Technical Report

Formal Security Analysis of the Transmitter Configuration Discovery of the OpenID Shared Signals Framework

Final Report on WP 4.1 (b)

Pedram Hosseyni Ralf Küsters Tim Würtele
{pedram.hosseyni, ralf.kuesters, tim.wuertele}@sec.uni-stuttgart.de
Institute of Information Security – University of Stuttgart, Germany

August 26, 2024

Summary. We here report on our results for Work Package 4.1 (b). In Work Package 4.1 (a), we created a formal model of the OpenID Shared Signals Framework and identified and formalized security properties that are relevant to the protocol. The goal of Work Package 4.1 (b) was to prove these formalized properties. However, when starting with the proof phase, we discovered several security-related issues w.r.t. to these security properties (see Section 2.9).

We worked with the Shared Signals WG to fix these issues and incorporate further recommendations we made in our report on Work Package 4.1 (a).

Based on the updated specifications, we were able to complete the proofs of all security properties identified in Work Package 4.1 (a).

1. Introduction

Security Event Tokens (SETs) as defined in RFC 8417 [RFC8417] are a JWT-based [RFC7519] data structure designed for transmitting information on security- and identity-related events, e.g., regarding resource management, token revocation, or changes related to user accounts. However, RFC 8417 does not define concrete events normatively and leaves concrete events open for SET profiles, which "SHOULD define syntax, semantics, subject identification, and validation" (see [RFC8417, Section 3]).

The *OpenID Shared Signals Framework (SSF)* [15] is a SET profile and specifies subject-related claims. In addition, it defines a configuration discovery mechanism for obtaining metadata about the SET Transmitter, and an event stream management API.

As agreed upon with the OIDF, this work is focused on the configuration discovery mechanism, although it also covers (a representative) part of the event stream management API. Furthermore, as agreed upon, this work is based on the third implementer's draft of the SSF specification [15].

We note that the remaining parts of SSF, as well as the CAEP and RISC specifications, do, in large part, only specify data structures (i.e., events), but do not define their semantics on either end, i.e., they neither define who exactly sends what kind of event to whom and when, nor do they

define the behavior of the SET Receiver upon receiving these events. Hence, any semantic security properties would necessarily rely on assumptions regarding event semantics and thus only apply to implementations sharing these assumptions. Therefore – and as agreed upon – these are omitted from this work.

2. Modeling Decisions and Assumptions

In the following, we describe our key modeling decisions and assumptions. We generally try to keep assumptions as minimal as possible, especially regarding security, i.e., we model the specifications with the "minimal" security allowed by the relevant specifications in mind in order to not miss possible attacks. This in particular applies to optional security measures. Where the specifications leave things to implementations or profiles, we try to make sensible assumptions on parties' behavior, balancing possible "false" attacks due to unreasonable assumptions against the potential to miss attacks due to too strong assumptions (e.g., related to what checks a party performs during a protocol execution).

In some cases, we also introduce what we call *over-approximations*, i.e., cases where our model is – if anything – less secure than a real (specification-following) implementation. Such over-approximations usually allow for a simpler model without jeopardizing the expressiveness of security proofs. However, they need to be chosen carefully, to not lead to false positives, i.e., attacks on the model which would not work in a real implementation.

Before explaining our SSF-specific assumptions, we give some background information and descriptions of assumptions inherent to the WIM methodology[1] – while some of these are rather strong assumptions, we note that the WIM methodology has been successfully applied to a wide range of protocols like OAuth 2.0, OIDC, both FAPI versions, Mozilla's (now inactive) BrowserID, etc. Hence, the WIM methodology (evidently) provides useful, yet sufficiently precise, abstractions.

Cryptography. The WIM is a symbolic, Dolev-Yao-style model, i.e., bytestrings of any kind are represented as formal terms over a set of function symbols (e.g., $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$), nonces, and constants. The nonces are considered to be infinite-entropy random values, which means they can never be guessed, and must instead be learned, e.g., from received messages. Constants, on the other hand, are considered to be publicly known. Additionally, the semantics of cryptographic primitives are defined by an equational theory (see Definition 13 in the appendix).

The latter implies that cryptography is considered to be perfect: the attacker cannot break any cryptographic primitive unless it learns the necessary keys (which are usually nonces).

Attacker Model. The WIM supports two types of attackers: Network attackers, and Web attackers. Network attackers are the original Dolev-Yao attacker model; such an attacker controls the network, i.e., can eavesdrop on all sent messages, can block or re-route messages, and inject arbitrary messages into the network, as long as it can *derive* (according to the equational theory) the message contents from its knowledge. A Web attacker is basically a corrupted endpoint in the network that may collude with other corrupted parties, send (derivable) messages with spoofed addresses, and so on. Of course, a network attacker always implies all possible Web attackers. Hence, one usually considers a network attacker unless a certain security property can only be proven under the assumption that there are only Web attackers.

Time. The WIM does not include any notion of time. Consequently, all time-based claims, values, and checks are omitted from WIM models, for example, not-before and expiration times of JWTs and tokens. Instead, one considers all these values as being valid forever.

[1] See Appendix E and the works referenced therein for a description of the WIM.

Note that strictly speaking, this is not an over-approximation: the WIM is a *possibilistic* model, i.e., anything that can happen – no matter how improbable – is considered to happen. Hence, even if we had a notion of time, the possibilistic nature of the model would still allow for arbitrarily complex attacks to happen in any non-zero time frame.

2.1. Event Types

We do not model event types and instead treat all events as being of the same type. For example, this means that we omit the corresponding claims, i.e., `events_supported`, `events_requested`, and `events_delivered` from stream configurations.

Note that this does not affect security results: event types' semantics w.r.t. the security properties are outside the scope of the analyzed specifications.

Similarly, since the actual events contained in a SET do not affect participants' behavior as far as the analyzed specifications are concerned, we do not model them.

2.2. Security Event Token Delivery Methods

Our model supports both the push (`urn:ietf:rfc:8935`) and poll (`urn:ietf:rfc:8936`) SET delivery methods. Hence, our security properties apply to those delivery methods when used (stand-alone or in parallel) as defined by [RFC8935, RFC8936] (with the respective URLs, etc. being exchanged via the event stream management API defined in [15, Section 7]). We note that this in particular implies that SETs are only transmitted via transport-protected connections, e.g., HTTPS.

2.3. Transmitter and Receiver Protocol Roles

In our model, each modeled entity, basically a single Internet-connected machine that listens to a number of IP addresses and "owns" a set of domains, can be both, Transmitter and Receiver at the same time. We chose to not separate these roles to make sure our model cannot miss mix-up attacks in which a single party plays both roles and gets confused between them.

We assume that under each domain, there is at most one issuer (whose issuer identifier URL contains an empty path element). However, we note that in our model, each entity owns multiple domains, i.e., we cover the case where one entity represents multiple issuers.

2.4. Stream ID Selection

We assume Receivers are implemented such that they do not mix up streams from different Transmitters, even if these streams use the same stream ID (such a stream ID duplication is not allowed "within" a Transmitter, see [15, Section 7.1.1]).

Consequently, we model Transmitters such that (honest) Transmitters always use fresh nonces as stream, allowing us to slightly simplify the Receiver model (and the associated security properties) in that it does not need to store stream information under an issuer identifier plus a stream ID.

Furthermore, we assume stream IDs to be public values, i.e., the attacker immediately learns a new stream's stream ID.

2.5. Stream Management API

While according to the contracted work items, our analysis only covers the configuration discovery subprotocol, we chose to include representative parts of the stream management API as well – after all, the discovery protocol serves the purpose of supplying the Receiver with information on the Transmitter, including important endpoint URLs for the stream management API. Including parts

of the stream management API also allows us to define much more natural security properties that are directly related to emitted and accepted SETs.

That said, we chose to limit our model of the stream management API to stream creation and subject adding. This selection is due to several considerations:

Firstly, these two management actions are the most security-critical, whoever can create a stream can – at least with the `urn:ietf:rfc:8935` delivery method – also receive SETs for that stream; and of course, adding subjects to existing streams can not only leak information when a sensitive subject is added to a stream of an attacker, but may also lead to honest Receivers receiving "unwanted" events (which may also leak information, e.g., if the honest Receiver publishes the received events). From a security standpoint, the other specified stream management actions can be somewhat subsumed by these (if the other stream management API endpoints employ the same kind of authentication/authorization).

Secondly, these two management actions are the minimal set of actions required to sensibly model (and formulate security properties on) the actual delivery of SETs when starting with configuration discovery.

In addition to the claims listed in Section 2.1, we also omit the `description` claim from stream configurations since it does not serve any "functional" purpose.

2.5.1. Authorization at the Stream Management API in General

In line with [15, Section 7], our Transmitter model requires Receiver authorization/authentication at the management API endpoints. Specifically, we model the OAuth 2.0 authorization defined in [15, Section 6.1.1] and pass the access token in the HTTP Authorization header.

Note that we do not model the access token issuance, since it is not part of the analyzed specifications. Instead, we initialize Transmitters and Receivers with such tokens in our model, effectively treating token issuance and management as an out-of-band mechanism.

2.5.2. Authorization at the Add Subject Endpoint

As described above, we assume that streams are initially "empty", i.e., no events will ever be delivered until at least one subject is added to the stream. This allows us to capture fine-grained access control to subject information.

Specifically, we model authorization at the add subject endpoint by means of the pre-shared (bearer) access tokens that are used to authenticate/authorize Receivers at the stream management APIs. These tokens are pre-shared such that each token represents the authorization to access information for a (disjoint) set of subjects; all subjects "within" any given token must be managed by a single issuer – that issuer is then initialized with the token. For Receivers, our model distributes these tokens such that no token is distributed twice.

2.6. Authorization during SET Delivery

For the push delivery method, we model the authorization mechanism defined in [15, Section 10.3.1.1] and [RFC8935], i.e., when creating a stream with push delivery, the Receiver may include an `authorization_header` parameter. If set by the Receiver, the Transmitter will then include this parameter's value in all push requests to the Receiver, and the Receiver will of course verify its presence. Note that this mechanism is optional and we model it as such, i.e., our security proofs do not depend on authorization during pushed SET delivery.

For poll delivery, we model a *mandatory* authorization in that the Receiver has to include the same access token in its polling requests that it used to create the stream in the first place. We note that such authorization is not mandated by SSF [15], nor by RFC 8936 [RFC8936]. However, without assuming such mandatory authorization, at least one of our security properties, namely SET

secrecy, is trivially broken: the poll endpoint URLs are not considered to be secrets, i.e., anyone can send requests to any polling endpoint.[2]

2.7. Configuration Discovery

For our model of configuration discovery, we assume (1) Receivers do not have any prior information on Transmitters (except for issuer identifiers); (2) Receivers only use issuer identifiers with the HTTPS scheme to assemble the configuration discovery endpoint URL (as recommended in [15, Section 6.2]); (3) all endpoints (including the `jwks_uri`) in an honest Transmitter's configuration document use the HTTPS scheme (as required by [15, Section 6.1]).

Since our model of the stream management API only supports the configuration and add subject endpoints, our Transmitter model omits the other endpoints from its configuration document.

Note: Assumption (1) implies that no streams are pre-configured or created out-of-band.[3] I.e., all streams in our model are set up via the stream management API (we allow an infinite number of streams for each Transmitter/Receiver pair).

Finally, according to [15, Section 10.3.1.2], polling endpoint URLs "MAY be reused across Receivers, but MUST be unique per stream for a given Receiver." To satisfy this requirement while still allowing Transmitters to reuse URLs for different Receivers, our Transmitter model includes a random value in polling URLs, making sure that these random values are unique for a given Receiver, but may collide for different Receivers.

2.8. Signed SETs

Since [RFC8417] requires implementations to either sign SETs or provide integrity protection and issuer authentication in another way, our Transmitter model always signs SETs and our Receiver model accepts SETs only if they carry a valid signature (valid w.r.t. a SET's claimed issuer, the verification keys are retrieved from the issuer's `jwks_uri` endpoint as specified in the issuer's configuration document).

2.9. Specification Version

We model the SSF specification in its third implementer's draft version [15], i.e., we assume that both Transmitters and Receivers behave according to that specification version.

Originally, i.e., for Work Package 4.1 (a), we modeled and started to analyze the second implementer's draft. However, several security related issues emerged during this analysis, leading to updates to the specification and the publication of a third implementer's draft that contains these updates (together with additional changes, some of them rooted in recommendations from our report on Work Package 4.1 (a)). Apart from our report on Work Package 4.1 (a), we refer to [10–14] for descriptions of the identified issues and further contributions.

3. Notes on the SSF Specification

Based on our work with the SSF specification, we suggest the working group consider the following changes (see our report on WP 4.1(a) for recommendations that have since been implemented).

[2] We assume the polling endpoints not to be secrets because there is no requirement for Transmitters to include any randomness in them. Furthermore, these URLs are allowed to be reused across multiple Receivers, see [15, Section 10.3.1.2].

[3] This is in line with [15, Section 7.1.1.1]: "In order to communicate events from a Transmitter to a Receiver, a Receiver MUST first create an Event Stream."

Audience Validation in Stream Configuration. While Receivers are mandated to validate the audience value in SETs (due to [RFC7519, Section 4.1.3]), they are currently not required to validate the audience value in stream configurations returned by a Transmitter, e.g., in a stream creation response. Our Receiver model respects this and hence mostly ignores streams' audience values. For SET validation, our Receiver model instead compares the SET's audience value against an expected value based on the access token used by the Receiver when requesting creation of the stream (since this is where the Transmitter is required to derive an audience value from the Receiver's authorization, see [15, Section 7]).

However, it is likely that implementers use the stream's audience value to validate SETs against, hence, we recommend to mandate Receivers-side validation of stream audience values.

Authorization at the Poll Delivery Endpoint. As we note in Section 2.6, poll endpoint URLs are not required to be secret, i.e., SETs could be requested by any party. For use cases requiring confidentiality of SETs, we recommend mandating authorization at the poll endpoint.

4. Informal Security Properties

In the following, we give an informal overview of the security properties that we proved as part of this Work Package. See Appendix C for the formalized properties.

4.1. Configuration Discovery Integrity

This integrity property considers the discovery mechanism in isolation and states that Receivers get the correct configuration documents (from honest Transmitters). I.e., when an honest Receiver accepts a configuration document whose issuer claim contains the identifier of an honest Transmitter, then all data in that configuration document is correct.

Note: After accepting a Transmitter's configuration document, the Receiver in our model requests the Transmitter's JWK Set (using the `jwks_uri` claim in the configuration document), and this property includes the correctness of the returned keys.

For example, this property captures attacks in which an attacker can trick a Receiver into using an attacker-controlled configuration endpoint.

4.2. Session Integrity for SETs

With this property, we capture that if an honest Receiver accepts a SET (regardless of the delivery method) whose `iss` claim contains the identifier of an honest Transmitter, then (1) that issuer did indeed issue the SET, and (2) the SET was indeed issued for that Receiver.

For example, this property captures injection attacks in which an attacker can trick a Receiver into accepting SETs that have been created by the attacker or at least have not been intended for that Receiver.

4.3. Confidentiality of SETs

Since SETs may contain sensitive data, this property captures that any SET whose `iss` claim contains the identifier of an honest Transmitter, that has a valid signature (w.r.t. that issuer), and whose `aud` claim refers to an honest Receiver, can not leak to the attacker.

Note that we have to limit this property to honest Transmitters and Receivers – a corrupted Transmitter can of course leak any SET it issues, and a corrupted Receiver can leak any SET it receives.

4.4. Authorization

With this property, we capture that whenever an honest Transmitter issues a SET for some subject, then the Receiver (as identified by the SET's `aud` claim) is authorized to receive information on that subject.

References

[RFC7515] M. B. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: https://www.rfc-editor.org/info/rfc7515.

[RFC7519] M. B. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519.

[RFC7617] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. Sept. 2015. DOI: 10.17487/RFC7617. URL: https://www.rfc-editor.org/info/rfc7617.

[RFC8417] P. Hunt, M. B. Jones, W. Denniss, and M. Ansari. *Security Event Token (SET)*. RFC 8417. July 2018. DOI: 10.17487/RFC8417. URL: https://www.rfc-editor.org/info/rfc8417.

[RFC8935] A. Backman, M. B. Jones, M. Scurtescu, M. Ansari, and A. Nadalin. *Push-Based Security Event Token (SET) Delivery Using HTTP*. RFC 8935. Nov. 2020. DOI: 10.17487/RFC8935. URL: https://www.rfc-editor.org/info/rfc8935.

[RFC8936] A. Backman, M. B. Jones, M. Scurtescu, M. Ansari, and A. Nadalin. *Poll-Based Security Event Token (SET) Delivery Using HTTP*. RFC 8936. Nov. 2020. DOI: 10.17487/RFC8936. URL: https://www.rfc-editor.org/info/rfc8936.

[1] R. Berjon et al., eds. *HTML5, W3C Recommendation*. Oct. 28, 2014. URL: http://www.w3.org/TR/html5/.

[2] L. Chen, S. Englehardt, M. West, and J. Wilander. *Cookies: HTTP State Management Mechanism*. Internet-Draft draft-ietf-httpbis-rfc6265bis-09. Work in Progress. Internet Engineering Task Force, Oct. 2021. 59 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-09.

[3] D. Fett. "An Expressive Formal Model of the Web Infrastructure". PhD thesis. 2018.

[4] D. Fett, P. Hosseyni, and R. Kusters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019. DOI: 10.1109/sp.2019.00067.

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

[6] D. Fett, R. Küsters, and G. Schmitz. "Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web". In: *ESORICS*. Vol. 9326. LNCS. 2015, pp. 43–65.

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

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

[9] D. Fett, R. Küsters, and G. Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *CSF*. 2017.

[10] P. Hosseyni, R. Küsters, and T. Würtele. *Attacker Stream Subject Insertion*. URL: https://github.com/openid/sharedsignals/issues/160.

[11] P. Hosseyni, R. Küsters, and T. Würtele. *Fix Indefinite Articles Before "SSF"*. URL: https://github.com/openid/sharedsignals/pull/145.

[12] P. Hosseyni, R. Küsters, and T. Würtele. *Issuer Mix-Up*. URL: https://github.com/openid/sharedsignals/issues/162.

[13] P. Hosseyni, R. Küsters, and T. Würtele. *Minor editorial nits*. URL: https://github.com/openid/sharedsignals/pull/146.

[14] P. Hosseyni, R. Küsters, and T. Würtele. *Stream Audience Mix-Up*. URL: https://github.com/openid/sharedsignals/issues/161.

[15] A. Tulshibagwale, T. Cappalli, M. Scurtescu, A. Backman, J. Bradley, and S. Miel. *OpenID Shared Signals Framework Specification 1.0. 3rd Implementer's Draft*. OpenID Foundation, June 25, 2024. URL: https://openid.net/specs/openid-sharedsignals-framework-1_0-03.html.

A. SSF Configuration Discovery Model

In the following, we define our formal model of the SSF configuration discovery (and parts of the event stream management API). Note that we use notation defined by the generic WIM (see Appendix E) without further introduction.

A.1. Protocol Participants

Let SSFTR be the (finite) set of atomic DY processes representing SSF Transceivers. We define these processes in Appendix A.4. Apart from the attacker, no further participants are modeled (see also Appendix B). Note that since the attacker is a network attacker, services like DNS and routing are subsumed (and controlled) by the attacker.

A.2. Identifiers in the Protocol

A.2.1. Receiver Identification

From the point of view of a Transmitter, its Receivers are identified by their authentication credentials (see Appendix A.3 below) at the stream management API endpoints. Since we require such authentication in our model, the Transmitter model uses the same stream management API endpoint URLs for all Receivers, while still being able to distinguish between Receivers (as required by SSF [15, Section 7.1]).

To capture the requirements on Receiver authentication in [15, Section 7.1] – specifically the identification of audience values – we define the following mapping from Receiver authentication credentials to Receiver audience values (see Appendix A.3 for the definitions of $\mathsf{ReceiverCreds}$ and $\mathsf{sharedSecrets}$):

$$\mathsf{receiverAudience}\colon \mathsf{ReceiverCreds} \to \mathbb{S}$$

we restrict this mapping such that the assigned audience values are unique for a given issuer (according to the $\mathsf{sharedSecrets}$ mapping), i.e.,

$$\begin{aligned}
&\forall aud \in \mathbb{S}, \forall rc_1, rc_2 \in \mathsf{receiverAudience}^{-1}(aud)\colon rc_1 \neq rc_2 \\
&\Longrightarrow \\
&\exists iss_1, iss_2 \in \mathsf{IssIDs}\colon iss_1 \neq iss_2 \wedge \\
&rc_1 \in \bigcup_{ssftr \in \mathsf{SSFTR}} \mathsf{sharedSecrets}(ssftr, iss_1) \wedge \\
&rc_2 \in \bigcup_{ssftr \in \mathsf{SSFTR}} \mathsf{sharedSecrets}(ssftr, iss_2)
\end{aligned}$$

A.2.2. Transmitter Identification

Transmitters, on the other hand, are identified by issuer identifiers [15, Section 7.1.1] that we model as follows:

Definition 1 (Issuer Identifiers). We define the (finite) set IssIDs of issuer identifiers as $\mathsf{IssIDs} \subset \mathsf{URLs}$ such that $\forall i \in \mathsf{IssIDs}\colon i \sim \langle \mathtt{URL}, \mathtt{S}, *, \varepsilon, \langle\rangle, \bot \rangle$.

Note that in conjunction with the dom mapping, this definition induces the set $issIDs^p$ of all issuer identifiers of a process p as $issIDs^p := \{iss \mid iss \in \mathsf{IssIDs} \wedge iss.\mathtt{host} \in \mathsf{dom}(p)\}$.

Furthermore, we define $\mathsf{supportedIssuers}\colon \mathsf{SSFTR} \to 2^{\mathsf{IssIDs}}$ to be a mapping from SSF Transceivers to the set of issuers they support (i.e., the issuers that the Transceiver supports in its role as a Receiver).

A.2.3. Subject Identification

In our model, we use a simplified form of the `sub_id` claim to identify a SET's subject. Specifically, such claims only contain a single string that fully identifies the subject.

Definition 2 (Subject Identifiers). We define the (finite) set SubIDs of subject identifiers as $\mathsf{SubIDs} \subsetneq \mathbb{S}$. Furthermore, we define a mapping $\mathsf{issuerOf}\colon \mathsf{SubIDs} \to \mathsf{IssIDs}$ that assigns an issuer identifier to each subject identifier. Given $iss = \mathsf{issuerOf}(subjID)$, we say that *iss manages subjID*.

Note that the above implies that each subject identifier is managed by exactly one issuer identifier.

A.3. Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into disjoint sets, an infinite set N, and finite sets K_{TLS}, K_{sign}, and $\mathsf{ReceiverCreds}$:

$$\mathcal{N} = N \uplus K_{\text{TLS}} \uplus K_{\text{sign}} \uplus \mathsf{ReceiverCreds}$$

These sets are used as follows:

- The set N contains the nonces that are available for the DY processes.
- 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 $tlskeys^p = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(p)\}\rangle$ (a sequence of pairs, i.e., a dictionary that maps domains to their respective private keys).
- The set K_{sign} contains the (private) keys that will be used by Transmitters to sign SETs. Let $\mathsf{signkey}\colon \mathsf{IssIDs} \to K_{\text{sign}}$ be an injective mapping that assigns a (different) signing key to each issuer identifier.
- The set $\mathsf{ReceiverCreds}$ contains symmetric, pre-shared secrets used for authentication of Receivers to Transmitters at the stream management APIs. Let $\mathsf{sharedSecrets}\colon \mathsf{SSFTR} \times \mathsf{IssIDs} \to 2^{\mathsf{ReceiverCreds}}$ be a mapping that assigns (possibly empty) sets of shared secrets to pairs of SSF Transceivers (in their role as Receivers) and issuer identifiers, such that no shared secret is used more than once, i.e.,

  $$\begin{aligned}&\forall ssftr_1, ssftr_2 \in \mathsf{SSFTR}, \forall iss_1, iss_2 \in \mathsf{IssIDs}, \forall rc \in \mathsf{ReceiverCreds}\colon\\ &rc \in \mathsf{sharedSecrets}(ssftr_1, iss_1) \land rc \in \mathsf{sharedSecrets}(ssftr_2, iss_2)\\ &\Rightarrow ssftr_1 = ssftr_2 \land iss_1 = iss_2\end{aligned}$$

  Furthermore, we define a mapping $\mathsf{permittedSubjects}\colon \mathsf{ReceiverCreds} \to 2^{\mathsf{SubIDs}}$ that assigns a (possibly empty) set of subject identifiers to each such shared secret, modeling the Transmitter's choice of whether to add a subject to a stream based on the stream's (authenticated) audience. We restrict $\mathsf{permittedSubjects}$ such that all subject identifiers for a shared secret rc are managed by the issuer that is associated with rc (see also Definition 2):

  $$\begin{aligned}&\forall rc \in \mathsf{ReceiverCreds}\ \exists ssftr_r \in \mathsf{SSFTR}\ \forall subjID \in \mathsf{permittedSubjects}(rc)\colon\\ &rc \in \mathsf{sharedSecrets}(ssftr_r, \mathsf{issuerOf}(subjID))\end{aligned}$$

A.4. SSF Transceiver Model

An SSF Transceiver $\mathsf{ssftr} \in \mathsf{SSFTR}$ is a Web server modeled as an atomic DY process $(I^{\mathsf{ssftr}}, Z^{\mathsf{ssftr}}, R^{\mathsf{ssftr}}, s_0^{\mathsf{ssftr}})$ with the addresses $I^{\mathsf{ssftr}} := \mathsf{addr}(\mathsf{ssftr})$. We define the set Z^{ssftr} of states of ssftr and the initial state s_0^{ssftr} as follows:

Definition 3 (SSF Transceiver State). A *state* $s \in Z^{\mathsf{ssftr}}$ *of an SSF Transceiver* ssftr is a term of the form $\langle \mathit{DNSaddress}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{corrupt}, \mathit{keyMapping}, \mathit{tlskeys}, \mathit{TXjwks}, \mathit{pendingStreamIds}, \mathit{TXstreams}, \mathit{TXreceiverCreds}, \mathit{RXstreams}, \mathit{RXissuers}, \mathit{RXcredentials}, \mathit{RXsets}, \mathit{RXpushEP} \rangle$ with $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in \big[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\big]$, $\mathit{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{keyMapping} \in \big[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\big]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (all former components as in Definition 57), $\mathit{TXjwks} \in [\mathsf{IssIDs} \times K_{\mathrm{sign}}]$, $\mathit{pendingStreamIds} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{TXstreams} \in \big[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\big]$, $\mathit{TXreceiverCreds} \in \big[\mathsf{IssIDs} \times \big[\mathsf{ReceiverCreds} \times \mathcal{T}_{\mathcal{N}}\big]\big]$, $\mathit{RXstreams} \in \big[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\big]$, $\mathit{RXissuers} \in \big[\mathsf{URLs} \times \mathcal{T}_{\mathcal{N}}\big]$, $\mathit{RXcredentials} \in \big[\mathsf{IssIDs} \times \mathcal{T}_{\mathcal{N}}\big]$, $\mathit{RXsets} \in \big[\mathsf{URLs} \times \mathcal{T}_{\mathcal{N}}\big]$, and $\mathit{RXpushEP} \in \mathsf{URLs}$.

An *initial state* s_0^{ssftr} *of* ssftr is a state of ssftr with

- $s_0^{\mathsf{ssftr}}.\mathtt{DNSaddress} \in \mathsf{IPs}$,
- $s_0^{\mathsf{ssftr}}.\mathtt{pendingDNS} \equiv \langle\rangle$,
- $s_0^{\mathsf{ssftr}}.\mathtt{pendingRequests} \equiv \langle\rangle$,
- $s_0^{\mathsf{ssftr}}.\mathtt{corrupt} \equiv \bot$,
- $s_0^{\mathsf{ssftr}}.\mathtt{keyMapping} \equiv \big\langle \big\{ \langle d, \mathsf{pub}(\mathsf{tlskey}(d)) \rangle \mid d \in \mathsf{Doms} \big\} \big\rangle$,
- $s_0^{\mathsf{ssftr}}.\mathtt{tlskeys} \equiv \mathit{tlskeys}^{\mathsf{ssftr}}$ (see Appendix A.3),
- $s_0^{\mathsf{ssftr}}.\mathtt{TXjwks} \equiv \big\langle \big\{ \langle \mathit{iss}, \mathsf{signkey}(\mathit{iss}) \rangle \mid \mathit{iss} \in \mathit{issIDs}^{\mathsf{ssftr}} \big\} \big\rangle$ (a dictionary that maps from issuer identifiers managed by this process to signing keys, see also Definition 1 and Appendix A.3),
- $s_0^{\mathsf{ssftr}}.\mathtt{pendingStreamIds} \equiv \langle\rangle$,
- $s_0^{\mathsf{ssftr}}.\mathtt{TXstreams} \equiv \langle\rangle$,
- $s_0^{\mathsf{ssftr}}.\mathtt{TXreceiverCreds} \equiv \big\langle \big\{ \langle \mathit{iss}, \mathit{txReceiverCreds}(\mathit{iss}) \rangle \mid \mathit{iss} \in \mathit{issIDs}^{\mathsf{ssftr}} \big\} \big\rangle$, i.e., a dictionary mapping from this ssftr's issuer identifiers to entries $\mathit{txReceiverCreds}(\mathit{iss})$, which we define as

$$\Big\langle \Big\{ \langle \mathit{rc}, \mathit{receiverInformation}(\mathit{rc}) \rangle \mid \mathit{rc} \in \bigcup_{\substack{\mathit{ssftr}_r \in \mathsf{SSFTR} \text{ s.t.} \\ \mathit{iss} \in \mathsf{supportedIssuers}(\mathit{ssftr}_r)}} \mathsf{sharedSecrets}(\mathit{ssftr}_r, \mathit{iss}) \Big\} \Big\rangle$$

  with $\mathit{receiverInformation}(\mathit{rc}) := [\mathtt{aud}\colon \mathsf{receiverAudience}(\mathit{rc}), \mathtt{subjects}\colon \mathsf{permittedSubjects}(\mathit{rc})]$, i.e., $\mathit{txReceiverCreds}(\mathit{iss})$ is a dictionary that maps each of the shared secrets of iss to information about the Receiver with whom the respective secret is shared.,
- $s_0^{\mathsf{ssftr}}.\mathtt{RXstreams} \equiv \langle\rangle$,
- $s_0^{\mathsf{ssftr}}.\mathtt{RXissuers} \equiv \langle\rangle$,
- $s_0^{\mathsf{ssftr}}.\mathtt{RXcredentials} \equiv \big\langle \big\{ \langle \mathit{iss}, \mathit{rxCreds}(\mathit{iss}, \mathsf{ssftr}) \rangle \mid \mathit{iss} \in \mathsf{supportedIssuers}(\mathsf{ssftr}) \big\} \big\rangle$, where $\mathit{rxCreds}(\mathit{iss}, \mathsf{ssftr})$ is defined as $\big\langle \big\{ \langle \mathit{rc}, \mathsf{receiverAudience}(\mathit{rc}) \rangle \mid \mathit{rc} \in \mathsf{sharedSecrets}(\mathsf{ssftr}, \mathit{iss}) \big\} \big\rangle$. I.e., overall, this state subterm maps from each supported issuer identifier to a dictionary of secrets (and the corresponding audience) shared with the respective issuer,
- $s_0^{\mathsf{ssftr}}.\mathtt{RXsets} \equiv \langle\rangle$, and

- $s_0^{\mathsf{ssftr}}.\texttt{RXpushEP} \equiv \langle \texttt{URL}, \texttt{S}, d, /\texttt{push-ep}, \langle\rangle, \bot\rangle$ for a domain $d \in \mathsf{dom}(\mathsf{ssftr})$.

The only thing left to define for the SSF Transceiver model is its relation R^{ssftr}. This relation is based on the WIM's generic HTTPS server model (see Appendix E.12). Hence, we only need to define those parts (functions) of R^{ssftr} that differ from (or do not exist in) the generic server model; we provide these in the following algorithms. Note that these algorithms contain *placeholders* (that we write as ν_x for some x) to model generation of fresh nonces.

Algorithm 1 Relation of an SSF Transceiver R^{ssftr} – Processing HTTPS requests

→ **Process an incoming HTTPS request.** Other message types are handled in separate functions. m is the incoming message (decrypted), 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 ssftr.

1: **function** PROCESS_HTTPS_REQUEST(m, k, a, f, s')

Transmitter: Configuration Discovery and JWKS Endpoints

2: **if** $m.\texttt{path} \equiv /.\texttt{well-known}/\texttt{ssf-configuration}$ **then**
3: **let** $issuer := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, \varepsilon, \langle\rangle, \bot\rangle$
4: **if** $issuer \notin s'.\texttt{TXjwks}$ **then**
5: **stop** → $m.\texttt{host}$ is not part of an issuer identifier of this Transmitter.
6: **let** $trConf := [\texttt{issuer}: issuer]$
7: **let** $trConf[\texttt{jwks\_uri}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{jwks}, \langle\rangle, \bot\rangle$
8: **let** $trConf[\texttt{configuration\_endpoint}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{configure-stream}, \langle\rangle, \bot\rangle$
9: **let** $trConf[\texttt{add\_subject\_endpoint}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{add-subject}, \langle\rangle, \bot\rangle$
10: **let** $trConf[\texttt{authorization\_schemes}] := \langle [\texttt{spec\_urn}: \texttt{urn:ietf:rfc:6749}] \rangle$
11: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 200, \langle\rangle, trConf \rangle, k)$
12: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
13: **else if** $m.\texttt{path} \equiv /\texttt{jwks}$ **then**
14: **let** $issuer := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, \varepsilon, \langle\rangle, \bot\rangle$
15: **if** $issuer \notin s'.\texttt{TXjwks}$ **then**
16: **stop** → $m.\texttt{host}$ is not an issuer identifier of this Transmitter.
17: **let** $jwks := \mathsf{pub}(s'.\texttt{TXjwks}[issuer])$
18: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 200, \langle\rangle, jwks \rangle, k)$
19: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'

Transmitter: Configuration Endpoint

20: **else if** $m.\texttt{path} \equiv /\texttt{configure-stream}$ **then**
21: **if** $m.\texttt{method} \equiv \texttt{POST}$ **then** → Create a new stream [15, Section 7.1.1.1].
22: **call** CREATE_STREAM(m, k, a, f, s') → See Algorithm 2.
23: **else**
24: **stop** → Not modeled, see Section 2.5.

Transmitter: Other Endpoints

25: **else if** $m.\texttt{path} \equiv /\texttt{add-subject}$ **then**
26: **call** PROCESS_ADD_SUBJECT_REQUEST(m, k, a, f, s') → See Algorithm 3.
27: **else if** $m.\texttt{path} \equiv /\texttt{poll}$ **then**
28: **call** PROCESS_POLL_REQUEST(m, k, a, f, s') → See Algorithm 4.

Receiver Endpoints

29: **else if** $m.\texttt{path} \equiv /\texttt{push-ep} \wedge m.\texttt{method} \equiv \texttt{POST}$ **then**
30: **call** PROCESS_PUSH_REQUEST(m, k, a, f, s') → See Algorithm 6.
31: **else**
32: **stop** → Unknown endpoint.

Algorithm 2 Relation of an SSF Transceiver R^{ssftr} – Processing a stream creation request

→ **Create a new stream.** m is the HTTP POST request, 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 ssftr.

function CREATE_STREAM(m, k, a, f, s')
  let *issuer* := $\langle$`URL`, `S`, m.`host`, ε, $\langle\rangle$, $\bot\rangle$
  if `Authorization` $\notin m$.`headers` **then**
    stop → Management endpoint requires authorization [15, Section 7].
  let *receiverCred* := m.`headers`[`Authorization`].2
  let *audience* := s'.`TXreceiverCreds`[*issuer*][*receiverCred*][`aud`]
  if *audience* $\equiv \langle\rangle$ **then**
    stop → No audience associated with the authorization token [15, Section 7].
  if m.`body` $\equiv \langle\rangle \vee m$.`body` $\equiv$ [`delivery`: [`method`: `urn:ietf:rfc:8936`]] **then**
    → Receiver requested poll or did not specify a delivery method → poll delivery [15, Section 7.1.1.1].
    let *deliveryMethod* := `urn:ietf:rfc:8936`
    let *pollRand* ← $\mathbb{S}$ **such that** → See [15, Section 10.3.1.2] and Section 2.7.
      ↪ $\nexists sc \in s'$.`TXstreams`: *sc*[`aud`] $\equiv$ *audience* $\wedge$
      ↪ *sc*[`endpoint_url`].`parameters`[`rand`] $\equiv$ *pollRand*
      ↪ **if possible; otherwise stop**
    let *deliveryEP* := $\langle$`URL`, `S`, m.`host`, /`poll`, [`rand`: *pollRand*], $\bot\rangle$
    let *token* := m.`headers`[`Authorization`].2 → Stream can only be polled with this token.
  else if m.`body` $\sim$ [`delivery`: [`method`: `urn:ietf:rfc:8935`, `endpoint_url`: $*$]] **then**
    let *deliveryMethod* := `urn:ietf:rfc:8935`
    let *deliveryEP* := m.`body`[`delivery`][`endpoint_url`] → Required, see [15, Section 7.1.1].
    let *token* := m.`body`[`authorization_header`] → Optional push secret, see [15, Section 10.3.1.1]. If `authorization_header` $\notin m$.`body`, then *token* takes the dummy value $\langle\rangle$.
  else
    stop $\langle\langle f, a, \mathsf{enc_s}(\langle$`HTTPResp`, m.`nonce`, 400, $\langle\rangle$, $\langle\rangle\rangle, k)\rangle\rangle, s'$ → Invalid request.
  → Stream IDs are provided by the attacker, see Algorithm 13.
  let *streamID* ← s'.`pendingStreamIds` **if possible; otherwise stop**
  let s'.`pendingStreamIds` := s'.`pendingStreamIds` $-^{\langle\rangle}$ *streamID*
  let *streamConf* := [`stream_id`: *streamID*, `iss`: *issuer*]
  let *streamConf*[`aud`] := *audience*
  let *streamConf*[`delivery`] := [`method`: *deliveryMethod*, `endpoint_url`: *deliveryEP*]
  → Note: We do not model event types, verification interval, and the description claim in stream configurations. See Section 2.
  let m' := $\mathsf{enc_s}(\langle$`HTTPResp`, m.`nonce`, 201, $\langle\rangle$, *streamConf*$\rangle, k)$
  let *streamConf*[`subjects`] := $\langle\rangle$
  let *streamConf*[`token`] := *token* → The token/secret used for delivery (which is not necessarily the receiver credential in case of push delivery).
  let s'.`TXstreams`[*streamID*] := *streamConf*
  stop $\langle\langle f, a, m'\rangle, \langle a, a, streamID\rangle\rangle, s'$ → Note that we leak the *streamID* here.

Algorithm 3 Relation of an SSF Transceiver R^{ssftr} – Processing an add subject request

→ **Add subject to a stream.** m is the HTTP request, 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 ssftr.

function PROCESS_ADD_SUBJECT_REQUEST(m, k, a, f, s')
  if m.method $\not\equiv$ POST **then**
    stop
  let *accessToken* := m.headers[Authorization][Bearer]
  let *streamID* := m.body[stream_id]
  let *subjectID* := m.body[subject]
→ Check access token validity for this operation:
  let *issuer* := $\langle$URL, S, m.host, ε, $\langle\rangle$, $\bot\rangle$
  let *receiverInfo* := s'.TXreceiverCreds[*issuer*][*accessToken*]
  let *audience* := *receiverInfo*[aud]
  let *permittedSubjects* := *receiverInfo*[subjects]
  if *audience* $\not\equiv$ s'.TXstreams[*streamID*][aud] **then**
    stop → Token does not authorize adding subjects to this stream.
  if *issuer* $\not\equiv$ s'.TXstreams[*streamID*][iss] **then**
    stop → Issuer identifier that received the add subject request is $\neq$ issuer of the stream.
  if *subjectID* $\notin^{\langle\rangle}$ *permittedSubjects* $\wedge$ *subjectID* $\not\equiv \langle\rangle$ **then**
    stop → Token insufficient to add subject to the stream.
→ Add subject identifier to the list of subjects of the stream:
  let s'.TXstreams[*streamID*][subjects] := s'.TXstreams[*streamID*][subjects] $+^{\langle\rangle}$ *subjectID*
  let m' := $\mathsf{enc_s}(\langle$HTTPResp, m.nonce, 200, $\langle\rangle$, $\langle\langle\rangle\rangle, k)$
  stop $\langle\langle f, a, m'\rangle\rangle$, s'

Algorithm 4 Relation of an SSF Transceiver R^{ssftr} – Processing a SET polling request

→ **Process a SET polling request.** m is the HTTP request, 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 ssftr.

function PROCESS_POLL_REQUEST(m, k, a, f, s')
  if Authorization $\notin$ m.headers **then**
    stop → We require authorization for SET polling, see Section 2.6.
  let *token* := m.headers[Authorization].2
  let *epUrl* := $\langle$URL, S, m.host, m.path, m.parameters, $\bot\rangle$
  let $\langle$*streamID*, *streamConfig*$\rangle$ $\leftarrow$ s'.TXstreams **such that**
    ↪ *streamConfig*[delivery] $\equiv$ [method: urn:ietf:rfc:8936, endpoint_url: *epUrl*] $\wedge$
    ↪ *streamConfig*[token] $\equiv$ *token* **if possible; otherwise stop**
  let *set* := CREATE_SET(*streamConfig*, s') → See Algorithm 5.
  let m' := $\mathsf{enc_s}(\langle$HTTPResp, m.nonce, 200, $\langle\rangle$, [sets: [*set*[jti]: *set*]]$\rangle, k)$
  stop $\langle\langle f, a, m'\rangle\rangle$, s'

Algorithm 5 Relation of an SSF Transceiver R^{ssftr} – Create/Issue a SET

→ **Create/Issue a SET.** *streamConfig* is the configuration of the stream for which a SET is to be issued and s' is the current state of the atomic DY process ssftr. This function returns a signed SET (without events, see Section 2.1) and does not modify the state.

1: **function** CREATE_SET(*streamConfig*, s')
2: **let** *issuer* := *streamConfig*[`issuer`]
3: **let** *streamAud* := *streamConfig*[`aud`]
4: **let** *jti* := ν_{SETjti}
5: **let** *subjectID* ← *streamConfig*[`subjects`] → If subjects ≡ ⟨⟩: no processing step (Appendix E.6).
6: **let** *setBody* := [`iss`: *issuer*, `jti`: *jti*, `aud`: *streamAud*, `sub_id`: *subjectID*]
7: **let** *signKey* := s'.`TXjwks`[*issuer*]
8: **let** *set* := sig(*setBody*, *signKey*)
9: **return** *set*

Algorithm 6 Relation of an SSF Transceiver R^{ssftr} – Process a pushed SET

→ **Process a pushed SET.** m is the HTTP POST request, 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 ssftr.

1: **function** PROCESS_PUSH_REQUEST(m, k, a, f, s')
2: **let** *set* := m.`body`
3: **let** *setBody* := extractmsg(*set*)
4: **let** *iss* := *setBody*[`iss`]
5: **let** *aud* := *setBody*[`aud`]
6: **let** *verificationKey* := s'.`RXissuers`[*iss*][`jwks`]
7: **if** checksig(*set*, *verificationKey*) ≢ ⊤ **then** → See [RFC8935, Section 2].
8: **stop**
9: **let** *token* := m.`headers`[`Authorization`]
10: **let** *streamID* ← $\mathcal{T}_{\mathcal{N}}$ **such that**
↪ *streamID* ∈ s'.`RXstreams` ∧
↪ s'.`RXstreams`[*streamID*][`issuer`] ≡ *iss* ∧ → See [15, Section 10.2].
↪ s'.`RXstreams`[*streamID*][`token`] ≡ *token*
↪ **if possible; otherwise stop**
11: **let** *expectedAud* := s'.`RXstreams`[*streamID*][`expectedAud`]
12: **if** *aud* ≢ *expectedAud* **then** → As required by [RFC7519, Section 4.1.3].
13: **stop**
14: **let** s'.`RXsets`[*iss*] := s'.`RXsets`[*iss*] $+^{\langle\rangle}$ *setBody* → Store, i.e., accept the SET.
15: **stop** ⟨⟩, s'

Algorithm 7 Relation of an SSF Transceiver R^{ssftr} – Processing HTTPS responses

→ **Process an incoming HTTPS response.** m is the incoming message (decrypted), *reference* is the reference term stored when sending the corresponding HTTPS request (usually via HTTPS_SIMPLE_SEND, see Algorithm 26), *request* is that request (prior to encryption), a is the receiver address, and f the sender address of the message. s' is the current state of the atomic DY process ssftr.

1: **function** PROCESS_HTTPS_RESPONSE(m, *reference*, *request*, a, f, s')

Receiver: Configuration Discovery and JWKS Responses

2: **if** *reference*[responseTo] ≡ DISCOVERY **then**
3: **let** *issuer* := m.body[issuer]
4: **if** *issuer* ≢ *reference*[iss] **then**
5: **stop** → Issuer does not match discovery request target, see [15, Section 6.2.4].
6: **let** s'.RXissuers[*issuer*] := m.body → Store configuration...
7: **let** *jwksUri* := m.body[jwks_uri] → ... and request JWKs for the issuer.
8: **let** *req* := ⟨HTTPReq, ν_{jwks}, GET, *jwksUri*.host, *jwksUri*.path, *jwksUri*.parameters, ⟨⟩, ⟨⟩⟩
9: **call** HTTPS_SIMPLE_SEND([responseTo: JWKS, issuer: *issuer*], *req*, a, s')
10: **else if** *reference*[responseTo] ≡ JWKS **then**
11: **let** s'.RXissuers[*reference*[issuer]][jwks] := m.body
12: **stop** ⟨⟩, s'

Receiver: Response to Create Stream Request

13: **else if** *reference*[responseTo] ≡ CREATE_STREAM **then**
14: **let** *streamID* := m.body[stream_id]
15: **if** *streamID* ∈ s'.RXstreams **then**
16: **stop** → Honest Transmitters use unique stream IDs, see Section 2.4.
17: **if** m.body[issuer] ≢ *reference*[issuer] **then** → See [15, Section 7.1.1.1].
18: **stop**
19: **let** s'.RXstreams[*streamID*] := m.body
20: **let** s'.RXstreams[*streamID*][expectedAud] := *reference*[expectedAud]
21: **if** *reference*[token] ≢ ⊥ **then** → Use a delivery token for this stream (see Section 2.2)?
22: **let** s'.RXstreams[*streamID*][token] := *reference*[token]
23: **stop** ⟨⟩, s'

Receiver: Response to Poll Request

24: **else if** *reference*[responseTo] ≡ POLL **then**
25: **let** *streamID* := *reference*[streamID]
26: **let** *expectedAud* := s'.RXstreams[*streamID*][expectedAud]
27: **let** *expectedIssuer* := s'.RXstreams[*streamID*][issuer]
28: **let** *verificationKey* := s'.RXissuers[*expectedIssuer*][jwks]
29: **for** *jti* ∈ m.body[sets] **do**
30: **let** *set* := m.body[sets][*jti*]
31: **if** checksig(*set*, *verificationKey*) ≢ ⊤ **then**
32: **stop** → See [RFC8417, Section 5.1] and [RFC7515, Section 5.2].
33: **let** *setBody* := extractmsg(*set*)
34: **if** *setBody*[aud] ≢ *expectedAud* **then** → As required by [RFC7519, Section 4.1.3].
35: **stop**
36: **if** *setBody*[iss] ≢ *expectedIssuer* **then** → As required by [15, Section 10.2].
37: **stop**
38: **let** s'.RXsets[*expectedIssuer*] := s'.RXsets[*expectedIssuer*] $+^{\langle\rangle}$ *setBody*
39: **stop** ⟨⟩, s'

Algorithm 8 Relation of an SSF Transceiver R^{ssftr} – Handle trigger events

→ **Perform random/asynchronous actions when triggered.** a is the address on which the SSF Transceiver received the trigger event, and s' is the current state of the atomic DY process ssftr.

function PROCESS_TRIGGER(a, s')
  let *action* ← {RX_DISCOVERY, RX_CREATE_STREAM, RX_ADD_SUB_REQ, RX_POLL, TX_PUSH}
  switch *action* **do**
    case RX_DISCOVERY → Initiate configuration discovery.
      let *issDom* ← Doms
      let *req* := ⟨HTTPReq, ν_{disc}, GET, *issDom*, /.well-known/ssf-configuration, ⟨⟩, ⟨⟩, ⟨⟩⟩
      let *iss* := ⟨URL, S, *issDom*, ε, ⟨⟩, $\bot$⟩ → Use HTTPS URL, see [15, Section 6.2].
      call HTTPS_SIMPLE_SEND([responseTo: DISCOVERY, iss: *iss*], *req*, *a*, s')
    case RX_CREATE_STREAM → Request creation of a new stream.
      call REQUEST_NEW_STREAM(a, s') → See Algorithm 9.
    case RX_ADD_SUB_REQ → Request addition of a subject to an existing stream.
      call SEND_ADD_SUBJECT_REQUEST(a, s') → See Algorithm 10.
    case RX_POLL → Poll SETs for an existing stream.
      call SEND_POLL_REQUEST(a, s') → See Algorithm 11.
    case TX_PUSH → Push SETs for an existing stream.
      call SEND_PUSH_REQUEST(a, s') → See Algorithm 12.

Algorithm 9 Relation of an SSF Transceiver R^{ssftr} – Request creation of a new stream

function REQUEST_NEW_STREAM(a, s')
  let *issuer* ← $\mathcal{T}_{\mathcal{N}}$ **such that** *issuer* ∈ s'.RXissuers **if possible; otherwise stop**
  let *issConf* := s'.RXissuers[*issuer*]
  let *receiverCredPair* ← s'.RXcredentials[*issuer*] **if possible; otherwise stop**
  let *accessToken* := *receiverCredPair*.1 → Pre-shared access token (see Section 2.5.1).
  let *receiverAud* := *receiverCredPair*.2 → SETs' aud claim will be compared against this.
  let *deliveryMethod* ← {urn:ietf:rfc:8936, urn:ietf:rfc:8935}
  if *deliveryMethod* ≡ urn:ietf:rfc:8935 **then** → Push delivery.
    let *epUrl* := s'.RXpushEP
    let *body* := [delivery: [method: *deliveryMethod*, endpoint_url: *epUrl*]]
    let *useDeliveryToken* ← {⊤, ⊥}
    if *useDeliveryToken* ≡ ⊤ **then**
      let *deliveryToken* := ν_{pushTok}
      let *body*[authorization_header] := *deliveryToken*
    else
      let *deliveryToken* := ⊥
  else → Poll delivery.
    let *deliveryToken* := *accessToken* → Poll requests must include this token, see Section 2.6.
    let *body* := ⟨⟩
  let *authZheader* := ⟨Bearer, *accessToken*⟩ → Request must be authorized [15, Section 7].
  let *headers* := [Authorization: *authZheader*]
  let *confEP* := *issConf*[configuration_endpoint]
  let *req* := ⟨HTTPReq, ν_{crStr}, POST, *confEP*.host, *confEP*.path, *confEP*.parameters, *headers*, *body*⟩
  call HTTPS_SIMPLE_SEND([responseTo: CREATE_STREAM, token: *deliveryToken*,
    ↪ expectedAud: *receiverAud*, issuer: *issuer*], *req*, *a*, s')

Algorithm 10 Relation of an SSF Transceiver R^{ssftr} – Send add subject request

→ Send a request to the add subject endpoint of an issuer (for some existing stream).

1: **function** SEND_ADD_SUBJECT_REQUEST(a, s')
2: **let** *streamID* ← $\mathcal{T}_{\mathcal{N}}$ **such that** *streamID* ∈ s'.`RXstreams` **if possible; otherwise stop**
3: **let** *streamConfig* := s'.`RXstreams`[*streamID*]
4: **let** *issuer* := *streamConfig*[`issuer`]
5: **let** *addSubEP* := s'.`RXissuers`[*issuer*][`add_subject_endpoint`]
6: **let** *receiverCredWithAud* ← s'.`RXcredentials`[*issuer*] **if possible; otherwise stop**
7: **let** *accessToken* := *receiverCredWithAud*.1
8: **let** *subject* ← $\mathbb{S}$ → This is an overapproximation (cf. Definition 2).
9: **let** *headers* := [`Authorization`: ⟨`Bearer`, *accessToken*⟩]
10: **let** *body* := [`stream_id`: *streamID*, `subject`: *subject*]
11: **let** *req* := ⟨`HTTPReq`, $\nu_{\text{addSubReq}}$, `POST`, *addSubEP*.`host`, *addSubEP*.`path`, *addSubEP*.`parameters`, ↪ *headers*, *body*⟩
12: **call** HTTPS_SIMPLE_SEND([`responseTo`: `ADD_SUB_REQUEST`], *req*, a, s')

Algorithm 11 Relation of an SSF Transceiver R^{ssftr} – Poll SETs of an existing stream

1: **function** SEND_POLL_REQUEST(a, s')
2: **let** *streamID* ← $\mathcal{T}_{\mathcal{N}}$ **such that**
↪ *streamID* ∈ s'.`RXstreams` ∧ s'.`RXstreams`[*streamID*][`delivery`][`method`] ≡ `urn:ietf:rfc:8936`
↪ **if possible; otherwise stop**
3: **let** *streamConfig* := s'.`RXstreams`[*streamID*]
4: **let** *pollEP* := *streamConfig*[`delivery`][`endpoint_url`]
5: **let** *deliveryToken* := *streamConfig*[`token`] → See Line 22 of Algorithm 7.
6: **let** *headers* := [`Authorization`: ⟨`Bearer`, *deliveryToken*⟩]
7: **let** *req* := ⟨`HTTPReq`, ν_{poll}, `GET`, *pollEP*.`host`, *pollEP*.`path`, *pollEP*.`parameters`, *headers*, ⟨⟩⟩
8: **call** HTTPS_SIMPLE_SEND([`responseTo`: `POLL`, `streamID`: *streamID*], *req*, a, s')

Algorithm 12 Relation of an SSF Transceiver R^{ssftr} – Push SETs of an existing stream

1: **function** SEND_PUSH_REQUEST(a, s')
2: **let** *streamID* ← $\mathcal{T}_{\mathcal{N}}$ **such that**
↪ *streamID* ∈ s'.`TXstreams` ∧ s'.`TXstreams`[*streamID*][`delivery`][`method`] ≡ `urn:ietf:rfc:8935`
↪ **if possible; otherwise stop**
3: **let** *streamConfig* := s'.`TXstreams`[*streamID*]
4: **let** *pushEP* := *streamConfig*[`delivery`][`endpoint_url`]
5: **if** `token` ∈ *streamConfig* **then**
6: **let** *authzHeader* := *streamConfig*[`token`]
7: **let** *headers* := [`Authorization`: *authzHeader*]
8: **else**
9: **let** *headers* := ⟨⟩
10: **let** *set* := CREATE_SET(*streamConfig*, s') → See Algorithm 5.
11: **let** *req* := ⟨`HTTPReq`, ν_{push}, `POST`, *pushEP*.`host`, *pushEP*.`path`, *pushEP*.`parameters`, *headers*, *set*⟩
12: **call** HTTPS_SIMPLE_SEND([`responseTo`: `PUSH`], *req*, a, s')

Algorithm 13 Relation of an SSF Transceiver R^{ssftr} – Processing other messages

→ Any message/event that is not an HTTP(S) or DNS message is processed here. For our model of an SSF Transceiver, we use such messages as stream IDs, thus allowing the attacker to choose the stream IDs for streams of both honest and dishonest receivers.

1: **function** PROCESS_OTHER(m, a, f, s')
2: **let** *streamID* := m → We interpret m as a stream ID chosen by and sent by an attacker process (see also Line 20 of Algorithm 2).
3: **if** *streamID* $\in s'$.`TXstreams` $\vee$ *streamID* $\in^{\langle\rangle}$ s'.`pendingStreamIds` **then**
→ Even though the attacker chooses the stream IDs, we have to prevent duplicate stream IDs as per [15, Section 7.1.1].
4: **stop**
5: **let** s'.`pendingStreamIds` := s'.`pendingStreamIds` $+^{\langle\rangle}$ *streamID*
6: **stop** $\langle\rangle, s'$

B. SSF Web System

Our formal model of the SSF is a Web system as defined in Definition 56.

Definition 4 (SSF Web System). We say that $\mathcal{SSF} := (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is an *SSF Web System with a network attacker*, and define its components as follows:

- $\mathcal{W} = \mathsf{SSFTR} \cup \mathsf{Net}$ consists of the network attacker process (in Net) and a (finite) set of SSF Transceivers SSFTR. We note that DNS servers are subsumed by the network attacker, i.e., DNS is controlled by the attacker, and are therefore not modeled explicitly.
- $\mathcal{S} = \emptyset$, and hence, script's relation is empty.
- E^0 is an infinite set of trigger events, with infinitely many events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for each address $a \in \mathsf{IPs}$.

C. Formal Security Properties

In this appendix, we show the formal security properties and refer to Section 4 for a detailed informal description. For formalizing the properties, we first introduce the following definitions:

Issuance of Audience Identifier. Within the model, a Transmitter identifies a Receiver by the Receiver's receiver credential used to authorize/authenticate the Receiver at the Transmitter's management API endpoints. These receiver credentials are modeled as described in Section 2.5.1 and Appendix A.3 and consequently, we define issuance of an audience value by a Transmitter to a Receiver as follows:

Definition 5 (Audience Identifier of Receiver at Transmitter). We say that $aud \in \mathbb{S}$ *is an audience identifier of* $ssftr_r \in \mathsf{SSFTR}$ *at* $ssftr_t \in \mathsf{SSFTR}$ *under the issuer identifier* iss within a run $\rho = ((S^0, E^0, N^0), ...)$ of an SSF Web system $\mathcal{SSF}$, if

(i) $iss \in issIDs^{ssftr_t}$, $\exists rc \in \mathsf{ReceiverCreds}$ such that

(ii) $S^0(ssftr_r).\texttt{RXcredentials}[iss][rc] = aud$, and

(iii) $S^0(ssftr_t).\texttt{TXreceiverCreds}[iss][rc][\texttt{aud}] = aud$.

Issuance of SETs. The following definition captures that a Transmitter issues a SET in a certain processing step.

Definition 6 (SET Issued by Transmitter). We say that $set \in \mathcal{T}_{\mathcal{N}}$ *is a SET that has been issued by* $ssftr_t \in \mathsf{SSFTR}$ *in processing step* P *in a run* ρ (of an SSF web system $\mathcal{SSF}$), if

(i) $P = (S^p, E^p, N^p) \xrightarrow[ssftr_t \to E^P_{\text{out}}]{e^P_{\text{in}} \to ssftr_t} (S^{p+1}, E^{p+1}, N^{p+1})$, and

(ii) in P, $ssftr_t$ executes Algorithm 5 ($\mathsf{CREATE\_SET}$), and, with *signedSet* being the return value of that function, it holds true that $\mathsf{extractmsg}(signedSet) \equiv set$.

C.1. Configuration Discovery Integrity

Definition 7 (Configuration Discovery Integrity). We say that an SSF Web System with a network attacker $\mathcal{SSF}$ provides *configuration discovery integrity* iff for every run ρ of $\mathcal{SSF}$, every configuration (S, E, N) in ρ, every SSF transceiver $ssftr \in \mathsf{SSFTR}$ that is honest in S, every issuer identifier $iss \in \mathsf{IssIDs}$, if $t := \mathsf{dom}^{-1}(iss.\texttt{host})$ is honest in S and $iss \in S(ssftr).\texttt{RXissuers}$, then, with $config := S(ssftr).\texttt{RXissuers}[iss]$, all of the following hold true:

(I) $config[\texttt{issuer}] \equiv iss$

(II) $config[\texttt{jwks\_uri}] \equiv \langle \texttt{URL}, \texttt{S}, iss.\texttt{host}, /\texttt{jwks}, \langle\rangle, \bot \rangle$

(III) $config[\texttt{configuration\_endpoint}] \equiv \langle \texttt{URL}, \texttt{S}, iss.\texttt{host}, /\texttt{configure-stream}, \langle\rangle, \bot \rangle$

(IV) $config[\texttt{add\_subject\_endpoint}] \equiv \langle \texttt{URL}, \texttt{S}, iss.\texttt{host}, /\texttt{add-subject}, \langle\rangle, \bot \rangle$

(V) $config[\texttt{authorization\_schemes}] \equiv \langle [\texttt{spec\_urn: urn:ietf:rfc:6749}] \rangle$

(VI) $\texttt{jwks} \in config \Rightarrow config[\texttt{jwks}] \equiv \mathsf{pub}(\mathsf{signkey}(iss))$

C.2. Session Integrity for SETs

Definition 8 (SET Session Integrity). We say that an SSF Web System with a network attacker $\mathcal{SSF}$ provides *session integrity for SETs* iff for every run ρ of $\mathcal{SSF}$, every configuration (S, E, N) in ρ, every pair of SSF transceivers $ssftr_r, ssftr_t \in \mathsf{SSFTR}$ that are honest in S, every pair of terms $set, x \in \mathcal{T}_{\mathcal{N}}$, if

(1) $set \in^{\langle\rangle} S(ssftr_r).\texttt{RXsets}[x]$ (i.e., $ssftr_r$ accepted set), and

(2) $ssftr_t = \mathsf{dom}^{-1}(set[\texttt{iss}].\texttt{host})$,

then all of the following hold true:

(I) set is a SET that has been issued by $ssftr_t$ in a processing step P prior to the configuration (S, E, N) (see Definition 6).

(II) $set[\texttt{aud}]$ is an audience identifier of $ssftr_r$ at $ssftr_t$ under the issuer identifier $set[\texttt{iss}]$ (see Definition 5).

C.3. Confidentiality of SETs

Definition 9 (SET Confidentiality). We say that an SSF Web System with a network attacker $\mathcal{SSF}$ provides *confidentiality for SETs* iff for every run ρ of $\mathcal{SSF}$, every configuration (S, E, N) in ρ, every pair of SSF transceivers $ssftr_r, ssftr_t \in \mathsf{SSFTR}$ that are honest in S, and every term $set \in \mathcal{T}_{\mathcal{N}}$, if

(1) $ssftr_t = \mathsf{dom}^{-1}(set[\texttt{iss}].\texttt{host})$, and

(2) $\mathsf{checksig}(set, \mathsf{pub}(\mathsf{signkey}(set[\texttt{iss}]))) \equiv \top$, and

(3) $set[\texttt{aud}]$ is an audience identifier of $ssftr_r$ at $ssftr_t$ under the issuer identifier $set[\texttt{iss}]$ in ρ (see Definition 5),

then we have that the term set is not derivable by the attacker, i.e., for the attacker process att, it holds true that $set \notin d_\emptyset(S(att))$.

C.4. Authorization

Definition 10 (Authorization). We say that an SSF Web System with a network attacker $\mathcal{SSF}$ provides *authorization* iff for every run ρ of $\mathcal{SSF}$, every processing step $P = (S^p, E^p, N^p) \rightarrow (S^{p+1}, E^{p+1}, N^{p+1})$ in ρ, every SSF transceiver $ssftr_t \in \mathsf{SSFTR}$ that is honest in S^{p+1}, every term $set \in \mathcal{T}_\mathcal{N}$, if set is a SET that has been issued by $ssftr_t$ in P (see Definition 6), then all of the following hold true:

(I) $\exists ssftr_r \in \mathsf{SSFTR}$ such that $set[\texttt{aud}]$ is an audience identifier of $ssftr_r$ at $ssftr_t$ under the issuer identifier $set[\texttt{iss}]$ (see Definition 5).

(II) $\exists ssftr'_r \in \mathsf{SSFTR}, rc \in \mathsf{sharedSecrets}(ssftr'_r, set[\texttt{iss}])$ s.t. $set[\texttt{sub\_id}] \in \mathsf{permittedSubjects}(rc)$.

(III) If $ssftr'_r$ is honest in S^{p+1}, then $ssftr_r = ssftr'_r$.

D. Proofs

D.1. Helper Lemmas

Lemma 1 (Correctness of Reference and Request). For any run ρ of an SSF Web System with a network attacker $\mathcal{SSF}$, every processing step $P = (S^P, E^P, N^P) \rightarrow (S^{P'}, E^{P'}, N^{P'})$ in ρ, every $ssftr \in \mathsf{SSFTR}$ being honest in S^P, it holds true that if $ssftr$ calls PROCESS_HTTPS_RESPONSE in P with *reference* being the second and *request* being the third input argument, then there exists a previous processing step in which $ssftr$ calls HTTPS_SIMPLE_SEND with *reference* being the first and *request* being the second input argument.

Proof. Since $ssftr$ is honest in S^P, it only calls PROCESS_HTTPS_RESPONSE in the generic HTTPS server algorithm in Line 26 of Algorithm 31. There, the values *reference* and *request* are taken from $S^P(ssftr).\texttt{pendingRequests}$ in Line 19 of Algorithm 31.

Since `pendingRequests` is initially empty (Definition 3), $ssftr$ must have added these values to `pendingRequests` in a previous processing step $O = (S^O, E^O, N^O) \rightarrow (S^{O'}, E^{O'}, N^{O'})$ by executing Line 15 of Algorithm 31, as this is the only location where an SSF Transceiver adds entries to `pendingRequests`.

In O, $ssftr$ takes *reference* and *request* from $S^O(ssftr).\texttt{pendingDNS}$ in Lines 13 and 14 of Algorithm 31, respectively.

Once again, since `pendingDNS` is initially empty (Definition 3), $ssftr$ must have added these values to `pendingDNS` earlier. Values are only added to `pendingDNS` in Line 2 of Algorithm 26, where the *reference* and *request* values are the input arguments of HTTPS_SIMPLE_SEND – and immediately afterwards, in Line 3 of Algorithm 26, the current processing step stops. Thus, in some processing step prior to O, $ssftr$ must have called HTTPS_SIMPLE_SEND with *reference* being the first and *request* being the second input argument. ∎

Lemma 2 (Origin of Values in `RXissuers`). For any run ρ of an SSF Web System with a network attacker $\mathcal{SSF}$, any configuration (S, E, N) in ρ, any SSF transceiver $ssftr \in \mathsf{SSFTR}$ that is honest in S, any term $iss \in \mathcal{T}_\mathcal{N}$, if $iss \in S(ssftr).\texttt{RXissuers}$, then

(I) the value stored in $S(ssftr).\texttt{RXissuers}[iss]$ (except for a possible `jwks` key inside this value) was stored there by executing Line 6 of Algorithm 7, and

(II) if $\texttt{jwks} \in S(ssftr).\texttt{RXissuers}[iss]$, then the value stored in $S(ssftr).\texttt{RXissuers}[iss][\texttt{jwks}]$ was stored there by executing Line 11 of Algorithm 7, and, when executing this line, we already had $iss \in S(ssftr).\texttt{RXissuers}$.

Proof. Since `RXissuers` is initially empty (Definition 3), $iss \in S(ssftr).\texttt{RXissuers}$ implies that during some processing step P prior to (S, E, N) in ρ one of Lines 6 and 11 of Algorithm 7 must have been executed as these are the only places where *ssftr* stores anything in `RXissuers`.

Line 11 of Algorithm 7 (which is part of PROCESS_HTTPS_RESPONSE) is only executed when processing the response to a JWKs request, i.e., with Lemma 1 and Line 10 of Algorithm 7, we have that the corresponding request must have been sent earlier (prior to P) using HTTPS_SIMPLE_SEND with a *reference* value such that $reference[\texttt{responseTo}] \equiv \texttt{JWKS}$. The only place in which *ssftr* calls HTTPS_SIMPLE_SEND with such a *reference* value is Line 9 of Algorithm 7. There, *ssftr* just stored a value in $S(ssftr).\texttt{RXissuers}[iss']$ and also added iss' to the *reference* value under the key `issuer` – this exact value iss' is also used as a key in Line 11 of Algorithm 7. Hence, we have proven (II).

We already established that a new key iss' is only added to $S(ssftr).\texttt{RXissuers}$ in Line 6 of Algorithm 7. Furthermore, the only value within such an entry that is modified by Line 11 of Algorithm 7 is stored under a key `jwks`. Hence, we also get (I). ∎

Lemma 3 (Properties of a SET Issued by Honest Transmitter). For any run ρ of an SSF Web System with a network attacker $\mathcal{SSF}$, any processing step $P = (S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ in ρ, any SSF transceiver $ssftr \in \mathsf{SSFTR}$ that is honest in S^{i+1}, any term $set \in \mathcal{T}_{\mathcal{N}}$, if *ssftr* issued SET *set* in P (see Definition 6), then all of the following hold true:

(I) $set[\texttt{iss}] \in issIDs^{ssftr}$ (see Definition 1).

(II) $signedSet \equiv \mathsf{sig}(set, \mathsf{signkey}(set[\texttt{iss}]))$ (*signedSet* as in (ii) of Definition 6).

Proof. **(I)** The issuer value in the issued SET *set* is taken from the *streamConfig* given to CREATE_SET as the first argument (Lines 2 and 6 of Algorithm 5). The function CREATE_SET is only called in two places, namely Line 7 of Algorithm 4 and Line 10 of Algorithm 12. In both cases, *streamConfig* is taken from the `TXstreams` state subterm. Items in `TXstreams` (and their `iss` value) are only added in Line 28 of Algorithm 2 (Line 17 of Algorithm 3 never adds entries to `TXstreams` due to the check in Line 11 of Algorithm 3), where the `iss` value is set to the value from Line 2 of Algorithm 2. Since Algorithm 2 is only called in Line 22 of Algorithm 1, we have that the message m processed by *ssftr* during P must be an HTTPS request for a domain of *ssftr*, i.e., $\mathsf{dec_a}((, m), k).1.\texttt{host} \in \mathsf{dom}(ssftr)$ (otherwise, decryption would fail in Line 7 of Algorithm 31 due to Definition 3 and the definition of $tlskeys^p$ in Appendix A.3).

Hence, due to Definition 1, we have $set[\texttt{iss}] \in issIDs^{ssftr}$.

(II) The return value of CREATE_SET is the *signedSet* created in Line 8 of Algorithm 5. The key used to sign *signedSet* there is taken from the `TXjwks` state subterm of *ssftr* using the key $set[\texttt{iss}]$ (see Lines 6f. of Algorithm 5). Note that an honest Transceiver never changes its `TXjwks` state subterm. Hence, $S^i(ssftr).\texttt{TXjwks}[set[\texttt{iss}]] = s_0^{ssftr}.\texttt{TXjwks}[set[\texttt{iss}]] = \mathsf{signkey}(set[\texttt{iss}])$ (Definition 3, and from (I) we have $set[\texttt{iss}] \in issIDs^{ssftr}$). ∎

Lemma 4 (Secrecy of Signing Keys). For any run ρ of an SSF Web System with a network attacker $\mathcal{SSF}$, any configuration (S, E, N) in ρ, any SSF transceiver $ssftr \in \mathsf{SSFTR}$ that is honest in S, any process $p \in \mathcal{W}$ different from *ssftr* (i.e., $p \neq ssftr$), any issuer identifier $iss \in \mathsf{IssIDs}$ with $ssftr = \mathsf{dom}^{-1}(iss.\texttt{host})$, it holds true that $\mathsf{signkey}(iss) \notin d_\emptyset(S(p))$, i.e., the signing key is only derivable by *ssftr*.

PROOF. Let $k := \mathsf{signkey}(iss)$ be the private signature key of *iss*. In the initial state of ρ, only *ssftr* stores k in its state, and no other process contains k in its state (i.e., not even as a subterm): The private signing keys are distributed according to the signkey mapping (see Appendix A.3). From Definition 1, it follows that $iss \in issIDs^{ssftr}$, and according the definition of initial states of SSF Transceivers (Definition 3), k is only stored in $s_0^{ssftr}.\texttt{TXjwks}[iss]$.

ssftr accesses this key only in two locations:

- In Line 17 of Algorithm 1, *ssftr* processes a request to the `/jwks` endpoint and responds with the public verification key.
- In Line 7 of Algorithm 5, *ssftr* creates the signature of a SET.

For both cases, the equational theory (Definition 13) does not allow the extraction of the key from the resulting terms. ∎

Lemma 5 (Properties of HTTPS_SIMPLE_SEND). For any run $\rho = ((S^0, E^0, N^0), ...)$ of an SSF Web System with a network attacker $\mathcal{SSF}$, any SSF Transceiver $ssftr \in \mathsf{SSFTR}$ that is honest in $S^{p''+1}$, any string $reqType \in \mathbb{S}$, any processing step

$$P = (S^p, E^p, N^p) \xrightarrow[ssftr \to E^P_{\text{out}}]{e^P_{\text{in}} \to ssftr} (S^{p+1}, E^{p+1}, N^{p+1})$$

in ρ, we have that if

(1) *ssftr* calls HTTPS_SIMPLE_SEND (Algorithm 26) during P with a *reference* and *message* as the first two arguments, and

(2) $reference[\texttt{responseTo}] \equiv reqType$,

then all of the following hold true:[4]

(I) Contents (i.e., subterms) from *reference* are not included in any event emitted by *ssftr* until a processing step $P' = s^{p'} \xrightarrow{e^{P'}_{\text{in}} \to ssftr} s^{p'+1}$ after P in ρ during which *ssftr* executes the function PROCESS_HTTPS_RESPONSE with *reference* and *message* as the second and third argument. Furthermore, no copies of *reference* stored in *ssftr*'s state as part of the HTTPS_SIMPLE_SEND execution are left after P'.

(II) Contents (i.e., subterms) from *message* are not included in any event emitted by *ssftr* until a processing step $P'' = s^{p''} \xrightarrow{e^{P''}_{\text{in}} \to ssftr} s^{p''+1}$ after P in ρ during which *ssftr* executes Lines 10ff. of Algorithm 31 with *reference* in Line 13 of Algorithm 31 and *message* in Line 14 of Algorithm 31. The (only) event emitted in P'' matches $\mathsf{enc_a}(\langle *, message\rangle, k)$ with $k = S^0(ssftr).\texttt{keyMapping}[message.\texttt{host}]$.

(III) Apart from the event emitted in P'', contents from *message* are not included in any event emitted by *ssftr* until P', and no copies of *message* stored in *ssftr*'s state as part of the HTTPS_SIMPLE_SEND execution are left after P'.

[4] By "contents from x are not included...", we denote inclusion in a data-flow sense. E.g., events emitted by *ssftr* between P and P' might still include subterms of *reference*, but these values are not read from *reference* or its copies created in *ssftr*'s state as part of the HTTPS_SIMPLE_SEND execution and further processing.

PROOF. **(I):** The *reference* value given to HTTPS_SIMPLE_SEND as the first argument is stored in *ssftr*'s `pendingDNS` state subterm in Line 2 of Algorithm 26 (and not included in any emitted events during P). The only place in which values from that subterm are accessed is Lines 10ff. of Algorithm 31, where the *reference* value is stored in another state subterm `pendingRequests` (Line 15) and removed from `pendingDNS` (Line 17).

Values in `pendingRequests` are only accessed in Lines 19ff. of Algorithm 31, where the *reference* value is removed from `pendingRequests` and handed to PROCESS_HTTPS_RESPONSE – along with the *message* that moved through the state subterms together with *reference*. Hence, this describes processing step P', and since the *reference* value was only moved through *ssftr*'s state, it cannot be included in any event emitted between P and P'.

(II): The *message* value given to HTTPS_SIMPLE_SEND as the second argument is stored in *ssftr*'s `pendingDNS` state subterm in Line 2 of Algorithm 26 (and not included in any emitted events during P). The only place in which values from that subterm are accessed is during a later processing step, say Q, in Lines 10ff. of Algorithm 31, i.e., *message* or parts of it cannot be included in any event emitted between P and Q. Furthermore, since *message* and *reference* were stored under the same key in P, they are also retrieved together in Q.

In Lines 10ff. of Algorithm 31, *message* and a fresh nonce are encrypted under the key $k = Q(ssftr).\texttt{keyMapping}[message.\texttt{host}]$ (Line 16 of Algorithm 31); the resulting ciphertext matches $\mathsf{enc_a}(\langle *, message\rangle, k)$ and is the payload of the only event emitted during Q (Line 18 of Algorithm 31).

An SSF Transceiver never modifies its `keyMapping` state subterm, so $Q(ssftr).\texttt{keyMapping} = S^0(ssftr).\texttt{keyMapping}$.

Finally, be choosing $P'' := Q$, we get (II).

(III): As noted in the proof for (I), the *message* value is passed through the `pendingDNS` and `pendingRequests` state subterms alongside the *reference* value – hence, the same argumentation and conclusion applies. ∎

Lemma 6 (Receiver Stream Configuration Integrity). For any run $\rho = ((S^0, E^0, N^0), ...)$ of an SSF Web System with a network attacker $\mathcal{SSF}$, any configuration (S, E, N) in ρ, any pair of SSF transceivers $ssftr_t, ssftr_r \in \mathsf{SSFTR}$ with both being honest in S, any issuer identifier $iss \in issIDs^{ssftr_t}$, and any $streamID \in \mathcal{T}_{\mathcal{N}}$, we have that if

(1) $streamID \in S(ssftr_r).\texttt{RXstreams}$, and

(2) $S(ssftr_r).\texttt{RXstreams}[streamID][\texttt{issuer}] \equiv iss$, and

(3) $S(ssftr_r).\texttt{RXstreams}[streamID][\texttt{delivery}][\texttt{method}] \equiv \texttt{urn:ietf:rfc:8936}$, then

$S(ssftr_r).\texttt{RXstreams}[streamID][\texttt{delivery}][\texttt{endpoint\_url}] \sim \langle \texttt{URL}, \texttt{S}, iss.\texttt{host}, /\texttt{poll}, [\texttt{rand}\colon *], \bot\rangle$.

PROOF. Stream configurations in $ssftr_r$'s state are only stored in Line 19 of Algorithm 7. I.e., due to precondition (2) and Lemma 1, there must be a processing step $P = s^p \to s^{p+1}$ (with s^p prior to (S, E, N) in ρ) during which $ssftr_r$ called HTTPS_SIMPLE_SEND with a reference value that contained value `CREATE_STREAM` under key `responseTo` and value *iss* under key `issuer` (see Lines 13ff. of Algorithm 7).

Such a call to HTTPS_SIMPLE_SEND only occurs in Line 24 of Algorithm 9, where the create stream request *req* given to HTTPS_SIMPLE_SEND is addressed to a URL *confEP*. That URL is taken from an entry in the `RXissuers` state subterm (Lines 3 and 22 of Algorithm 9); said entry is stored there under the key *iss* (otherwise, the reference's `issuer` value would not be *iss*).

Hence, we can apply Lemma 9 which implies $confEP.\texttt{host} \equiv iss.\texttt{host}$ and $confEP.\texttt{path} \equiv /\texttt{configure-stream}$. Due to Lemma 5 and Lemma 16, we also get that only $ssftr_t$ can decrypt and reply to *req*.

Such a response to *req* can only be created by $ssftr_t$ in Algorithm 2 (due to Lines 20ff. of Algorithm 1). Since precondition (3) and the fact that an SSF Transceiver never overwrites the `delivery` section of a stream configuration imply that $ssftr_t$'s response to *req* contains delivery method `urn:ietf:rfc:8936`, $ssftr_t$ must have executed Lines 10ff. of Algorithm 2 when creating the response. In particular, it must have executed Line 12 of Algorithm 2, included the value created there (that obviously matches $\langle \texttt{URL}, \texttt{S}, iss.\texttt{host}, /\texttt{poll}, [\texttt{rand}\colon *], \bot\rangle$) in its response, and that value is subsequently stored by $ssftr_r$ when processing $ssftr_t$'s response in Line 19 of Algorithm 7. ∎

Lemma 7 (Secrecy of Access Tokens). For any run $\rho = ((S^0, E^0, N^0), ...)$ of an SSF Web System with a network attacker $\mathcal{SSF}$, any configuration (S, E, N) in ρ, any pair of SSF transceivers $ssftr_t, ssftr_r \in \mathsf{SSFTR}$ with both being honest in S, any process $p \in \mathcal{W}$ different from $ssftr_t, ssftr_r$ (i.e., $p \neq ssftr_t, p \neq ssftr_r$), we have that any nonce $rc \in \mathsf{sharedSecrets}(ssftr_r, iss)$ for $iss \in issIDs^{ssftr_t}$ is only derivable by $ssftr_t$ and $ssftr_r$, i.e., $rc \notin d_\emptyset(S(p))$.

Proof.

(A) **In the initial state of ρ, only $ssftr_t$ and $ssftr_r$ can derive rc.** From the preconditions, we have $rc \in \mathsf{sharedSecrets}(ssftr_r, iss)$ for $iss \in issIDs^{ssftr_t}$, therefore, Definition 3 immediately gives us $rc \in S^0(ssftr_t).\texttt{TXreceiverCreds}[iss]$ as well as $rc \in S^0(ssftr_r).\texttt{RXcredentials}[iss]$. Furthermore, the definition of sharedSecrets (see Appendix A.3) implies that no other process (and in particular p) *knows* (Definition 63) rc in S^0.

(B) rc **cannot leak from `TXreceiverCreds`.** Values stored in `TXreceiverCreds` are only accessed in the following places:

Line 6 of Algorithm 2 Only the `aud` value *within* an entry in $S^j(ssftr_t).\texttt{TXreceiverCreds}[iss]$ is accessed, i.e., rc cannot leak from this access.

Line 8 of Algorithm 3 Like above, only values *within* an entry are accessed (the `aud` and `subjects` values), i.e., rc cannot leak from this access.

(C) rc **cannot leak from `RXcredentials`.** Values stored in `RXcredentials` are only accessed by $ssftr_r$ in Line 4 of Algorithm 9 and Line 6 of Algorithm 10. We show that rc cannot leak in either of those cases in (D) and (E) below. Hence, we have $rc \notin d_\emptyset(S(p))$.

(D) In **Line 4 of Algorithm 9**, the shared secret rc is extracted into a variable *accessToken*. This variable is used in (1) Line 18 of Algorithm 9, where it is stored into a variable *deliveryToken* which is only used in the reference value passed to HTTPS_SIMPLE_SEND in Line 24 of Algorithm 9 under key `token`. Furthermore, *accessToken* is accessed in (2) Line 20 of Algorithm 9 where the shared secret is copied to the constructed request's Authorization header and thus also passed to HTTPS_SIMPLE_SEND in Line 24 of Algorithm 9.

The values passed to HTTPS_SIMPLE_SEND (Algorithm 26) as part of the reference term do not leak until processing of the corresponding response, and the constructed request itself is only accessed when sending the request after DNS resolution and processing of the corresponding response (Lemma 5). Hence, we look at those two cases:

(D.i) **Processing the `CREATE_STREAM` response.** The reference value from Line 24 of Algorithm 9 contains the value `CREATE_STREAM` in its `responseTo` field. Responses to such requests are only processed in Lines 13ff. of Algorithm 7 (cf. Lemma 1), where the `token` key is only used in Line 22 of Algorithm 7 to store its value (i.e., rc) in the new stream's configuration under key `token`. A value under this key is only accessed in Line 5 of Algorithm 11, where it is included in the Authorization header of a polling request and thus passed to HTTPS_SIMPLE_SEND in Line 8 of Algorithm 11. As per Lemma 5, this value is not used except when sending the request and processing its response.

(D.i.1) **Processing the `POLL` response.** Since the call to HTTPS_SIMPLE_SEND contains the `responseTo` value `POLL`, response processing only happens in Lines 24ff. of Algorithm 7, where the request value is not used, i.e., *rc* cannot leak there.

(D.i.2) **Sending the `POLL` request.** From Lemma 5, we have that the request is sent encrypted with the TLS key associated with the host selected in Line 7 of Algorithm 11. That host is taken from the same stream configuration stored in $ssftr_r$'s `RXstreams` state subterm as *rc*. Since that stream configuration's `issuer` value must be *iss* (otherwise, $ssftr_r$ would not have stored *rc* in that stream configuration in Line 22 of Algorithm 7), we can apply Lemma 6. Therefore, the host in question is *iss*.`host`, i.e., the `POLL` request is encrypted for a TLS key of $ssftr_t$, so only $ssftr_t$ can decrypt the `POLL` request (Lemma 16).

From Lemma 6, we also know that the path value of that request is `/poll`. HTTPS requests with this path are only processed in Algorithm 4 (due to Line 28 of Algorithm 1). There, the *rc* value contained in the `POLL` request's Authorization header is not stored or emitted, hence, *rc* cannot leak there.

(D.ii) **Sending the `CREATE_STREAM` request.** Since *rc* is (only) stored in `RXcredentials` under key *iss*, we have (from Lemma 9) that the configuration endpoint selected in Line 22 of Algorithm 9 is for a domain of $ssftr_t$. Hence, the create stream request (with *rc* in the Authorization header) can only be decrypted by $ssftr_t$ (Lemma 16), and the path of that request is `/configure-stream`. Such a request is processed by $ssftr_t$ in Algorithm 2 (due to Lines 20ff. of Algorithm 1), where the Authorization header is not emitted, but stored in $ssftr_t$'s state subterm `TXstreams` under the key `token` (see Line 27 of Algorithm 2).

The value stored under key `token` can only be accessed (except for comparisons with incoming messages) in Line 6 of Algorithm 12. However, that line can only be reached for streams with push delivery – in which case the `token` value does not originate from an Authorization header, but from the message body of a create stream request (see Line 17 of Algorithm 2). Since we are currently looking at *rc* values originating from $ssftr_r$'s state subterm `RXcredentials`, and values from that state subterm can never end up in the message body of a create stream request (see Lines 12ff. of Algorithm 9), we conclude that *rc* cannot leak in Line 6 of Algorithm 12 (because *rc* will never be stored under the key `token` if Line 6 of Algorithm 12 can be reached).

(E) In **Line 6 of Algorithm 10**, *rc* (as variable *accessToken*) is subsequently used in the Authorization header of an add subject request, which is handed over to HTTPS_SIMPLE_SEND in Line 12 of Algorithm 10. The endpoint to which this request is sent is the add subject endpoint of the issuer identifier for which *rc* is stored in $ssftr_r$'s state, i.e., that issuer identifier is *iss*. Hence, with Lemma 9 instead of Lemma 6 and path `/add-subject` instead of `/poll`, we can reason as above in (D.i.2): processing of the add subject response does not leak *rc* (since such responses are ignored), and processing of the add subject request at $ssftr_t$ (who is the only process that can decrypt the add subject request) happens only in Algorithm 3 (due to Line 26 of Algorithm 1), where *rc* is neither stored, nor emitted, i.e., cannot leak. ∎

Lemma 8 (Uniqueness of Credentials at Transmitter). For any run $\rho = ((S^0, E^0, N^0), ...)$ of an SSF Web System with a network attacker $\mathcal{SSF}$, any SSF transceiver $ssftr_t \in \mathsf{SSFTR}$, any issuer identifier $iss \in issIDs^{ssftr_t}$, any terms $rc, rc' \in \mathcal{T}_{\mathcal{N}}$, if

(1) $S^0(ssftr_t).\texttt{TXreceiverCreds}[iss][rc][\texttt{aud}] = S^0(ssftr_t).\texttt{TXreceiverCreds}[iss][rc'][\texttt{aud}]$ and

(2) $S^0(ssftr_t).\texttt{TXreceiverCreds}[iss][rc][\texttt{aud}] \neq \langle\rangle$,

then $rc = rc'$.

PROOF. We assume that $rc \neq rc'$. From preconditions (1) and (2), together with Definition 3, it follows that $\exists ssftr_r, ssftr_{r'} \in \mathsf{SSFTR}$ such that

$$\begin{aligned} &rc \in \mathsf{sharedSecrets}(ssftr_r, iss) \wedge \\ &rc' \in \mathsf{sharedSecrets}(ssftr_{r'}, iss) \wedge \\ &\mathsf{receiverAudience}(rc) = \mathsf{receiverAudience}(rc'). \end{aligned}$$

Furthermore, the condition in Appendix A.2.1 implies that $\exists iss_1, iss_2 \in \mathsf{IssIDs}$, $\exists ssftr_{\tilde{r}}, ssftr_{\tilde{r}'} \in \mathsf{SSFTR}$ such that

$$\begin{aligned} &rc \in \mathsf{sharedSecrets}(ssftr_{\tilde{r}}, iss_1) \wedge \\ &rc' \in \mathsf{sharedSecrets}(ssftr_{\tilde{r}'}, iss_2) \wedge \\ &iss_1 \neq iss_2. \end{aligned}$$

From the condition in Appendix A.3, it follows that $iss_1 = iss = iss_2$, which is a contradiction to $iss_1 \neq iss_2$; i.e., the initial assumption $rc \neq rc'$ must be false, hence proving $rc = rc'$. ∎

D.2. Configuration Discovery Integrity

Lemma 9 (Configuration Discovery Integrity). Every SSF Web System with a network attacker $\mathcal{SSF}$ provides configuration discovery integrity (Definition 7).

PROOF. **(I)-(V)**: As shown in Lemma 2, there exists a processing step $P = (S^p, E^p, N^p) \rightarrow (S^{p\prime}, E^{p\prime}, N^{p\prime})$ in which *ssftr* added *config* to `RXissuers` by executing Line 6 of Algorithm 7. Let m be the (decrypted) HTTP response that *ssftr* processes in P in Algorithm 7 (i.e., the first input argument of the function). From Lines 3 and 6 of Algorithm 7, we get $S^{p\prime}(ssftr).\texttt{RXissuers}[iss][\texttt{issuer}] \equiv m.\texttt{body}[\texttt{issuer}] \equiv iss$ (note that Lemma 2 also implies that this value is not changed once it is stored in the `RXissuers` subterm).

Let *reference* and *request* be the second and third input arguments of the call of Algorithm 7 in P. As Line 6 of Algorithm 7 is executed in P, it follows that $reference[\texttt{responseTo}] \equiv \texttt{DISCOVERY}$ (Line 2 of Algorithm 7).

From Lemma 1, it follows that the corresponding HTTPS request was sent by calling Line 8 of Algorithm 8 in a processing step N prior to P (as this is the only location where *ssftr* calls HTTPS_SIMPLE_SEND with the `DISCOVERY` reference value). From the successful check done in Line 4 of Algorithm 7 in P, it follows that $reference[\texttt{iss}] \equiv iss$. Therefore, for the request *req* sent as a result of calling HTTPS_SIMPLE_SEND in N,[5] it holds true that $req.\texttt{host} \equiv iss.\texttt{host}$ and $req.\texttt{path} \equiv \texttt{/.well-known/ssf-configuration}$ (see Lines 6f. of Algorithm 8).

We can now apply Lemma 5 which, together with Definition 3 and Appendix A.3, tells us that *req* – once it is sent by *ssftr*– is encrypted asymmetrically with the public TLS key associated with the domain $iss.\texttt{host}$, i.e., $\mathsf{pub}(\mathsf{tlskey}(iss.\texttt{host}))$.

From Appendix A.3, it follows that $\langle iss.\texttt{host}, \mathsf{tlskey}(iss.\texttt{host})\rangle \in tlskeys^t$ (note that process $t = \mathsf{dom}^{-1}(iss.\texttt{host})$ is honest due to the preconditions of Definition 7). From Lemma 16, it follows that t created and emitted the HTTPS response m from above.

As the HTTPS request is sent to the `/.well-known/ssf-configuration` path, t processes the request in Line 2 of Algorithm 1. The *config* value stored by *ssftr* in P is the body of t's response, i.e., $m.\texttt{body}$ (Line 6 of Algorithm 7). Conditions (II)-(IV) of Definition 7 now follow directly from Lines 6ff.

[5] Note that *req* is not emitted during N, but at some point between N and P (see the proof of Lemma 1).

of Algorithm 1, where the `jwks_uri`, `configuration_endpoint`, `add_subject_endpoint`, and `authorization_schemes` values are chosen as in the postconditions.

(VI): Let `jwks` $\in$ *config*. As shown in Lemma 2, there exists a processing step $Q = (S^q, E^q, N^q) \to (S^{q\prime}, E^{q\prime}, N^{q\prime})$ in which *ssftr* adds this value by executing Line 11 of Algorithm 7.

As shown in Lemma 1, there exists a processing step $R = (S^r, E^r, N^r) \to (S^{r\prime}, E^{r\prime}, N^{r\prime})$ prior to Q in which *ssftr* calls HTTPS_SIMPLE_SEND with the value *reference*$'$ being the first input argument such that *reference*$'$[`responseTo`] $\equiv$ `JWKS` and *reference*$'$[`issuer`] $\equiv$ *iss*. *ssftr* calls HTTPS_SIMPLE_SEND with this `responseTo` value only in Line 9 of Algorithm 7. In R, *ssftr* calls the HTTPS_SIMPLE_SEND function with a request *req*$'$ such that

$$req'.\texttt{host} \equiv S^r(ssftr).\texttt{RXissuers}[iss][\texttt{jwks\_uri}].\texttt{host}$$

and

$$req'.\texttt{path} \equiv S^r(ssftr).\texttt{RXissuers}[iss][\texttt{jwks\_uri}].\texttt{path}$$

As shown for postcondition (II), the host value is *iss*.`host` and the path is `/jwks`. Similar to above, the corresponding response (that is processed in Q and stored into $S^{q\prime}$.`RXissuers`[*iss*][`jwks`]) was created by t by executing Lines 13ff. of Algorithm 1 in some processing step $U = (S^u, E^u, N^u) \to (S^{u\prime}, E^{u\prime}, N^{u\prime})$, where t responds with the value $\mathsf{pub}(S^u(ssftr).\texttt{TXjwks}[iss])$. From the successful check in Line 15 of Algorithm 1 and Definition 3, and as t never changes its `TXjwks` state subterm, it follows that $iss \in s_0^t.\texttt{TXjwks}$ and the body of the response, i.e., what gets stored into `RXissuers`[*iss*][`jwks`] during Q, is $\mathsf{pub}(\mathsf{signkey}(iss))$. ∎

D.3. Session Integrity for SETs

Lemma 10 (SET Session Integrity). Every SSF Web System with a network attacker $\mathcal{SSF}$ provides SET session integrity (Definition 8).

Proof. **(I)**:

Initially, the `RXsets` state subterm of $ssftr_r$ is empty (see Definition 3), i.e., there is a processing step $Q = (S^q, E^q, N^q) \to (S^{q\prime}, E^{q\prime}, N^{q\prime})$ prior to (S, E, N) in which $ssftr_r$ adds *set* to its `RXsets` state subterm. An honest SSF transceiver modifies its `RXsets` state subterm only in Line 14 of Algorithm 6 and Line 38 of Algorithm 7. We consider both cases separately:

- Case 1: Line 14 of Algorithm 6: In this case, $ssftr_r$ is processing a push request containing a set, and an honest transceiver only adds SETs if the check of the signature in Line 7 of Algorithm 6 was done successfully. $ssftr_r$ verifies the signature using the key $verificationKey := S^q(ssftr_r).\texttt{RXissuers}[iss][\texttt{jwks}]$, with $iss \equiv set[\texttt{iss}]$ (see Lines 4 and 6 of Algorithm 6). By applying Lemma 9 (Postcondition (VI)), it follows that $verificationKey \equiv \mathsf{pub}(\mathsf{signkey}(iss))$.

  As shown in Lemma 4, only $\mathsf{dom}^{-1}(iss.\texttt{host})$, i.e., $ssftr_t$, can derive $\mathsf{signkey}(iss)$. Valid signatures can only be created by a process that can also derive the signing key (see the equational theory in Definition 13), i.e., *set* was signed by $ssftr_t$ in a processing step P' prior to Q. $ssftr_t$ creates signatures only in Line 8 of Algorithm 5. When processing the push request, $ssftr_r$ stores the extracted SET (see Line 3 of Algorithm 6), i.e., $ssftr_t$ issued *set* in P' (according to Definition 6).

- Case 2: Line 38 of Algorithm 7: In this case, $ssftr_r$ is processing the response to a poll request. As in the previous case, the SET is added to `RXsets` after a successful check of the signature in Line 31 of Algorithm 7, using the key $S^q(ssftr_r).\texttt{RXissuers}[set[\texttt{iss}]][\texttt{jwks}]$ (see Lines 28 and 34 of Algorithm 7). The rest of the proof is analogous to the previous case.

(II):

We need to show that $set[\texttt{aud}]$ is an audience identifier of $ssftr_r$ at $ssftr_t$ under the issuer identifier $set[\texttt{iss}]$ in ρ, i.e., according to Definition 5:

(i) $set[\texttt{iss}] \in issIDs^{ssftr_t}$, $\exists rc \in \mathsf{ReceiverCreds}$ such that

(ii) $S^0(ssftr_r).\texttt{RXcredentials}[set[\texttt{iss}]][rc] = set[\texttt{aud}]$, and

(iii) $S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc][\texttt{aud}] = set[\texttt{aud}]$.

Let $iss := set[\texttt{iss}]$ be the issuer value contained in the SET. As shown for the previous postcondition, the SET set has been issued by $ssftr_t$ in a processing step prior to (S, E, N). From Lemma 3, it follows that $iss \in issIDs^{ssftr_t}$.

The receiver checks the audience value of the SET against values in RXstreams.

As shown for the previous postcondition, $ssftr_r$ stores set by executing Line 14 of Algorithm 6 or Line 38 of Algorithm 7 in some processing step $Q = (S^q, E^q, N^q) \to (S^{q\prime}, E^{q\prime}, N^{q\prime})$. In both cases, it checks the audience and issuer values of the SET against values stored in a stream configuration stored in the `RXstreams` state subterm:

- Case 1 (Pushed SET): Line 14 of Algorithm 6: From Lines 5, 11, 12, and Line 14 of Algorithm 6, it follows that $\exists streamID \in \mathcal{T}_{\mathcal{N}}$ s.t. $set[\texttt{aud}] \equiv S^q(ssftr_r).\texttt{RXstreams}[streamID][\texttt{expectedAud}]$. From Lines 4 and 10, it follows that $set[\texttt{iss}] \equiv S^q(ssftr_r).\texttt{RXstreams}[streamID][\texttt{issuer}]$.
- Case 2 (Polled SET): Line 38 of Algorithm 7: From Lines 26, 34, and Line 38 of Algorithm 7, it follows that $\exists streamID \in \mathcal{T}_{\mathcal{N}}$ s.t. $set[\texttt{aud}] \equiv S^q(ssftr_r).\texttt{RXstreams}[streamID][\texttt{expectedAud}]$. From Lines 27 and 36, it follows that $set[\texttt{iss}] \equiv S^q(ssftr_r).\texttt{RXstreams}[streamID][\texttt{issuer}]$.

RXstreams values are added when processing create-stream response.

Initially, the `RXstreams` state subterm of $ssftr_r$ is empty (Definition 3). Entries are only added to `RXstreams` in Lines 13ff. of Algorithm 7, i.e., when processing the response to a create-stream request, in some processing step $O = (S^o, E^o, N^o) \to (S^{o\prime}, E^{o\prime}, N^{o\prime})$.

Let *reference* be the second input argument of Algorithm 7 when called in O. The issuer values contained in the `RXstreams` entry is checked against $reference[\texttt{issuer}]$ (Lines 17ff. of Algorithm 7). Therefore, it follows that $reference[\texttt{issuer}] \equiv set[\texttt{iss}]$. From Line 20 of Algorithm 7, it follows that $reference[\texttt{expectedAud}] \equiv set[\texttt{aud}]$.

Create-stream request: Credential and audience.

As $reference[\texttt{responseTo}] \equiv \texttt{CREATE\_STREAM}$ (Line 13 of Algorithm 7), it follows that $ssftr_r$ called Line 24 of Algorithm 9 (Lemma 1) in some processing step $M = (S^m, E^m, N^m) \to (S^{m\prime}, E^{m\prime}, N^{m\prime})$, where the SSF transceiver calls HTTPS_SIMPLE_SEND. As $reference[\texttt{issuer}] \equiv set[\texttt{iss}]$, it follows that the value *issuer* chosen in Line 2 of Algorithm 9 in M is equal to $set[\texttt{iss}]$.

Let *receiverCredPair* be the value chosen in Line 4 of Algorithm 9 in M, i.e., $S^m(ssftr_r).\texttt{RXcredentials}[set[\texttt{iss}]]$. From $reference[\texttt{expectedAud}] \equiv set[\texttt{aud}]$, it follows that $receiverCredPair.2 \equiv set[\texttt{aud}]$. Let $rc := receiverCredPair.1$. As honest SSF transceivers never change their `RXcredentials` state subterm, it follows that $S^0(ssftr_r).\texttt{RXcredentials}[set[\texttt{iss}]][rc] = set[\texttt{aud}]$. From Definition 3 and $rc \in S^0(ssftr_r).\texttt{RXcredentials}[set[\texttt{iss}]]$, it follows that $rc \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$, $set[\texttt{iss}] \in \mathsf{supportedIssuers}(ssftr_r)$, and $set[\texttt{aud}] = \mathsf{receiverAudience}(rc)$.

Initial state of $ssftr_t$.

In the following, we show that $ssftr_t$ stores the same audience value under the dictionary keys $set[\texttt{iss}]$ and rc: According to Definition 3, $S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]] = txReceiverCreds(set[\texttt{iss}])$, as $set[\texttt{iss}] \in issIDs^{ssftr_t}$ (with $txReceiverCreds$ as defined inDefinition 3).

As $rc \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$ and $set[\texttt{iss}] \in \mathsf{supportedIssuers}(ssftr_r)$, it follows that $rc \in txReceiverCreds(set[\texttt{iss}])$, and $S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc][\texttt{aud}] = \mathsf{receiverAudience}(rc) = set[\texttt{aud}]$. ■

D.4. Confidentiality of SETs

Lemma 11 (SET Confidentiality). Every SSF Web System with a network attacker $\mathcal{SSF}$ provides SET confidentiality (Definition 9).

Proof.

(A) **Transmitter takes SET values from `TXstreams`.** Lemma 4 implies that the signing key $\mathsf{signkey}(set[\texttt{iss}])$ is only derivable by $ssftr_t$ (in S). Note that from the third precondition, it follows that $set[\texttt{iss}] \in issIDs^{ssftr_t}$. From the equational theory (Definition 13), it follows that no process other than $ssftr_t$ can create such a SET.

An honest SSF transceiver creates signatures only in Line 8 of Algorithm 5. Let $N = (S^n, E^n, N^n) \to (S^{n\prime}, E^{n\prime}, N^{n\prime})$ be the processing in which $ssftr_t$ creates set. Let $streamConfig$ be the first function argument of Algorithm 5 (when called in N).

From Lines 2, 3, 6, and 8 of Algorithm 5, it follows that $streamConfig[\texttt{issuer}] = set[\texttt{iss}]$ and $streamConfig[\texttt{aud}] = set[\texttt{aud}]$.

Algorithm 5 is only called in two places: Line 7 of Algorithm 4 (poll delivery) and Line 10 of Algorithm 12 (push delivery). In both cases, it holds true that $\exists sid \in \mathcal{T}_{\mathcal{N}}$: $streamConfig = S^n(ssftr_t).\texttt{TXstreams}[sid]$ (see Line 6 of Algorithm 4 and Line 3 of Algorithm 12).

(B) **Transmitter adds values to `TXstreams` only when processing create-stream requests.** Initially, the `TXstreams` state subterm of SSF transceivers is empty (Definition 3). Entries are only added in Line 28 of Algorithm 2 (the only other location where `TXstreams` is modified is Line 17 of Algorithm 3, where only the `subjects` entry of a stream is modified).

Let $L = (S^l, E^l, N^l) \to (S^{l\prime}, E^{l\prime}, N^{l\prime})$ be the processing step in which $ssftr_t$ adds $streamConfig$ to `TXstreams`. Let $req_{\texttt{create-stream}}$ be the HTTP request that $ssftr_t$ processes in L, i.e., the first function argument of Algorithm 2. This term is an HTTP request as Algorithm 2 is only called in Line 22 of Algorithm 1 with the first function argument of Algorithm 1. Algorithm 1 (PROCESS_HTTPS_REQUEST) is only called by the generic HTTPS server in Line 9 of Algorithm 31, where the server model checks the structure of the message (see Line 8 of Algorithm 31).

Let $rc := req_{\texttt{create-stream}}.\texttt{headers}[\texttt{Authorization}].2$ be the authorization credential in that request.

The value $issuer$ chosen in Line 2 of Algorithm 2 in processing step L is equal to $set[\texttt{iss}]$, as this value is stored in $streamConfig$ in Line 22 of Algorithm 2. The audience value of $streamConfig$, and therefore, of set, is chosen in Line 6 and Line 23 of Algorithm 2:

$set[\texttt{aud}] = S^l(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc][\texttt{aud}]$. As the $TXreceiverCreds$ state subterm is never modified, it follows that

$$set[\texttt{aud}] = S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc][\texttt{aud}] \tag{1}$$

Note that from the check in Line 7 of Algorithm 2, it follows that $set[\texttt{aud}] \not\equiv \langle\rangle$.

Due to the third precondition, it holds true that $set[\texttt{iss}] \in issIDs^{ssftr_t}$ and

$$\exists rc' \in \mathcal{T}_{\mathcal{N}} \colon set[\texttt{aud}] = S^0(ssftr_r).\texttt{RXcredentials}[set[\texttt{iss}]][rc'] \tag{2}$$

$$set[\texttt{aud}] = S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc'][\texttt{aud}] \tag{3}$$

From Equations 1 and 3 and Lemma 8, it follows that $rc = rc'$.

Together with Equation 2, this implies that

$$set[\texttt{aud}] = S^0(ssftr_r).\texttt{RXcredentials}[set[\texttt{iss}]][rc]$$

From Definition 3, it follows that $rxCreds(set[\texttt{iss}], ssftr_r)[rc] = set[\texttt{aud}]$ (with $rxCreds$ as defined in Definition 3), and $rc \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$.

By applying Lemma 7, it follows that the only $ssftr_t$ and $ssftr_r$ can derive rc in S. Therefore, the request $req_{\texttt{create-stream}}$ was created by $ssftr_t$ or $ssftr_r$ in some processing step $K = (S^k, E^k, N^k) \to (S^{k'}, E^{k'}, N^{k'})$.

(C) **Create-stream request was created by Receiver in Line 24 of Algorithm 9.** The request is a POST request containing the `Authorization` header and either an empty body, or a body with the dictionary key `delivery` (see Line 21 of Algorithm 1 and Line 3, 9, and Line 14 of Algorithm 2).

An honest SSF transceiver creates requests with an `Authorization` header only in

- Line 24 of Algorithm 9: Requesting a new stream.
- Line 12 of Algorithm 10: Sending add-subject requests. However, the body is not empty and does not contain the `delivery` key.
- Line 8 of Algorithm 11: Sending a poll request. In this case, the request is a GET request (Line 7 of Algorithm 11).
- Line 12 of Algorithm 12: Sending a push request. Here, the body contains a SET (Lines 10f. of Algorithm 12) and does not contain the `delivery` dictionary key (see also Algorithm 5)

I.e., $ssftr_t$ or $ssftr_r$ created $req_{\texttt{create-stream}}$ in Line 24 of Algorithm 9. We now show that $ssftr_r$ created the request: We assume that $ssftr_t \neq ssftr_r$ and that $ssftr_t$ created the request in K. Therefore, $\exists iss' \colon rc \in S^k(ssftr_t).\texttt{RXcredentials}[iss']$ (as this value is put into the Authorization header). From Definition 3, it follows that $rc \in \mathsf{sharedSecrets}(ssftr_t, iss')$ (note that the `RXcredentials` state subterm is never changed). As shown previously, $rc \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$, and from the restriction in Appendix A.3, it follows that $ssftr_t = ssftr_r$, which is a contradiction to our assumption. Therefore, $ssftr_r$ created $req_{\texttt{create-stream}}$ in Line 24 of Algorithm 9.

The new stream that $ssftr_r$ requests is associated with a push or poll delivery method. More precisely, $ssftr_r$ executes either Line 8 of Algorithm 9 or Line 17 of Algorithm 9 in processing step K.

- Case 1: Line 8 of Algorithm 9: In this case, it holds true that
  - $\texttt{delivery} \in req_{\texttt{create-stream}}.\texttt{body}$,
  - $req_{\texttt{create-stream}}.\texttt{body}[\texttt{delivery}][\texttt{method}] \equiv \texttt{urn:ietf:rfc:8935}$, and

– $req_{\texttt{create-stream}}.\texttt{body}[\texttt{delivery}][\texttt{endpoint\_url}] \equiv \langle \texttt{URL}, \texttt{S}, d_r, /\texttt{push-ep}, \langle\rangle, \bot \rangle$, with $d_r \in \mathsf{dom}(ssftr_r)$

(see Lines 8ff. of Algorithm 9 and Definition 3). Note that the endpoint URL is set to $S^k(ssftr_r).\texttt{RXpushEP}$, which is the same as in the initial state $S^0(ssftr_r)$.

- Case 2: Line 17 of Algorithm 9: In this case, the body of the request is empty, i.e., $req_{\texttt{create-stream}}.\texttt{body} \equiv \langle\rangle$ (see Line 19 of Algorithm 9)

(D) **SET does not leak to the attacker.** When processing the create-stream request in L, $ssftr_t$ associates either the push endpoint with the stream and sends the SET to this endpoint, or associates the authorization header with the stream and requires that this token is contained in the poll request. In either case, the SET does not leak to the attacker:

As noted before, $ssftr_t$ creates *streamConfig* in processing step L in Line 28 of Algorithm 2. It holds true that *streamConfig*[`delivery`][`method`] is either `urn:ietf:rfc:8936` or `urn:ietf:rfc:8935` (see Lines 10, 15, and 24 of Algorithm 2).

For the remainder of the proof, we distinguish between both delivery methods and show that the SET does not leak.

Case 1 *streamConfig*[`delivery`][`method`] $\equiv$ `urn:ietf:rfc:8936`. In this case, *streamConfig*[`token`] $= rc$ (Line 13 and Line 27 of Algorithm 2). As noted before, $ssftr_t$ creates *set* in processing step N by executing Algorithm 5. Algorithm 5 was called in Line 7 of Algorithm 4 (see the check of the delivery method in Line 6 of Algorithm 4; Algorithm 5 is also called in Line 10 of Algorithm 12, where the delivery method needs to be `urn:ietf:rfc:8935`, see Line 2 of Algorithm 12).

In Line 9 of Algorithm 4, $ssftr_t$ sends *set* as part of an HTTPS response to to an HTTPS request. Let $req_{\texttt{poll}}$ be this request, i.e., the first input argument of Algorithm 4. It holds true that $req_{\texttt{poll}}.\texttt{headers}[\texttt{Authorization}].2 \equiv$ *streamConfig*[`token`] $= rc$ (Line 4 and Line 6 of Algorithm 4). Furthermore, $req_{\texttt{poll}}$ was sent to the `/poll` endpoint of $ssftr_t$, as Algorithm 4 is only called in Line 28 of Algorithm 1.

As shown before, only $ssftr_t$ and $ssftr_r$ can derive rc. Furthermore, since $req_{\texttt{poll}}$ contains an `Authorization` header, it was created in one of the following locations in a processing step $M = (S^m, E^m, N^m) \rightarrow (S^{m\prime}, E^{m\prime}, N^{m\prime})$:

Line 24 of Algorithm 9 (requesting a new stream) The authorization header of the request contains the value rc. This request was not created by $ssftr_t$ if $ssftr_t \neq ssftr_r$ (see the contradiction in (C)). Thus, the request was created by $ssftr_r$. As shown before, $rc \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$. In M, $ssftr_r$ executed Line 4 of Algorithm 9, i.e., $\exists iss\colon rc \in S^m(ssftr_r).\texttt{RXcredentials}[iss] = S^0(ssftr_r).\texttt{RXcredentials}[iss]$.

This implies that $rc \in \mathsf{sharedSecrets}(ssftr_r, iss)$. With the restriction from Appendix A.3, it follows that $iss = set[\texttt{iss}]$. However, Lemma 9 implies that $S^m(ssftr_r).\texttt{RXissuers}[set[\texttt{iss}]][\texttt{configuration\_endpoint}].\texttt{path}$ is `/configure-stream`, i.e., $req_{\texttt{poll}}$ was not created in Line 24 of Algorithm 9.

Line 12 of Algorithm 10 (add subject request) In this case, the process would ignore the response, i.e., the SET contained in the response would not leak.

Line 8 of Algorithm 11 (poll request) Here, the response is processed in Lines 24ff. of Algorithm 7. The SET contained in the response is stored in the `RXsets` state subterm in Line 38 of Algorithm 7, and not used in any other way. An honest SSF transceiver does not retrieve the values stored in `RXsets`, i.e., the SET contained in the response does not leak.

Line 12 of Algorithm 12 (sending a push request) In this case, the SSF transceiver ignores the response.

Case 2 *streamConfig*[`delivery`][`method`] ≡ `urn:ietf:rfc:8935`. In this case, Algorithm 5 was called in Line 10 of Algorithm 12. The endpoint to which the SET is pushed is chosen in Line 4 of Algorithm 12, and is the delivery endpoint URL stored in *streamConfig*. As shown above, this endpoint is $\langle$`URL`, `S`, d_r, /`push-ep`, $\langle\rangle$, $\bot\rangle$, with $d_r \in \mathsf{dom}(ssftr_r)$. Thus, the request is sent to the /`push-ep` endpoint of $ssftr_r$, which is processed in Line 30 of Algorithm 1, i.e., in Algorithm 6. There, the process stores the SET into its `RXsets` state subterm (Line 14 of Algorithm 6). As shown before, values stored in `RXsets` are not accessed again and, thus, cannot leak. ■

D.5. Authorization

Lemma 12 (Authorization). Every SSF Web System with a network attacker $\mathcal{SSF}$ provides authorization (Definition 10).

Proof.
Postcondition (I):

(A) **Transmitter takes SET values from `TXstreams`.** During P, $ssftr_t$ calls Algorithm 5 (CREATE_SET). Let *streamConfig* be the first argument in that function call.

From Lines 2, 3, 6, and 8 of Algorithm 5, it follows that *streamConfig*[`issuer`] = *set*[`iss`] and *streamConfig*[`aud`] = *set*[`aud`].

Algorithm 5 is only called in two places: Line 7 of Algorithm 4 (poll delivery) and Line 10 of Algorithm 12 (push delivery). In both cases, it holds true that $\exists sid \in \mathcal{T}_{\mathcal{N}}$: *streamConfig* = $S^p(ssftr_t)$.`TXstreams`[*sid*] (see Line 6 of Algorithm 4 and Line 3 of Algorithm 12).

(B) **Transmitter adds values to `TXstreams` only when processing create-stream requests.** Initially, the `TXstreams` state subterm of SSF transceivers is empty (Definition 3). Entries are only added in Line 28 of Algorithm 2 (the only other location where `TXstreams` is modified is Line 17 of Algorithm 3, where only the `subjects` entry of a stream is modified).

Let $L = (S^l, E^l, N^l) \to (S^{l'}, E^{l'}, N^{l'})$ be the processing step in which $ssftr_t$ adds *streamConfig* to `TXstreams`. Let $req_{\texttt{create-stream}}$ be the HTTP request that $ssftr_t$ processes in L, i.e., the first function argument of Algorithm 2. This term is an HTTP request as Algorithm 2 is only called in Line 22 of Algorithm 1 with the first function argument of Algorithm 1. Algorithm 1 (PROCESS_HTTPS_REQUEST) is only called by the generic HTTPS server in Line 9 of Algorithm 31, where the server model checks the structure of the message (see Line 8 of Algorithm 31). Let $rc' := req_{\texttt{create-stream}}$.`headers`[`Authorization`].2 be the authorization credential of that request.

The value *issuer* chosen in Line 2 of Algorithm 2 in processing step L is equal to *set*[`iss`], as this value is stored in *streamConfig* in Line 22 of Algorithm 2 (and taken from there during P when creating *set*, see above).

The audience value of *streamConfig*, and therefore, of *set*, is chosen in Lines 6 and 23 of Algorithm 2 as *set*[`aud`] = $S^l(ssftr_t)$.`TXreceiverCreds`[*set*[`iss`]][rc'][`aud`]. As the *TXreceiverCreds* state subterm is never modified, it follows that

$$set[\texttt{aud}] = S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc'][\texttt{aud}] \tag{4}$$

Note that from the check in Line 7 of Algorithm 2, it follows that *set*[`aud`] $\not\equiv \langle\rangle$.

From Definition 3, it follows that $set[\texttt{iss}] \in issIDs^{ssftr_t}$ and $\exists ssftr_r \in \mathsf{SSFTR}$ such that $rc' \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$ and $set[\texttt{iss}] \in \mathsf{supportedIssuers}(ssftr_r)$. Furthermore, it holds true that $set[\texttt{aud}] = \mathsf{receiverAudience}(rc')$.

Finally, Definition 3 implies that the initial state of $ssftr_r$ contains the same values ($set[\texttt{iss}]$, rc', and $set[\texttt{aud}]$) in its `RXcredentials` state subterm due to $set[\texttt{iss}] \in \mathsf{supportedIssuers}(ssftr_r)$ and $rc' \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$:

$$S^0(ssftr_r).\texttt{RXcredentials}[set[\texttt{iss}]][rc'] = \mathsf{receiverAudience}(rc') = set[\texttt{aud}] \tag{5}$$

Therefore, it holds true that $set[\texttt{aud}]$ is an audience identifier of $ssftr_r$ at $ssftr_t$ under the issuer identifier $set[\texttt{iss}]$ (see Definition 5).

Postcondition (II): We start by noting that $set[\texttt{sub\_id}]$ is taken from $streamConfig[\texttt{subjects}]$ during P, see Line 5 of Algorithm 5.

When creating $streamConfig$ in L (see above), the list of subjects is empty, i.e., $S^l(ssftr_t).\texttt{TXstreams}[sid][\texttt{subjects}] = \langle\rangle$ (Line 26 of Algorithm 2). The `subjects` entry of a `TXstreams` entry is only modified in Line 17 of Algorithm 3, where $ssftr_t$ processes an add-subject request.

Algorithm 3 is only called in Line 26 of Algorithm 1, i.e., at the `/add-subject` endpoint. The first input of the algorithm is an HTTP request (as Algorithm 1 is only called by the generic HTTPS server in Line 9 of Algorithm 31, where the server checks the structure of the term).

Let $M = (S^m, E^m, N^m) \to (S^{m\prime}, E^{m\prime}, N^{m\prime})$ be the processing in which $ssftr_t$ adds $set[\texttt{sub\_id}]$ to the stream configuration entry, and let $req_{\texttt{add-sub}}$ be the corresponding HTTP request (i.e., the first function argument of Algorithm 3 when called in M). That request contains a stream ID $sid = req_{\texttt{add-sub}}.\texttt{body}[\texttt{stream\_id}]$ (see Lines 5 and 17 of Algorithm 3).

Let $rc := req_{\texttt{add-sub}}.\texttt{headers}[\texttt{Authorization}][\texttt{Bearer}]$ be the authorization token contained in the request. As noted above, $set[\texttt{aud}] \not\equiv \langle\rangle$.

From the check in Line 13 of Algorithm 3, it follows that the issuer value that $ssftr_t$ creates in Line 7 of Algorithm 3 is equal to $S^m(ssftr_t).\texttt{TXstreams}[sid][\texttt{iss}]$, i.e., $set[\texttt{iss}]$. Thus, it follows that $rc \in S^m(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]]$ (see Lines 4 and 8 of Algorithm 3). Definition 3 now implies that $rc \in txReceiverCreds(set[\texttt{iss}])$ (with $txReceiverCreds$ as in Definition 3). Therefore, $\exists ssftr'_r \in \mathsf{SSFTR}\colon rc \in \mathsf{sharedSecrets}(ssftr'_r, set[\texttt{iss}])$.

The subject identifier added to the stream configuration in Line 17 of Algorithm 3 is contained in the `TXreceiverCreds` entry, i.e., $set[\texttt{sub\_id}] \in txReceiverCreds(set[\texttt{iss}])[rc][\texttt{subjects}]$ (see Lines 8, 10, 15, and 17 of Algorithm 3).

Hence, with Definition 3, we can conclude $set[\texttt{sub\_id}] \in \mathsf{permittedSubjects}(rc)$.

Postcondition (III): From the check in Line 11 of Algorithm 3, it follows that $S^m(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc][\texttt{aud}] \equiv set[\texttt{aud}]$. As the `TXreceiverCreds` state subterm is never modified, we also have $S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc][\texttt{aud}] \equiv set[\texttt{aud}]$.

Furthermore, Equation 4 gives us

$$S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc][\texttt{aud}] \equiv S^0(ssftr_t).\texttt{TXreceiverCreds}[set[\texttt{iss}]][rc'][\texttt{aud}]$$

We can now apply Lemma 8 to get $rc = rc'$. Therefore, $rc \in \mathsf{sharedSecrets}(ssftr'_r, set[\texttt{iss}])$ and $rc \in \mathsf{sharedSecrets}(ssftr_r, set[\texttt{iss}])$. The definition of sharedSecrets in Appendix A.3 now immediately implies $ssftr_r = ssftr'_r$. ∎

E. Technical Definitions

Here, we provide technical definitions of the WIM. These follow the descriptions in [3–9].

E.1. Terms and Notations

As usual in Dolev-Yao-style models, there is an underlying term algebra, with formal terms over a signature Σ, and an equational theory defined by a set of equations over these terms. Messages, internal state, and protocol events are then expressed as terms.

Definition 11 (Signature). In the case of the WIM, the signature Σ consists of the following pairwise disjoint sets:

Constants $\mathcal{C} = \mathbb{S} \cup \mathsf{IPs} \cup \{\bot, \top, \diamond\}$ with the three sets pairwise disjoint. $\mathbb{S}$ is the set of all (ASCII) strings, including the empty string ε. We write string values in a `typewriter font`. IPs is the set of IP addresses.

Function Symbols to represent public keys, asymmetric encryption and decryption, symmetric encryption and decryption, signatures, signature verification, MACs, MAC verification, message extraction from signatures and MACs, and hashing, respectively: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot)$, $\mathsf{mac}(\cdot,\cdot)$, $\mathsf{checkmac}(\cdot,\cdot)$, $\mathsf{extractmsg}(\cdot)$, $\mathsf{hash}(\cdot)$.

Sequences of any length $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc. Note that formally, these sequence "constructors" are also function symbols.

Projection Symbols to access sequence elements: $\pi_i(\cdot)$ for all $i \in \mathbb{N}_\emptyset$. Note that formally, projection symbols are also function symbols.

Definition 12 (Nonces and Terms). Given this signature, we define $X = \{x_1, x_2, \ldots\}$ to be an infinite set of variables, and $\mathcal{N}$ to be an infinite set of constants (*nonces*) such that $\Sigma, X, \mathcal{N}$ are pairwise disjoint. With these, we can now define the set of terms $\mathcal{T}_N(X)$ over $\Sigma \cup X \cup N$ for any set $N \subseteq \mathcal{N}$ inductively as follows:

- If $t \in \mathcal{C} \cup N \cup X$, then $t \in \mathcal{T}_N(X)$.
- If $f \in \Sigma$ is an n-ary function symbol for some $n \in \mathbb{N}_0$, and $t_1, \ldots, t_n \in \mathcal{T}_N(X)$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_N(X)$.

Definition 13 (Equational Theory and Term Equivalence). Furthermore, we associate an equational theory with Σ, modeling the semantics of the function symbols. Our equational theory is defined by the following equations:

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \tag{6}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x \tag{7}$$
$$\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) = \top \tag{8}$$
$$\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x \tag{9}$$
$$\mathsf{checkmac}(\mathsf{mac}(x, y), y) = \top \tag{10}$$
$$\mathsf{extractmsg}(\mathsf{mac}(x, y)) = x \tag{11}$$
$$\pi_i(\langle x_1, \ldots, x_n\rangle) = x_i \text{ if } 1 \le i \le n \tag{12}$$
$$\pi_j(\langle x_1, \ldots, x_n\rangle) = \diamond \text{ if } j \notin \{1, \ldots, n\} \tag{13}$$
$$\pi_j(t) = \diamond \text{ if } t \text{ is not a sequence} \tag{14}$$

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

Definition 14 (Ground Terms, Messages, Placeholders, Protomessages). $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$ denotes the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

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

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

Definition 15 (Events and Protoevents). An *event (over* IPs *and* $\mathcal{M}$*)* is a term of the form $\langle a, f, m\rangle$, for a, $f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events. Events over IPs and $\mathcal{M}^\nu$ are called *protoevents* and are denoted $\mathcal{E}^\nu$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^\nu\langle\rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle\rangle$, $\langle\langle a, f, m\rangle, \langle a', f', m'\rangle, \dots\rangle$, etc.).

Definition 16 (Normal Form). Let t be a term. The *normal form* of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Definition 13. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 17 (Pattern Matching). Let *pattern* $\in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches pattern* iff t can be acquired from *pattern* by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim$ *pattern*. For a sequence of patterns *patterns* we write $t \dot{\sim}$ *patterns* to denote that t matches at least one pattern in *patterns*.

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

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

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

Definition 18 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$ a term, and $t_1, \dots, t_n \in \mathcal{T}_N$ ground terms. By $\tau[t_1/x_1, \dots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \dots, n\}$.

Definition 19 (Sequence Notations). Let $t = \langle t_1, \dots, t_n\rangle$ and $r = \langle r_1, \dots, r_m\rangle$ be sequences, s a set, and x, y terms. We define the following operations:

- $t \subset^{\langle\rangle} s \iff t_1, \dots, t_n \in s$
- $x \in^{\langle\rangle} t \iff \exists i\colon\ t_i = x$
- $t +^{\langle\rangle} y := \langle t_1, \dots, t_n, y\rangle$
- $t \cup r := \langle t_1, \dots, t_n, r_1, \dots, r_m\rangle$
- $t -^{\langle\rangle} y := \begin{cases} \langle t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n\rangle & \text{if } \exists i\colon t_i = x \text{ (i.e., } y \in^{\langle\rangle} t\text{)} \\ t & \text{otherwise (i.e., } y \notin^{\langle\rangle} t\text{)} \end{cases}$

  If y occurs more than once in t, $t -^{\langle\rangle} y$ non-deterministically removes one of the occurrences.
- $t -^{\langle\rangle *} y$ is t with all occurrences of y removed.

- $|t| := n$. If t' is not a sequence, we set $|t'| := \diamond$.
- For a finite set M with $M = \{m_1, \ldots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \ldots, m_n \rangle$. The order of the elements does not matter; one is chosen arbitrarily.

Definition 20 (Dictionaries). A *dictionary over X and Y* is a term of the form

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

where $k_1, \ldots, k_n \in X$, $v_1, \ldots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \ldots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1 \colon v_1, \ldots, k_n \colon v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$. Note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$; and dictionaries as such may contain duplicate keys (however, all dictionary operations are only defined on dictionaries with unique keys).

Definition 21 (Operations on Dictionaries). Let $z = [k_1 \colon v_1, k_2 \colon v_2, \ldots, k_n \colon v_n]$ be a dictionary with unique keys, i.e., $\forall i, j \colon k_i \neq k_j$. In addition, let t and v be terms. We define the following operations:

- $t \in z \iff \exists i \in \{1, \ldots, n\} \colon k_i = t$
- $z[t] := \begin{cases} v_i & \text{if } \exists k_i \in z \colon t = k_i \\ \langle\rangle & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- $z - t := \begin{cases} [k_1 \colon v_1, \ldots, k_{i-1} \colon v_{i-1}, k_{i+1} \colon v_{i+1}, \ldots, k_n \colon v_n] & \text{if } \exists k_i \in z \colon t = k_i \\ z & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- In our algorithm descriptions, we often write **let** $z[t] := v$. If $t \notin z$ prior to this operation, an element $\langle t, v \rangle$ is appended to z. Otherwise, i.e., if there already is an element $\langle t, x \rangle \in^{\langle\rangle} z$, this element is updated to $\langle t, v \rangle$.

We emphasize that these operations are only defined on dictionaries with unique keys.

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

Definition 22 (Pointers). A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

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

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

Definition 23 (Concatenation of Sequences). For a sequence $a = \langle a_1, \ldots, a_i \rangle$ and a sequence $b = \langle b_1, b_2, \ldots \rangle$, we define the *concatenation* as $a \cdot b := \langle a_1, \ldots, a_i, b_1, b_2, \ldots \rangle$.

Definition 24 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a non-deterministically chosen occurence of that element in X is removed.

E.2. Message and Data Formats

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

E.2.1. URLs

Definition 25. A *URL* is a term of the form

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

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

The *fragment* part of a URL can be omitted when writing the URL. Its value is then defined to be $\bot$. We sometimes also write $\mathsf{URL}_{path}^{host}$ to denote the URL $\langle \mathtt{URL}, \mathtt{S}, host, path, \langle\rangle, \bot \rangle$.

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 22):

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

E.2.2. Origins

Definition 26. An *origin* is a term of the form $\langle host, protocol \rangle$ with $host \in \mathsf{Doms}$ and $protocol \in \{\mathtt{P}, \mathtt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins.

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

E.2.3. Cookies

Definition 27. A *cookie* is a term of the form $\langle name, content \rangle$ where $name \in \mathcal{T}_{\mathcal{N}}$, and *content* is a term of the form $\langle value, secure, session, httpOnly \rangle$ where $value \in \mathcal{T}_{\mathcal{N}}$, $secure$, $session$, $httpOnly \in \{\top, \bot\}$. As *name* is a term, it may also be a sequence consisting of two parts. If the name consists of two parts, we call the first part of the sequence (i.e., $name.\mathtt{1}$) the *prefix* of the name. We write $\mathsf{Cookies}$ for the set of all cookies and $\mathsf{Cookies}^{\nu}$ for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

If the *secure* attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections.[6] If the *session* flag is set, this cookie will be deleted as soon as the browser is closed. The *httpOnly* attribute controls whether scripts have access to this cookie.

When the __Host prefix (see [2]) of a cookie is set (i.e., *name* consists of two parts and $name.\mathtt{1} \equiv \mathtt{\_\_Host}$), the browser accepts the cookie only if the *secure* attribute is set. As such cookies are only transferred over secure channels (i.e., with TLS), the cookie cannot be set by a network attacker. Note that the WIM does not model the domain attribute of the Set-Cookie header, so cookies in the WIM are always sent to the originating domain and not some subdomain. Therefore, the WIM models only the __Host prefix, but not the __Secure prefix.

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

[6] Note that *secure* cookies can be set over unencrypted connections (c.f. RFC 6265).

E.2.4. HTTP Messages

Definition 28. An *HTTP request* is a term of the form shown in (15). An *HTTP response* is a term of the form shown in (16).

$$\langle \texttt{HTTPReq}, \mathit{nonce}, \mathit{method}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body} \rangle \tag{15}$$

$$\langle \texttt{HTTPResp}, \mathit{nonce}, \mathit{status}, \mathit{headers}, \mathit{body} \rangle \tag{16}$$

The components are defined as follows:

- $\mathit{nonce} \in \mathcal{N}$ serves to map each response to the corresponding request.
- $\mathit{method} \in \mathsf{Methods}$ is one of the HTTP methods.
- $\mathit{host} \in \mathsf{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $\mathit{path} \in \mathbb{S}$ indicates the resource path at the server side.
- $\mathit{status} \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard).
- $\mathit{parameters} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains URL parameters.
- $\mathit{headers} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \texttt{Origin}, o \rangle$ where o is an origin,
  - $\langle \texttt{Set-Cookie}, c \rangle$ where c is a sequence of cookies,
  - $\langle \texttt{Cookie}, c \rangle$ where $c \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$ (note that in this header, only names and values of cookies are transferred, i.e., no attributes),
  - $\langle \texttt{Location}, l \rangle$ where $l \in \mathsf{URLs}$,
  - $\langle \texttt{Referer}, r \rangle$ where $r \in \mathsf{URLs}$,
  - $\langle \texttt{Strict-Transport-Security}, \top \rangle$,
  - $\langle \texttt{Authorization}, \langle \mathit{username}, \mathit{password} \rangle \rangle$ where $\mathit{username}, \mathit{password} \in \mathbb{S}$ (this header models the 'Basic' HTTP Authentication Scheme, see [RFC7617]),
  - $\langle \texttt{ReferrerPolicy}, p \rangle$ where $p \in \{\texttt{noreferrer}, \texttt{origin}\}$.
- $\mathit{body} \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

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

Example 6 (HTTP Request and Response).

$$r := \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, /\texttt{show}, \langle\langle \texttt{index}, 1 \rangle\rangle, \\ [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S} \rangle], \langle \texttt{foo}, \texttt{bar} \rangle \rangle \tag{17}$$

$$s := \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top \rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x \rangle \rangle \tag{18}$$

An HTTP `POST` request for the URL `http://example.com/show?index=1` is shown in (17), with an Origin header and a body that contains $\langle \texttt{foo}, \texttt{bar} \rangle$. A possible response is shown in (18), which contains an httpOnly cookie with name `SID` and value n_2 as well as a string `somescript` representing a script that can later be executed in the browser (see Section E.11) and the scripts initial state x.

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

Definition 29. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k' \rangle, k)$, where $k \in \mathit{terms}$, $k' \in \mathcal{N}$, and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc_s}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

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

Example 7.

$$\mathsf{enc_a}(\langle r, k' \rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{19}$$

$$\mathsf{enc_s}(s, k') \tag{20}$$

The term (19) shows an encrypted request (with r as in (17)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (20) is a response (with s as in (18)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (19).

E.2.5. DNS Messages

Definition 30. A *DNS request* is a term of the form $\langle \texttt{DNSResolve}, \mathit{domain}, \mathit{nonce} \rangle$ where $\mathit{domain} \in \mathsf{Doms}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$.

Definition 31. A *DNS response* is a term of the form $\langle \texttt{DNSResolved}, \mathit{domain}, \mathit{result}, \mathit{nonce} \rangle$ with $\mathit{domain} \in \mathsf{Doms}$, $\mathit{result} \in \mathsf{IPs}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS responses $\mathsf{DNSResponses}$.

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

E.3. Atomic Processes, Systems and Runs

Entities that take part in a network are modeled as atomic processes. An atomic process takes a term that describes its current state and an event as input, and then (non-deterministically) outputs a new state and a sequence of events.

Definition 32 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

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

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

Definition 33 (Configurations). A *configuration of a system* $\mathcal{P}$ is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence[7] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$.

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

Definition 34 (Processing Steps). A *processing step of the system* $\mathcal{P}$ is of the form

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

where

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

such that there exists

1. a sequence (term) $E^{\nu}_{\text{out}} \subseteq 2^{\mathcal{E}^{\nu}\langle\rangle}$ of protoevents,
2. a term $s^{\nu} \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
3. a sequence $(v_1, v_2, \ldots, v_i)$ of all placeholders appearing in E^{ν}_{out} (ordered lexicographically),
4. a sequence $N^{\nu} = (\eta_1, \eta_2, \ldots, \eta_i)$ of the first i elements in N

with

1. $((e_{\text{in}}, S(p)), (E^{\nu}_{\text{out}}, s^{\nu})) \in R^p$ and $a \in I^p$,
2. $E_{\text{out}} = E^{\nu}_{\text{out}}[\eta_1/v_1, \ldots, \eta_i/v_i]$,
3. $S'(p) = s^{\nu}[\eta_1/v_1, \ldots, \eta_i/v_i]$ and $S'(p') = S(p')$ for all $p' \neq p$,
4. $E' = E_{\text{out}} \cdot (E \setminus \{e_{\text{in}}\})$,
5. $N' = N \setminus N^{\nu}$.

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

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

Definition 35 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0* is a finite sequence of configurations $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

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

When we write that a processing step $P = (S, E, N) \to (S', E', N')$ is in a run ρ of some system, we mean that there is an index i such that $(S, E, N) = (S^i, E^i, N^i) \in \rho$ and $(S', E', N') = (S^{i+1}, E^{i+1}, N^{i+1}) \in \rho$.

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

E.4. Atomic Dolev-Yao Processes

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

Definition 36 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with variables V* if there exist $m_1, \ldots, m_n \in M$ with $n \geq 0$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \ldots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.

For example, the term a can be derived from the set of terms $\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\}$, i.e., $a \in d_\emptyset(\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\})$.

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

Definition 37 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ such that p is an atomic process and for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_\mathcal{N}$, $s' \in \mathcal{T}_\mathcal{N}(V_{\text{process}})$, with $((e, s), (E, s')) \in R^p$ it holds true that $E, s' \in d_{V_{\text{process}}}(\{e, s\})$.

E.5. Attackers

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

Definition 38 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$* is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e, and $s \in \mathcal{T}_\mathcal{N}$ we have that $((e, s), (E, s')) \in R$ iff $s' = \langle e, E, s\rangle$ and $E = \langle\langle a_1, f_1, m_1\rangle, \ldots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_1, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e, s\})$.

Note that in a Web system, we distinguish between two kinds of attacker processes: Web attackers and network attackers. Both kinds match the definition above, but differ in the set of assigned addresses in the context of a Web system. While for Web attackers, the set of addresses I^p is disjoint from other Web attackers and honest processes, i.e., Web attackers participate in the network as any other party, the set of addresses I^p of a network attacker is not restricted. Hence, a network attacker can intercept events addressed to any party as well as spoof all addresses. Note that one network attacker subsumes any number of Web attackers as well as any number of network attackers.

E.6. Notations for Functions and Algorithms

When describing algorithms, we use the following notations:

E.6.1. Non-deterministic choosing and iteration

The notation **let** $n \leftarrow N$ is used to describe that n is chosen non-deterministically from the set (or sequence) N. If N is empty, the corresponding processing step in which this selection happens does not finish. We write **for** $s \in M$ **do** to denote that the following commands are repeated for every element in M, where the variable s is the current element. The order in which the elements are processed is chosen non-deterministically. We write, for example,

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

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

E.6.2. Function calls

When calling functions that do not return anything, we write

call FUNCTION_NAME(x, y)

to describe that a function FUNCTION_NAME is called with two variables x and y as parameters. If that function executes the command **stop** E, s', the processing step terminates, where E is the sequence of events output by the associated process and s' is its new state. If that function does not terminate with a **stop**, the control flow returns to the calling function at the next line after the call.

When calling a function that has a return value, we omit the **call** and directly write

let $z :=$ FUNCTION_NAME(x, y)

to assign the return value to a variable z after the function returns. Note that the semantics for execution of **stop** within such functions is the same as for functions without a return value.

E.6.3. Stop without output

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

E.6.4. Placeholders

In several places throughout the algorithms we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 12). Table 1 shows a list of some of the placeholders, generally denoted by ν with some subscript to distinguish between multiple fresh values.

E.6.5. Abbreviations for URLs and Origins

We sometimes use an abbreviation for URLs. We write URL^d_{path} to describe the following URL term: $\langle \texttt{URL}, \texttt{S}, d, path, \langle\rangle \rangle$. If the domain d belongs to some distinguished process P and it is the only domain associated to this process, we may also write $\mathsf{URL}^{\mathsf{P}}_{path}$. For a (secure) origin $\langle d, \texttt{S} \rangle$ of some domain d, we also write origin_d. Again, if the domain d belongs to some distinguished process P and d is the only domain associated to this process, we may write $\mathsf{origin}_{\mathsf{P}}$.

E.7. Browsers

Here, we present the formal model of browsers.

E.7.1. Scripts

Recall that a *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below).

Definition 39 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Placeholder	Usage
ν_1	Algorithm 22, new window nonces
ν_2	Algorithm 22, new HTTP request nonce
ν_3	Algorithm 22, lookup key for pending HTTP requests entry
ν_4	Algorithm 20, new HTTP request nonce (multiple lines)
ν_5	Algorithm 20, new subwindow nonce
ν_6	Algorithm 21, new HTTP request nonce
ν_7	Algorithm 21, new document nonce
ν_8	Algorithm 17, lookup key for pending DNS entry
ν_9	Algorithm 14, new window nonce
$\nu_{10}, \ldots$	Algorithm 20, replacement for placeholders in script output

Table 1: List of placeholders used in browser algorithms.

Definition 40 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{\text{script}}}(s)$.

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

Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

Definition 41 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$.

E.7.2. Web Browser State

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

Definition 42. A *window* is a term of the form $w = \langle \mathit{nonce}, \mathit{documents}, \mathit{opener} \rangle$ with $\mathit{nonce} \in \mathcal{N}$, $\mathit{documents} \subset^{\langle\rangle}$ Documents (defined below), $\mathit{opener} \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle} \mathit{documents}$ if *documents* is not empty (we then call d the *active document of* w). We write Windows for the set of all windows. We write $w.\texttt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else.

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

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

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

Definition 43. A *document* d is a term of the form

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

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

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

Definition 44. For two window terms w and w' we write

$$w \xrightarrow{\text{childof}} w'$$

if $w \in^{\langle\rangle} w'.\texttt{activedocument.subwindows}$. We write $\xrightarrow{\text{childof}^+}$ for the transitive closure and we write $\xrightarrow{\text{childof}^*}$ for the reflexive transitive closure.

In the Web browser state, HTTP(S) messages are tracked using *references*, where we distinguish between references for XMLHttpRequests and references for normal HTTP(S) requests.

Definition 45. A reference for a normal HTTP(S) request is a sequence of the form $\langle \texttt{REQ}, nonce\rangle$, where $nonce$ is a window reference. A reference for a XMLHttpRequest is a sequence of the form $\langle \texttt{XHR}, nonce, xhrreference\rangle$, where $nonce$ is a document reference and $xhrreference$ is some nonce that was chosen by the script that initiated the request.

We can now define the set of states of Web browsers. Note that we use the dictionary notation that we introduced in Definition 20.

Definition 46. The *set of states* $Z_{\text{webbrowser}}$ *of a Web browser atomic Dolev-Yao process* consists of the terms of the form

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

with the subterms as follows:

- $windows \subset^{\langle\rangle} \mathsf{Windows}$ contains a list of window terms (modeling top-level windows, or browser tabs) which contain documents, which in turn can contain further window terms (iframes).
- $ids \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ is a list of identities that are owned by this browser (i.e., belong to the user of the browser).
- $secrets \in [\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}]$ contains a list of secrets that are associated with certain origins (i.e., passwords of the user of the browser at certain websites). Note that this structure allows to have a single secret under an origin or a list of secrets under an origin.
- $cookies$ is a dictionary over Doms and sequences of $\mathsf{Cookies}$ modeling cookies that are stored for specific domains.
- $localStorage \in [\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}]$ stores the data saved by scripts using the localStorage API (separated by origins).
- $sessionStorage \in [OR \times \mathcal{T}_{\mathcal{N}}]$ for $OR := \{\langle o, r\rangle | \, o \in \mathsf{Origins}, r \in \mathcal{N}\}$ similar to localStorage, but the data in sessionStorage is additionally separated by top-level windows.
- $keyMapping \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$ maps domains to TLS encryption keys.

- $sts \subset^{\langle\rangle}$ Doms stores the list of domains that the browser only accesses via TLS (strict transport security).
- $DNSaddress \in$ IPs defines the IP address of the DNS server.
- $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ contains one pairing per unanswered DNS query of the form $\langle reference, request, url \rangle$. In these pairings, *reference* is an HTTP(S) request reference (as above), *request* contains the HTTP(S) message that awaits DNS resolution, and *url* contains the URL of said HTTP request. The pairings in *pendingDNS* are indexed by the DNS request/response nonce.
- $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ contains pairings of the form $\langle reference, request, url, key, f \rangle$ with the terms *reference*, *request*, and *url* as in *pendingDNS*, *key* is the symmetric encryption key if HTTPS is used or $\perp$ otherwise, and f is the IP address of the server to which the request was sent.
- $isCorrupted \in \{\perp, \texttt{FULLCORRUPT}, \texttt{CLOSECORRUPT}\}$ specifies the corruption level of the browser.

In corrupted browsers, certain subterms are used in different ways (e.g., *pendingRequests* is used to store all observed messages).

E.7.3. Web Browser Relation

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

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

Subwindows and Docs. Given a browser state s, Subwindows(s) denotes the set of all pointers[8] to windows in the window list $s.\texttt{windows}$ and (recursively) the subwindows of their active documents. We exclude subwindows of inactive documents and their subwindows. With Docs(s) we denote the set of pointers to all active documents in the set of windows referenced by Subwindows(s).

Definition 47. For a browser state s we denote by Subwindows(s) the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\texttt{windows}$ there is a $\overline{p} \in$ Subwindows(s) such that $s.\overline{p} = w$. (2) For all $\overline{p} \in$ Subwindows(s), the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in$ Subwindows(s) such that $s.\overline{p'} = w$.

Given a browser state s, the set Docs(s) of pointers to active documents is the minimal set such that for every $\overline{p} \in$ Subwindows(s) with $s.\overline{p}.\texttt{activedocument} \not\equiv \langle\rangle$, there exists a pointer $\overline{p'} \in$ Docs(s) with $s.\overline{p'} = s.\overline{p}.\texttt{activedocument}$.

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

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

[8] Recall the definition of a pointer in Definition 22.

Definition 48. Let s be a browser state and d a document. By $\mathsf{Clean}(s, d)$ we denote the term that equals $s.\texttt{windows}$ but with (1) all inactive documents removed (including their subwindows etc.), (2) all subterms that represent non-same-origin documents w.r.t. d replaced by a limited document d' with the same nonce and the same subwindow list, and (3) the values of the subterms `headers` for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.)

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

Definition 49. For a sequence of cookies (with pairwise different names) *oldcookies*, a sequence of cookies *newcookies*, and a string $protocol \in \{\texttt{P}, \texttt{S}\}$, the set $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\texttt{content.httpOnly} \equiv \top$ or where $(c.\texttt{name.1} \equiv \texttt{\_\_Host}) \wedge ((protocol \equiv \texttt{P}) \vee (c.\texttt{secure} \equiv \bot))$. For any c, $c' \in^{\langle\rangle}$ *newcookies*, $c.\texttt{name} \equiv c'.\texttt{name}$, remove the cookie that appears left of the other in *newcookies*. Let m be the set of cookies that have a name that either appears in *oldcookies* or in *newcookies*, but not in both. For all pairs of cookies $(c_{\text{old}}, c_{\text{new}})$ with $c_{\text{old}} \in^{\langle\rangle}$ *oldcookies*, $c_{\text{new}} \in^{\langle\rangle}$ *newcookies*, $c_{\text{old}}.\texttt{name} \equiv c_{\text{new}}.\texttt{name}$, add c_{new} to m if $c_{\text{old}}.\texttt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ is m.

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

Definition 50. For a sequence of cookies (with pairwise different names) *oldcookies*, a cookie c, and a string $protocol \in \{\texttt{P}, \texttt{S}\}$ (denoting whether the HTTP response was received from an insecure or a secure origin), the sequence $\mathsf{AddCookie}(oldcookies, c, protocol)$ is defined by the following algorithm: Let $m :=$ *oldcookies*. If $(c.\texttt{name.1} \equiv \texttt{\_\_Host}) \wedge \neg((protocol \equiv \texttt{S}) \wedge (c.\texttt{secure} \equiv \top))$, then return m, else: Remove any c' from m that has $c.\texttt{name} \equiv c'.\texttt{name}$. Append c to m and return m.

NavigableWindows. The function NavigableWindows returns a set of windows that a document is allowed to navigate. We closely follow [1], Section 5.1.4 for this definition.

Definition 51. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

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

Algorithm 14 Web Browser Model: Determine window for navigation.

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

Algorithm 15 Web Browser Model: Determine same-origin window.

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

Functions

- The function GETNAVIGABLEWINDOW (Algorithm 14) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When it is given a window reference (nonce) *window*, this function returns a pointer to a selected window term in s':
  - If *window* is the string `_BLANK`, a new window is created and a pointer to that window is returned.
  - If *window* is a nonce (reference) and there is a window term with a reference of that value in the windows in s', a pointer $\overline{w'}$ to that window term is returned, as long as the window is navigable by the current window's document (as defined by NavigableWindows above).

  In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).
- The function GETWINDOW (Algorithm 15) takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.
- The function CANCELNAV (Algorithm 16) is used to stop any pending requests for a specific window. From the pending requests and pending DNS requests it removes any requests with the given window reference.

Algorithm 16 Web Browser Model: Cancel pending requests for given window.

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

Algorithm 17 Web Browser Model: Prepare headers, do DNS resolution, save message.

function HTTP_SEND(*reference*, *message*, *url*, *origin*, *referrer*, *referrerPolicy*, *a*, s')
  if *message*.host $\in^{\langle\rangle}$ s'.sts **then**
    let *url*.protocol := S
  let *cookies* := $\langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle \mid c \in^{\langle\rangle} s'.\texttt{cookies}[\mathit{message}.\texttt{host}]$
  $\hookrightarrow \wedge (c.\texttt{content.secure} \equiv \top \implies (\mathit{url}.\texttt{protocol} \equiv \texttt{S}))\}\rangle$
  let *message*.headers[Cookie] := *cookies*
  if *origin* $\not\equiv \bot$ **then**
    let *message*.headers[Origin] := *origin*
  if *referrerPolicy* $\equiv$ no-referrer **then**
    let *referrer* := $\bot$
  if *referrer* $\not\equiv \bot$ **then**
    if *referrerPolicy* $\equiv$ origin **then**
      let *referrer* := $\langle$URL, *referrer*.protocol, *referrer*.host, /, $\langle\rangle$, $\bot\rangle$
      → Referrer stripped down to origin.
    let *referrer*.fragment := $\bot$
    → Browsers do not send fragment identifiers in the Referer header.
    let *message*.headers[Referer] := *referrer*
  let s'.pendingDNS[ν_8] := $\langle$*reference*, *message*, *url*$\rangle$
  stop $\langle\langle s'.\texttt{DNSaddress}, a, \langle\texttt{DNSResolve}, \mathit{message}.\texttt{host}, \nu_8\rangle\rangle\rangle$, s'

Algorithm 18 Web Browser Model: Navigate a window backward.

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

Algorithm 19 Web Browser Model: Navigate a window forward.

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

Algorithm 20 Web Browser Model: Execute a script.

function RUNSCRIPT($\overline{w}$, $\overline{d}$, a, s')
  let *tree* := Clean(s', $s'.\overline{d}$)
  let *cookies* := $\langle \{ \langle c.\texttt{name}, c.\texttt{content.value} \rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
  ↪ $\wedge\ c.\texttt{content.httpOnly} \equiv \bot$
  ↪ $\wedge\ \big(c.\texttt{content.secure} \equiv \top \implies \big(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S}\big)\big)\}\rangle$
  let *tlw* ← $s'.\texttt{windows}$ **such that** *tlw* is the top-level window containing $\overline{d}$
  let *sessionStorage* := $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, \mathit{tlw}.\texttt{nonce}\rangle\right]$
  let *localStorage* := $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
  let *secrets* := $s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
  let R := $\mathsf{script}^{-1}(s'.\overline{d}.\texttt{script})$ **if possible; otherwise stop**
  let *in* := $\langle \mathit{tree}, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}, s'.\overline{d}.\texttt{scriptinputs}, \mathit{cookies},$
  ↪ $\mathit{localStorage}, \mathit{sessionStorage}, s'.\texttt{ids}, \mathit{secrets}\rangle$
  let $\mathit{state}' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, $\mathit{cookies}' \leftarrow \mathsf{Cookies}^{\nu}$, $\mathit{localStorage}' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
  ↪ $\mathit{sessionStorage}' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, $\mathit{command} \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
  ↪ $\mathit{out} := \langle \mathit{state}', \mathit{cookies}', \mathit{localStorage}', \mathit{sessionStorage}', \mathit{command}\rangle$
  ↪ **such that** $\mathit{out} := \mathit{out}^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$ **with** $(\mathit{in}, \mathit{out}^{\lambda}) \in R$
  let $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$:=
  ↪ $\langle \mathsf{CookieMerge}(s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right], \mathit{cookies}', s'.\overline{d}.\texttt{origin.protocol})\rangle$
  let $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$:= $\mathit{localStorage}'$
  let $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, \mathit{tlw}.\texttt{nonce}\rangle\right]$:= $\mathit{sessionStorage}'$
  let $s'.\overline{d}.\texttt{scriptstate}$:= state'
  let *referrer* := $s'.\overline{d}.\texttt{location}$
  let *referrerPolicy* := $s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
  let *docorigin* := $s'.\overline{d}.\texttt{origin}$
  switch *command* **do**
    case $\langle \texttt{HREF}, \mathit{url}, \mathit{hrefwindow}, \mathit{noreferrer}\rangle$
      let $\overline{w'}$:= GETNAVIGABLEWINDOW($\overline{w}$, *hrefwindow*, *noreferrer*, s')
      let *reference* := $\langle \texttt{REQ}, s'.\overline{w'}.\texttt{nonce}\rangle$
      let *req* := $\langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, \mathit{url}.\texttt{host}, \mathit{url}.\texttt{path}, \mathit{url}.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
      if *noreferrer* $\equiv \top$ **then**
        let *referrerPolicy* := noreferrer
      let s' := CANCELNAV(*reference*, s')
      call HTTP_SEND(*reference*, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, a, s')
    case $\langle \texttt{IFRAME}, \mathit{url}, \mathit{window}\rangle$
      if *window* $\equiv$ _SELF **then**
        let $\overline{w'}$:= $\overline{w}$
      else
        let $\overline{w'}$:= GETWINDOW($\overline{w}$, *window*, s')
      let *req* := $\langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, \mathit{url}.\texttt{host}, \mathit{url}.\texttt{path}, \mathit{url}.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
      let w' := $\langle \nu_5, \langle\rangle, \bot\rangle$
      let $s'.\overline{w'}.\texttt{activedocument.subwindows}$:= $s'.\overline{w'}.\texttt{activedocument.subwindows} +^{\langle\rangle} w'$
      call HTTP_SEND($\langle \texttt{REQ}, \nu_5\rangle$, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, a, s')

This algorithm is continued on the next page.

```
36:     case ⟨FORM, url, method, data, hrefwindow⟩
37:         if method ∉ {GET, POST} then
38:             stop
39:         let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s′)
40:         let reference := ⟨REQ, s′.w̄′.nonce⟩
41:         if method = GET then
42:             let body := ⟨⟩
43:             let parameters := data
44:             let origin := ⊥
45:         else
46:             let body := data
47:             let parameters := url.parameters
48:             let origin := docorigin
49:         let req := ⟨HTTPReq, ν4, method, url.host, url.path, parameters, ⟨⟩, body⟩
50:         let s′ := CANCELNAV(reference, s′)
51:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
52:     case ⟨SETSCRIPT, window, script⟩
53:         let w̄′ := GETWINDOW(w̄, window, s′)
54:         let s′.w̄′.activedocument.script := script
55:         stop ⟨⟩, s′
56:     case ⟨SETSCRIPTSTATE, window, scriptstate⟩
57:         let w̄′ := GETWINDOW(w̄, window, s′)
58:         let s′.w̄′.activedocument.scriptstate := scriptstate
59:         stop ⟨⟩, s′
60:     case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
61:         if method ∈ {CONNECT, TRACE, TRACK} ∨ xhrreference ∉ V_process ∪ {⊥} then
62:             stop
63:         if url.host ≢ docorigin.host ∨ url.protocol ≢ docorigin.protocol then
64:             stop
65:         if method ∈ {GET, HEAD} then
66:             let data := ⟨⟩
67:             let origin := ⊥
68:         else
69:             let origin := docorigin
70:         let req := ⟨HTTPReq, ν4, method, url.host, url.path, url.parameters, ⟨⟩, data⟩
71:         let reference := ⟨XHR, s′.d̄.nonce, xhrreference⟩
72:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
73:     case ⟨BACK, window⟩
74:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
75:         call NAVBACK(w̄′, s′)
76:     case ⟨FORWARD, window⟩
77:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
78:         call NAVFORWARD(w̄′, s′)
79:     case ⟨CLOSE, window⟩
80:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
81:         remove s′.w̄′ from the sequence containing it
82:         stop ⟨⟩, s′
83:     case ⟨POSTMESSAGE, window, message, origin⟩
84:         let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
85:         if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
              ↪ ∧ (origin ≢ ⊥ ⟹ s′.w̄′.documents.j̄.origin ≡ origin) then
86:             let s′.w̄′.documents.j̄.scriptinputs := s′.w̄′.documents.j̄.scriptinputs
                  ↪ +⟨⟩ ⟨POSTMESSAGE, s′.w̄.nonce, docorigin, message⟩
87:         stop ⟨⟩, s′
88:     case else
89:         stop
```

Algorithm 21 Web Browser Model: Process an HTTP response.

1: **function** PROCESSRESPONSE(*response*, *reference*, *request*, *requestUrl*, *a*, *f*, *s′*)
2: **if** `Set-Cookie` ∈ *response*.`headers` **then**
3: **for each** *c* ∈^⟨⟩ *response*.`headers`[`Set-Cookie`], *c* ∈ Cookies **do**
4: **let** *s′*.`cookies`[*request*.`host`]
↪ := AddCookie(*s′*.`cookies`[*request*.`host`], *c*, *requestUrl*.`protocol`)
5: **if** `Strict-Transport-Security` ∈ *response*.`headers` ∧ *requestUrl*.`protocol` ≡ `S` **then**
6: **let** *s′*.`sts` := *s′*.`sts` +^⟨⟩ *request*.`host`
7: **if** `Referer` ∈ *request*.`headers` **then**
8: **let** *referrer* := *request*.`headers`[`Referer`]
9: **else**
10: **let** *referrer* := ⊥
11: **if** `Location` ∈ *response*.`headers` ∧ *response*.`status` ∈ {303, 307} **then**
12: **let** *url* := *response*.`headers`[`Location`]
13: **if** *url*.`fragment` ≡ ⊥ **then**
14: **let** *url*.`fragment` := *requestUrl*.`fragment`
15: **let** *method′* := *request*.`method`
16: **let** *body′* := *request*.`body`
17: **if** `Origin` ∈ *request*.`headers`
↪ ∧ *request*.`headers`[`Origin`] ≠ ◇
↪ ∧ (⟨*url*.`host`, *url*.`protocol`⟩ ≡ ⟨*request*.`host`, *requestUrl*.`protocol`⟩
↪ ∨ ⟨*request*.`host`, *requestUrl*.`protocol`⟩ ≡ *request*.`headers`[`Origin`]) **then**
18: **let** *origin* := *request*.`headers`[`Origin`]
19: **else**
20: **let** *origin* := ◇
21: **if** *response*.`status` ≡ 303 ∧ *request*.`method` ∉ {`GET`, `HEAD`} **then**
22: **let** *method′* := `GET`
23: **let** *body′* := ⟨⟩
24: **if** ∃ $\overline{w}$ ∈ Subwindows(*s′*) **such that** *s′*.$\overline{w}$.`nonce` ≡ π_2(*reference*) **then** → Do not redirect XHRs.
25: **let** *req* := ⟨`HTTPReq`, ν_6, *method′*, *url*.`host`, *url*.`path`, *url*.`parameters`, ⟨⟩, *body′*⟩
26: **let** *referrerPolicy* := *response*.`headers`[`ReferrerPolicy`]
27: **call** HTTP_SEND(*reference*, *req*, *url*, *origin*, *referrer*, *referrerPolicy*, *a*, *s′*)
28: **else**
29: **stop** ⟨⟩, *s′*

This algorithm is continued on the next page.

```
30:   switch π1(reference) do
31:     case REQ
32:       let w̄ ← Subwindows(s′) such that s′.w̄.nonce ≡ π2(reference) if possible;
            ↪ otherwise stop  → normal response
33:       if response.body ⊀ ⟨∗, ∗⟩ then
34:         stop ⟨⟩, s′
35:       let script := π1(response.body)
36:       let scriptstate := π2(response.body)
37:       let d := ⟨ν7, requestUrl, response.headers, referrer, script, scriptstate, ⟨⟩, ⟨⟩, ⊤⟩
38:       if s′.w̄.documents ≡ ⟨⟩ then
39:         let s′.w̄.documents := ⟨d⟩
40:       else
41:         let ī ← ℕ such that s′.w̄.documents.ī.active ≡ ⊤
42:         let s′.w̄.documents.ī.active := ⊥
43:         remove s′.w̄.documents.(ī + 1) and all following documents
              ↪ from s′.w̄.documents
44:         let s′.w̄.documents := s′.w̄.documents +⟨⟩ d
45:       stop ⟨⟩, s′
46:     case XHR
47:       let w̄ ← Subwindows(s′), d̄ such that s′.d̄.nonce ≡ π2(reference)
            ↪ ∧ s′.d̄ = s′.w̄.activedocument if possible; otherwise stop
            → process XHR response
48:       let headers := response.headers − Set-Cookie
49:       let s′.d̄.scriptinputs := s′.d̄.scriptinputs +⟨⟩
          ⟨XMLHTTPREQUEST, headers, response.body, π3(reference)⟩
50:       stop ⟨⟩, s′
```

- The function HTTP_SEND (Algorithm 17) takes an HTTP request *message* as input, adds cookie and origin headers to the message, creates a DNS request for the hostname given in the request and stores the request in s'.pendingDNS until the DNS resolution finishes. *reference* is a reference as defined in Definition 45. *url* contains the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). *origin* is the origin header value that is to be added to the HTTP request.

- The functions NAVBACK (Algorithm 18) and NAVFORWARD (Algorithm 19), navigate a window backward or forward. More precisely, they deactivate one document and activate that document's preceding document or succeeding document, respectively. If no such predecessor/successor exists, the functions do not change the state.

- The function RUNSCRIPT (Algorithm 20) performs a script execution step of the script in the document $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted.

- The function PROCESSRESPONSE (Algorithm 21) is responsible for processing an HTTP response (*response*) that was received as the response to a request (*request*) that was sent earlier. *reference* is a reference as defined in Definition 45. *requestUrl* contains the URL used when retrieving the document.

  The function first saves any cookies that were contained in the response to the browser state, then checks whether a redirection is requested (Location header). If that is not the case, the function creates a new document (for normal requests) or delivers the contents of the response

to the respective receiver (for XHR responses).

Browser Relation We can now define the *relation* $R_{\text{webbrowser}}$ *of a Web browser atomic process* as follows:

Definition 52. The pair $((\langle a, f, m\rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 22 (or any of the functions called therein), when given $(\langle a, f, m\rangle, s)$ as input, terminates with **stop** M, s', i.e., with output M and s'.

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

E.8. Definition of Web Browsers

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

Definition 53 (Web Browser atomic Dolev-Yao Process). A Web browser atomic Dolev-Yao process is an atomic Dolev-Yao process of the form $p = (I^p, Z_{\text{webbrowser}}, R_{\text{webbrowser}}, s_0^p)$ for a set I^p of addresses, $Z_{\text{webbrowser}}$ and $R_{\text{webbrowser}}$ as defined above, and an initial state $s_0^p \in Z_{\text{webbrowser}}$.

Definition 54 (Web Browser Initial State). An initial state $s_0^p \in Z_{\text{webbrowser}}$ for a browser process p is a Web browser state (Definition 46) with the following properties:

- $s_0^p.\texttt{windows} \equiv \langle\rangle$
- $s_0^p.\texttt{ids} \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- $s_0^p.\texttt{secrets} \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- $s_0^p.\texttt{cookies} \equiv \langle\rangle$
- $s_0^p.\texttt{localStorage} \equiv \langle\rangle$
- $s_0^p.\texttt{sessionStorage} \equiv \langle\rangle$
- $s_0^p.\texttt{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- $s_0^p.\texttt{sts} \equiv \langle\rangle$
- $s_0^p.\texttt{DNSaddress} \in \mathsf{IPs}$ (note that this includes the possibility of using an attacker-controlled address)
- $s_0^p.\texttt{pendingDNS} \equiv \langle\rangle$
- $s_0^p.\texttt{pendingRequests} \equiv \langle\rangle$
- $s_0^p.\texttt{isCorrupted} \equiv \bot$

Note that instantiations of the Web Infrastructure Model may define different conditions for a Web browser's initial state.

E.9. Helper Functions

In order to simplify the description of scripts, we use several helper functions.

Algorithm 22 Web Browser Model: Main Algorithm.

Input: $\langle a, f, m\rangle, s$

1: **let** $s' := s$
2: **if** $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
3: **let** $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests}\rangle$ → Collect incoming messages
4: **let** $m' \leftarrow d_V(s')$
5: **let** $a' \leftarrow \mathsf{IPs}$
6: **stop** $\langle\langle a', a, m'\rangle\rangle$, s'
7: **if** $m \equiv \texttt{TRIGGER}$ **then** → A special trigger message.
8: **let** $switch \leftarrow \{\texttt{script}, \texttt{urlbar}, \texttt{reload}, \texttt{forward}, \texttt{back}\}$
9: **if** $switch \equiv \texttt{script}$ **then** → Run some script.
10: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ ↪ **if possible; otherwise stop** → Pointer to some window.
11: **let** $\overline{d} := \overline{w} +^{\langle\rangle} \texttt{activedocument}$
12: **call** RUNSCRIPT($\overline{w}$, $\overline{d}$, a, s')
13: **else if** $switch \equiv \texttt{urlbar}$ **then** → Create some new request.
14: **let** $newwindow \leftarrow \{\top, \bot\}$
15: **if** $newwindow \equiv \top$ **then** → Create a new window.
16: **let** $windownonce := \nu_1$
17: **let** $w' := \langle windownonce, \langle\rangle, \bot\rangle$
18: **let** $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
19: **else** → Use existing top-level window.
20: **let** $\overline{tlw} \leftarrow \mathbb{N}$ **such that** $s'.\overline{tlw}.\texttt{documents} \neq \langle\rangle$ ↪ **if possible; otherwise stop** → Pointer to some top-level window.
21: **let** $windownonce := s'.\overline{tlw}.nonce$
22: **let** $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$
23: **let** $host \leftarrow \mathsf{Doms}$
24: **let** $path \leftarrow \mathbb{S}$
25: **let** $fragment \leftarrow \mathbb{S}$
26: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
27: **let** $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$
28: **let** $req := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, host, path, parameters, \langle\rangle, \langle\rangle\rangle$
29: **call** HTTP_SEND($\langle \texttt{REQ}, windownonce\rangle$, req, url, $\bot$, $\bot$, $\bot$, a, s')
30: **else if** $switch \equiv \texttt{reload}$ **then** → Reload some document.
31: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ ↪ **if possible; otherwise stop** → Pointer to some window.
32: **let** $url := s'.\overline{w}.\texttt{activedocument.location}$
33: **let** $req := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
34: **let** $referrer := s'.\overline{w}.\texttt{activedocument.referrer}$
35: **let** $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\texttt{nonce}, s')$
36: **call** HTTP_SEND($\langle \texttt{REQ}, s'.\overline{w}.\texttt{nonce}\rangle$, req, url, $\bot$, $referrer$, $\bot$, a, s')
37: **else if** $switch \equiv \texttt{forward}$ **then**
38: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ ↪ **if possible; otherwise stop** → Pointer to some window.
39: **call** NAVFORWARD($\overline{w}$, s')
40: **else if** $switch \equiv \texttt{back}$ **then**
41: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ ↪ **if possible; otherwise stop** → Pointer to some window.
42: **call** NAVBACK($\overline{w}$, s')
43: **else if** $m \equiv \texttt{FULLCORRUPT}$ **then** → Request to corrupt browser
44: **let** $s'.\texttt{isCorrupted} := \texttt{FULLCORRUPT}$
45: **stop** $\langle\rangle$, s'
46: **else if** $m \equiv \texttt{CLOSECORRUPT}$ **then** → Close the browser
47: **let** $s'.\texttt{secrets} := \langle\rangle$
48: **let** $s'.\texttt{windows} := \langle\rangle$
49: **let** $s'.\texttt{pendingDNS} := \langle\rangle$

This algorithm is continued on the next page.

50: **let** s'.pendingRequests := $\langle\rangle$
51: **let** s'.sessionStorage := $\langle\rangle$
52: **let** s'.cookies $\subset^{\langle\rangle}$ Cookies **such that**
↪ $(c \in^{\langle\rangle} s'.\texttt{cookies}) \iff (c \in^{\langle\rangle} s.\texttt{cookies} \wedge c.\texttt{content.session} \equiv \bot)$
53: **let** s'.isCorrupted := CLOSECORRUPT
54: **stop** $\langle\rangle$, s'
55: **else if** $\exists \langle reference, request, url, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
↪ $\pi_1(\mathsf{dec_s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
56: **let** m' := $\mathsf{dec_s}(m, key)$
57: **if** $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
58: **stop**
59: **remove** $\langle reference, request, url, key, f\rangle$ **from** s'.pendingRequests
60: **call** PROCESSRESPONSE(m', *reference*, *request*, *url*, a, f, s')
61: **else if** $\pi_1(m) \equiv \texttt{HTTPResp} \wedge \exists \langle reference, request, url, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
↪ $m.\texttt{nonce} \equiv request.\texttt{nonce}$ **then** → Plain HTTP Response
62: **remove** $\langle reference, request, url, \bot, f\rangle$ **from** s'.pendingRequests
63: **call** PROCESSRESPONSE(m, *reference*, *request*, *url*, a, f, s')
64: **else if** $m \in$ DNSResponses **then** → Successful DNS response
65: **if** $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin$ IPs
↪ $\vee\ m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].\texttt{request.host}$ **then**
66: **stop**
67: **let** $\langle reference, message, url\rangle$:= $s.\texttt{pendingDNS}[m.\texttt{nonce}]$
68: **if** $url.\texttt{protocol} \equiv \texttt{S}$ **then**
69: **let** s'.pendingRequests := s'.pendingRequests
↪ $+^{\langle\rangle} \langle reference, message, url, \nu_3, m.\texttt{result}\rangle$
70: **let** *message* := $\mathsf{enc_a}(\langle message, \nu_3\rangle, s'.\texttt{keyMapping}\,[message.\texttt{host}])$
71: **else**
72: **let** s'.pendingRequests := s'.pendingRequests
↪ $+^{\langle\rangle} \langle reference, message, url, \bot, m.\texttt{result}\rangle$
73: **let** s'.pendingDNS := s'.pendingDNS $- m.\texttt{nonce}$
74: **stop** $\langle\langle m.\texttt{result}, a, message\rangle\rangle$, s'
75: **stop**

Algorithm 23 Function to retrieve an unhandled input message for a script.

1: **function** CHOOSEINPUT(s', *scriptinputs*)
2: **let** *iid* **such that** $iid \in \{1, \cdots, |scriptinputs|\} \wedge iid \notin^{\langle\rangle} s'.\texttt{handledInputs}$ **if possible; otherwise return** $(\bot, s')$
3: **let** $input := \pi_{iid}(scriptinputs)$
4: **let** $s'.\texttt{handledInputs} := s'.\texttt{handledInputs} +^{\langle\rangle} iid$
5: **return** $(input, s')$

Algorithm 24 Function to extract the first script input message matching a specific pattern.

1: **function** CHOOSEFIRSTINPUTPAT(*scriptinputs*, *pattern*)
2: **let** i **such that** $i = \min\{j : \pi_j(scriptinputs) \sim pattern\}$ **if possible; otherwise return** $\bot$
3: **return** $\pi_i(scriptinputs)$

CHOOSEINPUT (Algorithm 23) The state of a document contains a term, say *scriptinputs*, which records the input this document has obtained so far (via XHRs and postMessages). If the script of the document is activated, it will typically need to pick one input message from *scriptinputs* and record which input it has already processed. For this purpose, the function CHOOSEINPUT(s', *scriptinputs*) is used, where s' denotes the scripts current state. It saves the indexes of already handled messages in the scriptstate s' and chooses a yet unhandled input message from *scriptinputs*. The index of this message is then saved in the scriptstate (which is returned to the script).

CHOOSEFIRSTINPUTPAT (Algorithm 24) Similar to the function CHOOSEINPUT above, we define the function CHOOSEFIRSTINPUTPAT. This function takes the term *scriptinputs*, which as above records the input this document has obtained so far (via XHRs and postMessages, append-only), and a pattern. If called, this function chooses the first message in *scriptinputs* that matches *pattern* and returns it. This function is typically used in places, where a script only processes the first message that matches the pattern. Hence, we omit recording the usage of an input.

PARENTWINDOW To determine the nonce referencing the parent window in the browser, the function PARENTWINDOW(*tree*, *docnonce*) is used. It takes the term *tree*, which is the (partly cleaned) tree of browser windows the script is able to see and the document nonce *docnonce*, which is the nonce referencing the current document the script is running in, as input. It outputs the nonce referencing the window which directly contains in its subwindows the window of the document referenced by *docnonce*. If there is no such window (which is the case if the script runs in a document of a top-level window), PARENTWINDOW returns $\bot$.

PARENTDOCNONCE The function PARENTDOCNONCE(*tree*, *docnonce*) determines (similar to PARENTWINDOW above) the nonce referencing the active document in the parent window in the browser . It takes the term *tree*, which is the (partly cleaned) tree of browser windows the script is able to see and the document nonce *docnonce*, which is the nonce referencing the current document the script is running in, as input. It outputs the nonce referencing the active document in the window which directly contains in its subwindows the window of the document referenced by *docnonce*. If there is no such window (which is the case if the script runs in a document of a top-level window) or no active document, PARENTDOCNONCE returns *docnonce*.

SUBWINDOWS This function takes a term *tree* and a document nonce *docnonce* as input just as the function above. If *docnonce* is not a reference to a document contained in *tree*, then SUBWINDOWS(*tree*, *docnonce*) returns $\langle\rangle$. Otherwise, let $\langle$*docnonce*, *location*, $\langle\rangle$, *referrer*, *script*,

scriptstate, *scriptinputs*, *subwindows*, *active*⟩ denote the subterm of *tree* corresponding to the document referred to by *docnonce*. Then, SUBWINDOWS(*tree*, *docnonce*) returns *subwindows*.

AUXWINDOW This function takes a term *tree* and a document nonce *docnonce* as input as above. From all window terms in *tree* that have the window containing the document identified by *docnonce* as their opener, it selects one non-deterministically and returns its nonce. If there is no such window, it returns the nonce of the window containing *docnonce*.

AUXDOCNONCE Similar to AUXWINDOW above, the function AUXDOCNONCE takes a term *tree* and a document nonce *docnonce* as input. From all window terms in *tree* that have the window containing the document identified by *docnonce* as their opener, it selects one non-deterministically and returns its active document's nonce. If there is no such window or no active document, it returns *docnonce*.

OPENERWINDOW This function takes a term *tree* and a document nonce *docnonce* as input as above. It returns the window nonce of the opener window of the window that contains the document identified by *docnonce*. Recall that the nonce identifying the opener of each window is stored inside the window term. If no document with nonce *docnonce* is found in the tree *tree* or the document with nonce *docnonce* is not directly contained in a top-level window, $\diamond$ is returned.

GETWINDOW This function takes a term *tree* and a document nonce *docnonce* as input as above. It returns the nonce of the window containing *docnonce*.

GETORIGIN To extract the origin of a document, the function GETORIGIN(*tree*, *docnonce*) is used. This function searches for the document with the identifier *docnonce* in the (cleaned) tree *tree* of the browser's windows and documents. It returns the origin o of the document. If no document with nonce *docnonce* is found in the tree *tree*, $\diamond$ is returned.

GETPARAMETERS Works exactly as GETORIGIN, but returns the document's parameters instead.

E.10. DNS Servers

Definition 55. A *DNS server* d (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form

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

DNS queries are answered according to this table (if the requested DNS name cannot be found in the table, the request is ignored).

The relation $R^d \subseteq (\mathcal{E} \times \{s_0^d\}) \times (2^{\mathcal{E}} \times \{s_0^d\})$ of d above is defined by Algorithm 25.

E.11. Web Systems

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

Algorithm 25 Relation of a DNS server R^d.

Input: $\langle a, f, m\rangle, s$

1: **let** $domain, n$ **such that** $\langle \texttt{DNSResolve}, domain, n\rangle \equiv m$ **if possible; otherwise stop** $\langle\rangle, s$
2: **if** $domain \in s$ **then**
3: **let** $addr := s[domain]$
4: **let** $m' := \langle \texttt{DNSResolved}, domain, addr, n\rangle$
5: **stop** $\langle\langle f, a, m'\rangle\rangle, s$
6: **stop** $\langle\rangle, s$

Definition 56. A *Web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a tuple with its components defined as follows:

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

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

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

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

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

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

E.12. Generic HTTPS Server Model

This base model can be used to ease modeling of HTTPS server atomic processes. It defines placeholder algorithms that can be superseded by more detailed algorithms to describe a concrete relation for an HTTPS server.

Definition 57 (Base state for an HTTPS server). The state of each HTTPS server that is an instantiation of this relation must contain at least the following subterms: $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \mathsf{IPs}$ (containing the IP address of a DNS server), $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (containing a mapping from domains to public keys), $tlskeys \in [\mathsf{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $corrupt \in \mathcal{T}_{\mathcal{N}}$ (either $\perp$ if the server is not corrupted, or an arbitrary term otherwise).

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

Let ν_{n0} and ν_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. We now define the default functions of the generic Web server in Algorithms 26–30, and the main relation in Algorithm 31.

Algorithm 26 Generic HTTPS Server Model: Sending a DNS message (in preparation for sending an HTTPS message).

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

Algorithm 27 Generic HTTPS Server Model: Default HTTPS response handler.

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

Algorithm 28 Generic HTTPS Server Model: Default trigger event handler.

1: **function** PROCESS_TRIGGER(a, s')
2: **stop**

Algorithm 29 Generic HTTPS Server Model: Default HTTPS request handler.

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

Algorithm 30 Generic HTTPS Server Model: Default handler for other messages.

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

Algorithm 31 Generic HTTPS Server Model: Main relation of a generic HTTPS server

Input: $\langle a, f, m\rangle, s$

let $s' := s$
if $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
  let $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle$, s'
if $\exists\, m_{\text{dec}}$, k, k', *inDomain* **such that** $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\texttt{tlskeys}$ **then**
  let n, *method*, *path*, *parameters*, *headers*, *body* **such that**
  ↪ $\langle \texttt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
  ↪ **if possible; otherwise stop**
  call PROCESS_HTTPS_REQUEST(m_{dec}, k, a, f, s')
else if $m \in \mathsf{DNSResponses}$ **then** → Successful DNS response
  if $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin \mathsf{IPs}$
  ↪ $\vee\ m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].2.\texttt{host}$ **then**
    stop
  let *reference* $:= s.\texttt{pendingDNS}[m.\texttt{nonce}].1$
  let *request* $:= s.\texttt{pendingDNS}[m.\texttt{nonce}].2$
  let $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle \mathit{reference}, \mathit{request}, \nu_{n1}, m.\texttt{result}\rangle$
  let *message* $:= \mathsf{enc_a}(\langle \mathit{request}, \nu_{n1}\rangle, s'.\texttt{keyMapping}[\mathit{request}.\texttt{host}])$
  let $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
  stop $\langle\langle m.\texttt{result}, a, \mathit{message}\rangle\rangle$, s'
else if $\exists\, \langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
  ↪ **such that** $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
  let $m' := \mathsf{dec_s}(m, \mathit{key})$
  if $m'.\texttt{nonce} \not\equiv \mathit{request}.\texttt{nonce}$ **then**
    stop
  if $m' \notin \mathsf{HTTPResponses}$ **then**
    call PROCESS_OTHER(m, a, f, s')
  remove $\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle$ **from** $s'.\texttt{pendingRequests}$
  call PROCESS_HTTPS_RESPONSE(m', *reference*, *request*, a, f, s')
else if $m \equiv \texttt{TRIGGER}$ **then** → Process was triggered
  call PROCESS_TRIGGER(a, s')
else
  call PROCESS_OTHER(m, a, f, s')
stop

E.13. General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [6].

Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ be a Web system. In the following, we write $s_x = (S_x, E_x)$ for the states of a Web system.

Definition 58 (Emitting Events). Given an atomic process p, an event e, and a finite run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \ldots)$ we say that p *emits* e iff there is a processing step in ρ of the form

$$(S^i, E^i, N^i) \xrightarrow[p \to E]{} (S^{i+1}, E^{i+1}, N^{i+1})$$

for some $i \geq 0$ and a sequence of events E with $e \in^{\langle\rangle} E$. We also say that p *emits* m iff $e = \langle x, y, m \rangle$ for some addresses x, y.

Definition 59. We say that a term t *is derivably contained in (a term)* t' *for (a set of DY processes)* P *(in a processing step* $s_i \to s_{i+1}$ *of a run* $\rho = (s_0, s_1, \ldots)$*)* if t is derivable from t' with the knowledge available to P, i.e.,

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

Definition 60. We say that *a set of processes* P *leaks a term* t *(in a processing step* $s_i \to s_{i+1}$*) to a set of processes* P' if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.

Definition 61. We say that a DY process p *created* a message m in a processing step

$$(S^i, E^i, N^i) \xrightarrow[p \to E_{\text{out}}]{e_{\text{in}} \to p} (S^{i+1}, E^{i+1}, N^{i+1})$$

of a run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ if all of the following hold true

- m is a subterm of one of the events in E_{out}
- m is and was not derivable by any other set of processes

$$m \notin d_\emptyset\Big(\bigcup_{\substack{p' \in \mathcal{W} \setminus \{p\} \\ 0 \leq j \leq i+1}} S^j(p') \Big)$$

We note a process p creating a message does not imply that p can derive that message.

Definition 62. We say that *a browser* b *accepted* a message (as a response to some request) if the browser decrypted the message (if it was an HTTPS message) and called the function PROCESSRESPONSE, passing the message and the request (see Algorithm 21).

Definition 63. We say that an atomic DY process p *knows a term* t in some state $s = (S, E, N)$ of a run if it can derive the term from its knowledge, i.e., $t \in d_\emptyset(S(p))$.

Definition 64. Let $N \subseteq \mathcal{N}$, $t \in \mathcal{T}_N(X)$, and $k \in \mathcal{T}_N(X)$. We say that k *appears only as a public key* in t, if

1. If $t \in N \cup X$, then $t \neq k$

2. If $t = f(t_1, \ldots, t_n)$, for $f \in \Sigma$ and $t_i \in \mathcal{T}_{\mathcal{N}}(X)$ $(i \in \{1, \ldots, n\})$, then $f = \mathsf{pub}$ or for all t_i, k appears only as a public key in t_i.

Definition 65. We say that a *script initiated a request* r if a browser triggered the script (in Line 10 of Algorithm 20) and the first component of the *command* output of the script relation is either `HREF`, `IFRAME`, `FORM`, or `XMLHTTPREQUEST` such that the browser issues the request r in the same step as a result.

Definition 66. We say that *an instance of the generic HTTPS server* s *accepted* a message (as a response to some request) if the server decrypted the message (if it was an HTTPS message) and called the function PROCESS_HTTPS_RESPONSE, passing the message and the request (see Algorithm 31).

For a run $\rho = s_0, s_1, \ldots$ of any $\mathcal{WS}$, we state the following lemmas:

Lemma 13. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b

(I) emits an HTTPS request of the form

$$m = \mathsf{enc_a}(\langle req, k \rangle, \mathsf{pub}(k'))$$

(where *req* is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and

(II) in the initial state s_0, for all processes $p \in \mathcal{W} \setminus \{u\}$, the private key k' appears only as a public key in $S^0(p)$, and

(III) u never leaks k',

then all of the following statements are true:

(1) There is no state of $\mathcal{WS}$ where any party except for u knows k', thus no one except for u can decrypt m to obtain *req*.

(2) If there is a processing step $s_j \to s_{j+1}$ where the browser b leaks k to $\mathcal{W} \setminus \{u, b\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ or the browser is fully corrupted in s_j.

(3) The value of the host header in *req* is the domain that is assigned the public key $\mathsf{pub}(k')$ in the browsers' keymapping $s_0.\texttt{keyMapping}$ (in its initial state).

(4) If b accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and b is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ prior to s_j, then u created the HTTPS response m' to the HTTPS request m, i.e., the nonce of the HTTP request *req* is not known to any atomic process p, except for the atomic processes b and u.

Proof. **(1)** follows immediately from the preconditions.

The process u never leaks k', and initially, the private key k' appears only as a public key in all other process states. As the equational theory does not allow the extraction of a private key x from a public key $\mathsf{pub}(x)$, the other processes can never derive k'.

Thus, even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u, and k' cannot be known to any other party. Thus, nobody except for u can derive *req* from m.

(2) We assume that b leaks k to $\mathcal{W} \setminus \{u, b\}$ in the processing step $s_j \to s_{j+1}$ without u prior leaking the key k to anyone except for u and b and that the browser is not fully corrupted in s_j, and lead this to a contradiction.

The browser is honest in s_i. From the definition of the browser b, we see that the key k is always chosen as a fresh nonce (placeholder ν_3 in Lines 64ff. of Algorithm 22) that is not used anywhere else. Further, the key is stored in the browser's state in *pendingRequests*. The information from *pendingRequests* is not extracted or used anywhere else (in particular it is not accessible by scripts). If the browser becomes closecorrupted prior to s_j (and after s_i), the key cannot be used anymore (compare Lines 46ff. of Algorithm 22). Hence, b does not leak k to any other party in s_j (except for u and b). This proves (2).

(3) Per the definition of browsers (Algorithm 22), a host header is always contained in HTTP requests by browsers. From Line 70 of Algorithm 22 we can see that the encryption key for the request *req* was chosen using the host header of the message. It is chosen from the `keyMapping` in the browser's state, which is never changed during ρ. This proves (3).

(4) An HTTPS response m' that is accepted by b as a response to m has to be encrypted with k. The nonce k is stored by the browser in the *pendingRequests* state information. The browser only stores freshly chosen nonces there (i.e., the nonces are not used twice, or for other purposes than sending one specific request). The information cannot be altered afterwards (only deleted) and cannot be read except when the browser checks incoming messages. The nonce k is only known to u (which did not leak it to any other party prior to s_j) and b (which did not leak it either, as u did not leak it and b is honest, see (2)). The browser b cannot send responses. This proves (4).

Corollary 1. In the situation of Lemma 13, as long as u does not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ and the browser does not become fully corrupted, k is not known to any DY process $p \notin \{u, b\}$ (i.e., $\nexists\, s' = (S', E') \in \rho\colon\ k \in d_{N^p}(S'(p))$).

Lemma 14. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\texttt{windows}$ with the origin $\langle dom, \mathsf{S}\rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then b extracted (in Line 37 in Algorithm 21) the script in that document from an HTTPS response that was created by p.

Proof. The origin of the document d is set only once: In Line 37 of Algorithm 21. The values (domain and protocol) used there stem from the information about the request (say, *req*) that led to the loading of d. These values have been stored in *pendingRequests* between the request and the response actions. The contents of *pendingRequests* are indexed by freshly chosen nonces and can never be altered or overwritten (only deleted when the response to a request arrives). The information about the request *req* was added to *pendingRequests* in Line 69 (or Line 72 which we can exclude as we will see later) of Algorithm 22. In particular, the request was an HTTPS request iff a (symmetric) key was added to the information in *pendingRequests*. When receiving the response to *req*, it is checked against that information and accepted only if it is encrypted with the proper key and contains the same nonce as the request (say, n). Only then the protocol part of the origin of the newly created document becomes `S`. The domain part of the origin (in our case *dom*) is taken directly from the *pendingRequests* and is thus guaranteed to be unaltered.

From Line 70 of Algorithm 22 we can see that the encryption key for the request *req* was actually chosen using the host header of the message which will finally be the value of the origin of the document d. Since b therefore selects the public key $S_i(b).\texttt{keyMapping}[dom] = S_0(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ for p (the key mapping cannot be altered during a run), we can see that *req* was encrypted using a public key that matches a private key which is only (if at all) known to p. With Lemma 13 we see that the symmetric encryption key for the response, k, is only known to b and the respective

Web server. The same holds for the nonce n that was chosen by the browser and included in the request. Thus, no other party than p can encrypt a response that is accepted by the browser b and which finally defines the script of the newly created document.

Lemma 15. If in a processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, \mathtt{S} \rangle$ where $S_i(b).\mathtt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then

- that request was initiated by a script that b extracted (in Line 37 in Algorithm 21) from an HTTPS response that was created by p, or
- that request is a redirect to a response of a request that was initiated by such a script.

Proof. The browser algorithms create HTTP requests with an origin header by calling the HTTP_SEND function (Algorithm 17), with the origin being the fourth input parameter. This function adds the origin header only if this input parameter is not $\perp$.

The browser calls the HTTP_SEND function with an origin that is not $\perp$ only in the following places:

- Line 51 of Algorithm 20
- Line 72 of Algorithm 20
- Line 27 of Algorithm 21

■

In the first two cases, the request was initiated by a script. The Origin header of the request is defined by the origin of the script's document. With Lemma 14 we see that the content of the document, in particular the script, was indeed provided by p.

In the last case (Location header redirect), as the origin is not $\diamond$, the condition of Line 17 of Algorithm 21 must have been true and the origin value is set to the value of the origin header of the request. In particular, this implies that an origin header does not change during redirects (unless set to $\diamond$; in this case, the value stays the same in the subsequent redirects). Thus, the original request must have been created by the first two cases shown above.

The following lemma is similar to Lemma 13, but is applied to the generic HTTPS server (instead of the Web browser).

Lemma 16. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest instance s of the generic HTTPS server model

(I) emits an HTTPS request of the form

$$m = \mathsf{enc_a}(\langle req, k \rangle, \mathsf{pub}(k'))$$

(where req is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and

(II) in the initial state s_0, for all processes $p \in \mathcal{W} \setminus \{u\}$, the private key k' appears only as a public key in $S^0(p)$,

(III) u never leaks k',

(IV) the instance model s does not read or write the *pendingRequests* subterm of its state,

(V) the instance model s does not emit messages in $\mathsf{HTTPSRequests}$,

(VI) the instance model s does not change the values of the *keyMapping* subterm of its state, and

(VII) when receiving HTTPS requests of the form $\mathsf{enc_a}(\langle req', k_2\rangle, \mathsf{pub}(k'))$, u uses the nonce of the HTTP request req' only as nonce values of HTTPS responses encrypted with the symmetric key k_2,

(VIII) when receiving HTTPS requests of the form $\mathsf{enc_a}(\langle req', k_2\rangle, \mathsf{pub}(k'))$, u uses the symmetric key k_2 only for symmetrically encrypting HTTP responses (and in particular, k_2 is not part of a payload of any messages sent out by u),

then all of the following statements are true:

(1) There is no state of $\mathcal{WS}$ where any party except for u knows k', thus no one except for u can decrypt m to obtain *req*.

(2) If there is a processing step $s_j \to s_{j+1}$ where some process leaks k to $\mathcal{W} \setminus \{u, s\}$, there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ or the process s is corrupted in s_j.

(3) The value of the host header in *req* is the domain that is assigned the public key $\mathsf{pub}(k')$ in $S^0(s).\texttt{keyMapping}$ (i.e., in the initial state of s).

(4) If s accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and s is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ prior to s_j, then u created the HTTPS response m' to the HTTPS request m, i.e., the nonce of the HTTP request *req* is not known to any atomic process p, except for the atomic processes s and u.

Proof. **(1)** follows immediately from the preconditions. The proof is the same as for Lemma 13:

The process u never leaks k', and initially, the private key k' appears only as a public key in all other process states. As the equational theory does not allow the extraction of a private key x from a public key $\mathsf{pub}(x)$, the other processes can never derive k'.

Thus, even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u, and k' cannot be known to any other party. Thus, nobody except for u can derive *req* from m.

(2) We assume that some process leaks k to $\mathcal{W} \setminus \{u, s\}$ in the processing step $s_j \to s_{j+1}$ without u prior leaking the key k to anyone except for u and s and that the process s is not corrupted in s_j, and lead this to a contradiction.

The process s is honest in s_i. s emits HTTPS requests like m only in Line 18 of Algorithm 31:

- The message emitted in Line 3 of Algorithm 26 has a different message structure
- As s is honest, it does not send the message of Line 6 of Algorithm 31
- There is no other place in the generic HTTPS server model where messages are emitted and due to precondition (V), the application-specific model does not emit HTTPS requests. ∎

The value k, which is the placeholder ν_{n1} in Algorithm 31, is only stored in the *pendingRequests* subterm of the state of s, i.e., in $S^{i+1}(s).\texttt{pendingRequests}$. Other than that, s only accesses this value in Line 19 of Algorithm 31, where it is only used to decrypt the response in Line 20 (in particular, the key is not propagated to the application-specific model, and the key cannot be

contained within the payload of an response due to (VIII)). We note that there is no other line in the model of the generic HTTPS server where this subterm is accessed and the application-specific model does not access this subterm due to precondition (IV). Hence, s does not leak k to any other party in s_j (except for u and s). This proves (2).

(3) From Line 16 of Algorithm 31 we can see that the encryption key for the message m was chosen using the host header of the request. It is chosen from the `keyMapping` subterm of the state of s, which is never changed during ρ by the HTTPS server and never changed by the application-specific model due to precondition (VI). This proves (3).

(4)

Response was encrypted with k. An HTTPS response m' that is accepted by s as a response to m has to be encrypted with k:

The decryption key is taken from the *pendingRequests* subterm of its state in Line 19 of Algorithm 31, where s only stores fresh nonces as keys that are added to requests as symmetric keys (see also Lines 15 and 16). The nonces (symmetric keys) are not used twice, or for other purposes than sending one specific request.

Only s and u can create the response. As shown previously, only s and u can derive the symmetric key (as s is honest in s_j). Thus, m' must have been created by either s or u.

s cannot have created the response. We assume that s emitted the message m' and lead this to a contradiction.

The generic server algorithms of s (when being honest) emit messages only in two places: In Line 3 of Algorithm 26, where a DNS request is sent, and in Line 18 of Algorithm 31, where a message with a different structure than m' is created (as m' is accepted by the server, m' must be a symmetrically encrypted ciphertext).

Thus, the instance model of s must have created the response m'.

Due to Precondition (IV), the instance model of s cannot read the `pendingRequests` subterm of its state. The symmetric key is generated freshly by the generic server algorithm in Lines 15 and 16 of Algorithm 31 and stored only in `pendingRequests`.

As the generic algorithms do not call any of the handlers with a symmetric key stored in `pendingRequests`., it follows that the instance model derived the key from a message payload in the instantiation of one of the handlers. Let $\tilde{m}$ denote this message payload.

As the server instance model cannot derive the symmetric key without processing a message from which it can derive the symmetric key, and as the server algorithm only create the original request m as the only message with the symmetric key as a payload, it follows that u must have created $\tilde{m}$, as no other process can derive the symmetric key from m.

However, when receiving m, u will use the symmetric key only as an encryption key, and in particular, will not create a message where the symmetric key is a payload (Precondition (VIII)).

Thus, the symmetric key cannot be derived by the instance of the server model, which is a contradiction to the statement that the instance model of s must have created the response m'.