entries for an access token in the *tokenBindings* subterm cannot change, it must hold that $S^j(as).\mathtt{tokenBindings}[accessToken][\mathtt{type}] \equiv \mathtt{bearer}$. By Lemma 10 it must thus hold that the bearer token is derivable in S^j only for as, the client instance that sent the grant request to as, 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 request), c must be the client instance that sent the grant request to $as = \mathsf{governor}(u)$.

[2]This is required to rule out cuckoos token attack, for example.

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 one of the following lines of Algorithm A.21:

- Line 17: In this line, VALIDATE_KEY_PROOF validates a MAC. The key for this key proof is loaded by rs in Line 16 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 142 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 8 of Algorithm A.14 when as processed the grant request. So when rs loads the symmetric key from the $symKeys$ subterm in Line 16 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 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 8 can be applied in this proof because by precondition (4) $\mathsf{ownerOfID}(u)$ 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 25: 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 22). This can be either the key used by the client instance in the grant request (returned by as in Line 140 of Algorithm A.13 whereby the returned value was stored in Line 33 of Algorithm A.14) or a key generated by AS only for binding to this access token (returned in Line 134 of Algorithm A.13). If the key is the key from the grant request, it must hold by Lemma 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 one generated by as for this access token, it must have been loaded in Line 134 of Algorithm A.13 from $S^{i'}(as)$.tokenBindings[$accessToken$][publicKey] for a previous state $S^{i'}$. By Lemma 5, only the client instance to which as sent the associated private key after generating it can be able to derive that private key. Since VALIDATE_KEY_PROOF returned, the signature validation in Line 26 of the algorithm must have been successful. As seen in the proof of Lemma 8, m must thus have been sent by this client instance (again, headers other than the Authorization header may differ, but are irrelevant). We will now show that the client instance to which the private key was sent after its generation must also be the client instance that sent the grant request to as. New private keys for access tokens are generated in Line 10 of Algorithm A.17 (CREATE_GRANT_RESPONSE). When CREATE_GRANT_RESPONSE was called, the second parameter must have been endUser in this case, since a resource of an end user is returned. CREATE_GRANT_RESPONSE is called only as a result of a request to the /continue path of as with the endUser parameter. According to Lemma 9, it must hold that this request was sent by the same client instance as the request that led to the creation of the grant ID, i.e. the grant request. The client instance that sent m must therefore also be the client instance that sent the grant request.
- Line 27: 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 26 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 cuckoo token 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 22 of Algorithm A.17). If c received an access token issued by as from a corrupted AS 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.

Due to Lemma 10, a corrupted AS cannot send a bearer token created by as to c, so we only have to look at key-bound access tokens.

We assume that c received an access token that was issued by as from a corrupted AS and included this access token in the `Authorization` header of m. If the access token is bound to a new key generated by as, the corrupted AS from which c received the access token is unable to transmit the associated private key to c due to Lemma 5. However, since c, when using the access token in Line 43 of Algorithm A.8, uses the private key received with the access token, which is stored in Line 26 of Algorithm A.7, c will not use the private key bound to the access token to create the signature in Line 46 of Algorithm A.8. Nevertheless, in the token introspection as will return the public key of the private key it created for the access token since the introspection request is sent to the AS for which the is_issuer function returns $\top$ (Line 5 of Algorithm A.20). is_issuer will only return $\top$ when given a domain of as as input, since corrupted ASs do not store leaked access tokens in the *tokenBindings* subterm. Thus, when rs validates the signature in Line 25 of Algorithm A.21 the signature validation must fail, since the public key used by rs to validate the signature cannot be the public key of the private key that was used by c to create the signature. Therefore, VALIDATE_KEY_PROOF will not return in this line and rs will not send m'.

A similar situation occurs when the access token in m is bound to the key that c used for the grant request. 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 140 resp. Line 142 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 17 or Line 25 of Algorithm A.21 and rs will not send m'. The same reasoning can be applied to the use of MTLS and Line 27 of Algorithm A.21.

Thus, (III) must hold for all types of access tokens, and so must Lemma 15. ■

Lemma 16 (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 80 of Algorithm A.6 c stores $\langle k, a, f, m.\mathtt{nonce}\rangle$ under $S(c).\mathtt{browserRequests}[\mathit{grantID}][\mathtt{finishRequest}]$ (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 10 of Algorithm A.6 as long as b is not fully corrupted in S.

Proof. If c executes Line 61 or Line 80 of Algorithm A.6, it must hold that

$$m.\mathtt{headers}[\mathtt{Cookie}][\langle\mathtt{__Host}, \mathtt{sessionID}\rangle] \equiv S(c).\mathtt{grants}[\mathit{grantID}][\mathtt{sessionID}]$$

due to the check in Line 59 resp. Line 78. 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 $\langle\mathtt{__Host}, \mathtt{sessionID}\rangle$ 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 90 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 $\langle\mathtt{__Host}, \mathtt{sessionID}\rangle$ 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, $S(c).\mathtt{grants}[\mathit{grantID}][\mathtt{sessionID}]$, must have been stored during a call to the `/startGrantRequest` path of c in Line 27 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 10 of Algorithm A.6. The value stored in Line 27 of Algorithm A.6 is either a session ID that was transferred by the process that called the `/startGrantRequest` path (Line 21) or a new session ID generated by c for this session (Line 25) 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 $\langle\mathtt{__Host}, \mathtt{sessionID}\rangle$ cookie. As seen, only b and c are able to derive this value in S, so if Line 61 or Line 80 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 17 (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 $c \in \mathsf{CI}$ that is honest in S^j, every grant ID $\mathit{grantID} \in S^j(c).\mathtt{grants}$, and every identity $u \in \mathsf{ID}$ it holds true that if $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID})$ (for some previous processing step $Q = (S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ and some integer $i < j$) and c calls $\mathsf{SEND_CONTINUATION_REQUEST}(\mathit{grantID}, \mathit{interactRef}, \mathit{hash}, s', a)$ (for some *interactRef*, *hash*, s', a) in the processing step $(S^j, E^j, N^j) \to (S^{j+1}, E^{j+1}, N^{j+1})$, then $\mathsf{ownerOfID}(u)$ must have sent the request to the `/startGrantRequest` path of c that led to the creation of *grantID* in Line 10 of Algorithm A.6 as long as $\mathsf{ownerOfID}(u)$ is not fully corrupted in S^j and $\mathsf{governor}(u)$ is honest in S^j.

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

Redirect Interaction Start Mode + Redirect Interaction Finish Mode We have given that $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID})$ and c has thereupon called $\mathsf{SEND_CONTINUATION_REQUEST}(\mathit{grantID}, \mathit{interactRef}, \mathit{hash}, s', a)$. 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 54). We will now show that this call must have been made by $\mathsf{ownerOfID}(u)$. In Line 56, the *grantID* is determined using the *finishURLnonce* from the parameters of the request. The *finishURLnonce* is created by c in Line 14 of Algorithm A.6 and then transferred to the AS within the grant request using HTTPS. Since the AS $\mathsf{governor}(u)$ is honest by precondition, $\mathsf{governor}(u)$ must have forwarded $\mathsf{ownerOfID}(u)$ to the finish URL with the *finishURLnonce* of c immediately after the authentication in processing step Q, which happens in Line 25 of Algorithm A.15. Since c and $\mathsf{governor}(u)$ are honest and the *finishURLnonce* is not otherwise used or sent, the call to the `/finish` path must thus have been made by $\mathsf{ownerOfID}(u)$, 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 16, 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 $\mathsf{ownerOfID}(u)$.

Redirect Interaction Start Mode + Push Interaction Finish Mode Since the push interaction finish mode is used, the call to SEND_CONTINUATION_REQUEST must have occurred in Line 72 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 14 of Algorithm A.6) and only transmits it within the grant request. This AS must be $\mathsf{governor}(u)$, since $\mathsf{ownerOfID}(u)$ would otherwise not have authenticated to the AS in processing step Q due to our browser model. The request to the `/push` path made by $\mathsf{governor}(u)$, which is honest by precondition, is sent in Line 29 of Algorithm A.15 (FINISH_INTERACTION). FINISH_INTERACTION must have been called in Line 109 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 $\mathsf{governor}(u)$. Since $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID})$ holds, this request to the `/redirectLogin` path of $\mathsf{governor}(u)$ must have been sent by $\mathsf{ownerOfID}(u)$ in processing step Q. When a request to the `/redirectLogin` path is received, Line 107 ensures that $\mathsf{ownerOfID}(u)$ 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 91 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 14 of Algorithm A.6 using Lemma 1. Thus, $\mathsf{ownerOfID}(u)$ must have been forwarded by c in the context of this grant (which is uniquely identified by the *redirectNonce* in Line 106 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 19 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 10. Thus, the browser forwarded to $\mathsf{governor}(u)$ must thus have sent this request. Since the forwarded browser is $\mathsf{ownerOfID}(u)$, $\mathsf{ownerOfID}(u)$ must have sent the request to the `/startGrantRequest` path of c that led to the creation of *grantID* in Line 10.

User Code Interaction Start Mode In the case of the user code interaction start mode, the proof is independent of the chosen interaction finish mode. Since $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), \mathit{grantID})$ 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 $\mathsf{governor}(u)$ (Line 14). $\mathsf{governor}(u)$ retrieves the user code *uc* from the request to the `/userCodeLogin` path in Line 111 of Algorithm A.13. Since the interaction completed successfully, *uc* must be the user code that $\mathsf{governor}(u)$ generated in Line 21 of Algorithm A.13 upon receiving the grant request. Otherwise, Algorithm A.13 would have stopped in Line 112 and the interaction would not have been finished.

The user code must have been included in the *scriptinputs* in Line 49 of Algorithm A.2 since only in this line such an entry can be added to the *scriptinputs*. Due to Line 48 this happens only if $\mathit{domainCI} \equiv \mathit{domainUsedCI}$. *domainCI* is the domain of the client instance that sent the grant request associated with *uc* to $\mathsf{governor}(u)$. This value was returned by $\mathsf{governor}(u)$ along with `script_as_login` in Line 102 of Algorithm A.13. *domainUsedCI* is the domain of the client instance to which $\mathsf{ownerOfID}(u)$ 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 39 of Algorithm A.2 in the *usedCIs* subterm under the key *uc*. Since *domainCI* must be equivalent to *domainUsedCI*, the client instance to which $\mathsf{ownerOfID}(u)$ sent the request to the `/startGrantRequest` path must be the client instance that sent the grant request associated with *uc* to $\mathsf{governor}(u)$, 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 19 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 $\mathsf{ownerOfID}(u)$ has received *uc*, $\mathsf{ownerOfID}(u)$ thus must have sent the request to the `/startGrantRequest` path of c. Hence, $\mathsf{ownerOfID}(u)$ must be responsible for the creation of *grantID* in Line 10 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. ■

Lemma 18 (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 $c \in \mathsf{CI}$ that is honest in S it holds true that if the client instance c stores $m.\mathsf{body}$ under $S(c).\mathtt{grants}[grantID][\mathtt{resources}][domainRS]$ in Line 79 of Algorithm A.7 (for some m, $grantID$, $domainRS$) and thereby $m.\mathsf{body} \equiv s_0^{rs}.\mathtt{userResources}[u]$ for $rs = \mathsf{dom}^{-1}(domainRS)$ and some identity u, then $\mathsf{ownerOfResource}(m.\mathsf{body})$ must have sent the request to the `/startGrantRequest` path of c that led to the creation of $grantID$ in Line 10 of Algorithm A.6 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 and all key records $k \in s_0^{c'}.\mathtt{keyRecords}[dmnAS]$ (for any $dmnAS \in \mathsf{dom}(\mathsf{governor}(u))$) it holds true that $k.\mathtt{method} \not\equiv \mathtt{mac}$ or $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is honest in S.

Proof. In Line 79 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 78. 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 54 of Algorithm A.21, since only in this line matching responses are sent by an honest RS. Since m contains $s_0^{rs}.\mathtt{userResources}[u]$, according to Lemma 15, it must hold that there is a processing step Q such that $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), grantID)$.[3]

c can only store a resource in $S(c).\mathtt{grants}[grantID][\mathtt{resources}][domainRS]$ in Line 79 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 $\mathsf{SEND_CONTINUATION_REQUEST}(grantID, interactRef, hash, s', a)$ (for some $interactRef$, $hash$, s', a) in a processing step after Q in order to finish the interaction. Using Lemma 17, 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 10 of Algorithm A.6. Since by definition $\mathsf{ownerOfResource}(m.\mathsf{body}) = \mathsf{ownerOfID}(u)$, the lemma is shown. ∎

Lemma 19 (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 19 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,

[3] Since c is honest in S, we can ignore precondition (4) of Lemma 15, since an honest client instance will not use leaked bearer tokens or the keys of other corrupted client instances.

(2) the RS $rs = \mathsf{dom}^{-1}(S(c).\mathtt{grants}[\mathit{grantID}][\mathtt{domainFirstRS}])$ (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 $s_0^{rs}.\mathtt{userResources}[u] \equiv n$, it holds true that $\mathsf{governor}(u)$ is honest in S, and

(4) for all client instances c' that are honest in S and all key records $k \in s_0^{c'}.\mathtt{keyRecords}[\mathit{dmnAS}]$ (for any $\mathit{dmnAS} \in \mathsf{dom}(\mathsf{governor}(u))$) it holds true that $k.\mathtt{method} \not\equiv \mathtt{mac}$ or $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is honest in S.

Proof. In Line 19 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 $S(c).\mathtt{browserRequests}[\mathit{grantID}][\mathtt{finishRequests}]$ (Line 16). The `finishRequest` entry is written only in Line 61 and Line 80 of Algorithm A.6. Since these are the only sections where the value of this entry is written, Lemma 16 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 10 of Algorithm A.6.

The resource n is loaded from $S(c).\mathtt{grants}[\mathit{grantID}][\mathtt{resources}][\mathit{domainFirstRS}]$ in Line 13 of Algorithm A.11 (with $\mathit{domainFirstRS} = S(c).\mathtt{grants}[\mathit{grantID}][\mathtt{domainFirstRS}]$). This value can only have been stored in Line 79 of Algorithm A.7. Thus, by Lemma 18, it must hold that the request to the `/startGrantRequest` path of c that led to the creation of *grantID* in Line 10 of Algorithm A.6 must have been sent by $\mathsf{ownerOfResource}(n) = b$.

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

Lemma 20 (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 $n \equiv S^j(rs).\mathtt{userResources}[u]$ is derivable from the attacker's knowledge in S^j (i.e., $n \in d_\emptyset(S^j(na))$), it follows that

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

(2) the browser $b = \mathsf{ownerOfResource}(n)$ that owns this resource is fully corrupted in S^j, or

(3) there exist a client instance c that is honest in S^j and a key record $k \in s_0^c.\mathtt{keyRecords}[\mathit{dmnAS}]$ (for some domain $\mathit{dmnAS} \in \mathsf{dom}(\mathsf{governor}(u))$) such that $k.\mathtt{method} \equiv \mathtt{mac}$ and $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is corrupted in S^j (an honest client instance shares a symmetric key with $\mathsf{governor}(u)$ and a corrupted RS), or

(4) there exist a client instance c, a grant ID gid, and a processing step $Q = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$, such that $i < j$, $\mathsf{authenticated}_\rho^Q(\mathsf{ownerOfID}(u), c, u, \mathsf{governor}(u), gid)$, 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).

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

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

(2) b 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 any $dmnAS \in \mathsf{dom}(\mathsf{governor}(u))$) it holds true that $k.\mathtt{method} \not\equiv \mathtt{mac}$ or $\mathsf{dom}^{-1}(k.\mathtt{rs})$ is honest in S^j, and

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

   (a) c is corrupted in S^j, 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.

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 53 of Algorithm A.21. As all conditions for Lemma 15 are given, it must hold that rs sends n only to client instances c for which $\mathsf{authenticated}_\rho^{Q'}(b, c, u, \mathsf{governor}(u), gid)$ holds for some grant ID gid and some previous processing step Q'. Due to assumption (4), 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 19 of Algorithm A.11. By Lemma 19, 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. ∎

D.4 Authorization Property

Theorem 1 (Authorization Property). Let $\mathcal{GWS}$ be a GNAP web system. We say that $\mathcal{GWS}$ *is secure w.r.t. authorization* iff $\mathcal{GWS}$ fulfills the authorization property for software-only authorization and the authorization property for end users.

Proof. This directly follows from Lemma 12 and Lemma 20. ∎

Declaration

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

---

place, date, signature