

Technical Report

Formal Security Analysis of the OpenID for Verifiable Presentations Specification

Final Report on Deliverable A.1(A)

Fabian Hauck Pedram Hosseyni Ralf Küsters Tim Würtele

{fabian.hauck,pedram.hosseyni,ralf.kuesters,tim.wuerтеле}@sec.uni-stuttgart.de

Institute of Information Security – University of Stuttgart, Germany

April 9, 2025

We here report on our results for Deliverable A.1(A), i.e., formal modeling of the protocol as well as identification and formalization of security properties for the *OpenID for Verifiable Presentations* specification when used over the *Digital Credentials API*.

1. Introduction

Overall, the goal of this analysis is to study the security of the OpenID for Verifiable Presentations (OID4VP) [OID4VP] specification when used over the Digital Credentials API (DC API) [2]. More specifically, on a very high level, we aim to prove that using OID4VP over the DC API fulfills a basic “claims unforgeability” property, i.e., an attacker cannot convince an honest verifier to have the properties/claims asserted by an honest issuer for an honest user in a credential issued to an honest wallet.

The underlying WIM (*Web Infrastructure Model*) methodology has been successfully used on various Web protocols in the past, uncovering previously unknown attacks and often resulting in new standards to mitigate them. Many of these protocols are at least somewhat related to OIDF standards, for example: OAuth 2.0 [5, 11], Open ID Connect [5, 12], OpenID FAPI 1.0 [6], OpenID FAPI 2.0 [15], and Mozilla’s BrowserID [8, 9]. One of the key strengths of the WIM methodology lies in identifying errors on the protocol specification level – i.e., a WIM analysis usually assumes that a protocol specification has been implemented correctly w.r.t. the specification(s) and excludes attack vectors like Cross-Site Scripting or vulnerabilities in cryptographic primitives and their implementation (e.g., side-channel attacks), as such attacks are usually outside the scope of protocol specifications.

On a high level, a WIM analysis consists of three key steps:

Modeling Carefully define a detailed, mathematical model of the protocol as specified, taking into account any options left as implementation choices, such that the model covers all possible protocol executions that are not explicitly prohibited by the specification(s). This may include “strange” choices, such as a browser possibly sending the same authentication request multiple times to different wallets. Furthermore, this model is built such that it allows for an arbitrary number of protocol participants with arbitrary trust relations, running arbitrarily

many protocol instances in parallel, this covering all possible interleavings of protocol sessions. To account for implementation and user choices, the model usually uses non-deterministic selection; for example, when creating a presentation, the wallet model non-deterministically selects the credential to use. Overall, the final model induces an infinite set of runs, i.e., sequences of steps, of the model, representing possible runs of the modeled protocol.

Formulate Security Properties Using security goals explicitly stated in the specification(s) and/or implicit goals, formulate formal security properties w.r.t. the formal protocol model. These properties are mathematical statements about the model, or more precisely, about the set of possible model runs, i.e., they can be proven (in a mathematical sense) to be true or false. This step also includes defining a suitable attacker model – depending on the analyzed protocol, a more or less powerful attacker may be sensible. Two common attacker models (not just with the WIM, but in protocol security research as a whole) are network attackers and web attackers – whereas the network attacker completely controls the communication network, the web attacker only controls an arbitrary, but finite number of protocol participants. Both attackers may corrupt any party at any time, e.g., to account for successful hacking attacks. Of course, the security properties will typically require certain parties (e.g., the browser and wallet used in a given flow) to be honest, but do not limit corruption of all other parties.

Proofs Finally, one tries to prove the formulated security properties. This means proving that all the (infinitely many) possible model runs fulfill all security properties, i.e., these proofs must consider all possible protocol executions at the same time. The latter is the reason why we sometimes say that the WIM is a *possibilistic* model: it considers all possible executions at once.

The remainder of this report is structured as follows: in [Section 2](#), we describe important modeling decisions and assumptions made in our model; in [Section 3](#), we suggest some improvements to the OID4VP specification. [Section 4](#) contains informal descriptions of the security properties we propose to analyze. Finally, the appendices contain the full formal model and the formalized security properties. Note that throughout this document, we assume the reader to be familiar with the OID4VP [OID4VP] and DC API [2] specifications, as well as many of the documents referenced by them.

2. Assumptions and Modeling Decisions

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 OID4VP-specific assumptions, we give some background information and descriptions of assumptions inherent to the WIM methodology – while some of these are rather strong assumptions, we note that (1) they are standard assumptions in security protocol research,

and – more importantly – (2) the WIM methodology has been successfully applied to a wide range of protocols like OAuth 2.0 [5, 11], OIDC [5, 12], both FAPI versions [6, 7, 14, 15], Mozilla’s (now inactive) BrowserID [8, 9, 16], etc. Hence, the WIM methodology (evidently) provides useful, yet sufficiently precise, abstractions.

Cryptography The WIM is a symbolic, Dolev-Yao-style [4] model, i.e., bytestrings of any kind are represented as formal terms over a set of function symbols (e.g., $\text{sig}(\cdot, \cdot)$, $\text{enc}_s(\cdot, \cdot)$), 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. We use such nonces whenever the specifications require high-entropy values. Constants, on the other hand, are considered to be publicly known, e.g., domains, fixed strings, network addresses, and fixed URI paths. Additionally, the semantics of cryptographic primitives are defined by an equational theory (see Definition 17 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 other tokens. Instead, one considers all these values as being valid forever.

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. Credential Format

In our model, we strive to make only minimal assumptions on the credential format, but of course, we need to have some concrete model of credentials. Hence, we analyzed the credential formats listed in the OID4VP specification and found that they share common characteristics: First of all, credentials must of course be verifiable, hence, a (valid) credential in our model is signed by its issuer. Furthermore, a credential is only meaningful if it contains some claim about the credential’s subject (e.g., the email address of a user). Therefore, our modeled credentials contain an *id*, which is our abstract representation of a user’s identity at a certain issuer. Note that while this *id* technically models a user’s identity, it just serves as a placeholder for arbitrary claims about the credential subject. And finally, following the assumption “that a Verifiable Credential is always presented with a cryptographic proof of possession” from [OID4VP, Section 14.1], a valid credential in our model contains a public key, the holder key, to facilitate such a proof of possession.

To validate a given credential, the verifier of course also needs some idea of which issuer to expect, i.e., we need a way for the verifier to determine which issuer(s) it is willing to accept for a given credential. In our model, the aforementioned *id* contains an issuer-local part, and a domain, similar

to an email address. The domain then identifies the expected issuer for a credential for *id*, i.e., the verifier selects the key to validate the issuer’s signature based on that domain. In our model, these validation keys are preconfigured, i.e., verifiers are initialized with public keys for the issuers they trust.

2.2. Credential Issuance

As agreed upon with the OIDF, for this first Deliverable, we do not model the issuance protocol(s). Instead, we initialize the wallets in our model such that they already hold a set of (valid) credentials at the beginning of a model run.

In this initial state, all credentials in all wallets’ states are valid and signed by the “correct” issuer (see [Section 2.1](#)). However, in a real ecosystem, an honest wallet may of course also contain corrupt credentials; we distinguish the following types of such corrupt credentials: (1) Structurally sound credentials signed by a corrupted issuer; these follow the correct credential format, but may contain any *id* and public key. (2) Structurally unsound credentials. (3) Credentials for an attacker-controlled identity/claims about the attacker that are signed by an honest issuer; we assume that the credential issuance process ensures that such a credential cannot contain the public key of an honest wallet, e.g., because the credential issuance process requires a proof of possession of the holder key.

To account for such corrupt credentials, our wallet model contains an endpoint where it accepts arbitrary messages and stores the message contents in its credential store. This allows the attacker in our model to “inject” arbitrary things into an honest wallet (as long as the attacker can *derive* (see above) that thing), thus covering the three cases described above: (1) Initially honest issuers may become corrupted during a model run, therefore leaking their private signing keys to the attacker which can subsequently create arbitrary credentials in that issuer’s name. (3) Likewise, an initially honest wallet (that holds credentials for an *id*) may become corrupted during a protocol run, which (a) makes that *id* an attacker-controlled identity, and (b) leaks this wallet’s credentials (that may be issued by an honest issuer) to the attacker. And finally, for (2), the attacker can, at any point in time, create some bitstring (formally speaking: term) that is not a well-formed credential.

Note that this modeling of corrupted credentials will generally break session integrity properties (depending on their exact formulation). However, as agreed upon with the OIDF, session integrity properties are out of scope for this part of the analysis, hence, we avoid restricting the attacker’s abilities in that regard.

To allow issuers to become corrupted by the attacker, our model does contain issuers, but they are nothing more than a “database” storing the issuer’s signing key – that may then at some point become corrupted by the attacker.

2.3. Presentation Format

Similar to credentials, the OID4VP and DC API specifications are format-agnostic w.r.t. presentations. However, we of course need to have some representation of presentations in our model, but want to make minimal assumptions, i.e., keep it as generic as possible. Hence, we analyzed the different presentation formats described in the OID4VP specification and found the following common characteristics: First of all, a presentation “is derived from one or more Verifiable Credentials” [OID4VP, [Section 2](#)]. Hence, presentations in our model contain a credential (see [Section 2.1](#)). Second, a verifiable presentation needs to be signed by the credential holder [OID4VP, [Section 2](#)], therefore, presentations in our model are signed (by the holder in case of an honestly created presentation). And finally, since credentials must be presented with a proof of possession that is bound to an intended audience and linked to a transaction [OID4VP, [Section 14.1](#)], presentations in our model include both the audience and the nonce from the authentication request.

Hence, a presentation in our model consists of a credential, an intended audience, a nonce, and a signature over these three values.

2.4. DC API

As of this writing, the DC API specification [2] is a very early draft with many details still missing. Consequently, besides the “usual” modeling decisions, we have to make several assumptions on how the DC API operates that we describe in the following.

2.4.1. Same-Device and Cross-Device Requests

The DC API itself supports both same-device and cross-device requests. However, except for the need to prevent cross-device phishing attacks, e.g., by establishing proximity between the involved parties, both types of flows carry the same information. Hence, the same-device flow can be seen as a special case of cross-device flow where both devices happen to be the same entity. This in turn allows us to keep our model slightly simpler by only modeling cross-device flows.

2.4.2. Communication Model and Cross-Device Proximity

For communication between two instances of the DC API on different devices, we model a symmetrically encrypted channel where the encryption keys are fresh for each request/response pair; and the DC API instances only accept one response per request (by invalidating the encryption key after accepting a response).

These keys also serve as our model of proximity between parties: in the initial state of the model, these keys are distributed among all parties (including the attacker) according to a **nearby** predicate that captures whether two parties are nearby each other. I.e., only parties that are nearby each other share such keys; and consequently, any given party’s DC API instance can only decrypt (and therefore only accept) requests that have been sent by a nearby party’s DC API instance.

2.4.3. Browser DC API Only Available for HTTPS Origins

We assume that the browser implementation of the DC API will only accept authorization requests from scripts running under an HTTPS origin. We note that while the DC API specification is in a very early draft stage, its WebIDL definition of the `DigitalCredential` interface in [2, Section 7] hints at the intent of only making this API available in secure contexts.

2.4.4. Wallet Selection by the DC API

When processing an authorization request, the receiving DC API instance needs to forward that request to a wallet application of the user’s choice [OID4VP, Appendix A]. In our model, we take into account the possibility of an attacker wallet application being installed on an otherwise honest device. In other words: the DC API instance may forward an authorization request to the attacker wallet (and subsequently accept whatever response that attacker wallet provides).

This is modeled by a non-deterministic decision within the DC API instance model: when processing an authorization request, the model either forwards the request to an honest wallet or publishes the request to the network and awaits a response from there (recall: we assume a network attacker, i.e., the attacker can read anything that is sent on the network, and they can schedule and inject messages).

2.5. Client ID Schemes

As mentioned in [Section 2.1](#), verifiers in our model identify issuers based on domains, and have the necessary metadata configured in their initial states. Likewise, we identify verifiers by domains and distribute verifier metadata, and in particular the verifier signing key to validate signed requests, in the initial model state. I.e., as agreed upon, our model does not include the actual client metadata fetching by the wallet and instead assumes that the metadata fetching works as intended: given a client identifier (including the client ID scheme), a party can obtain the correct metadata for that client identifier. Note that this implicitly assumes that client IDs are unambiguous from the wallet’s point of view.

2.6. Authorization Request

In the following, we describe our modeling decisions and assumptions regarding authorization requests.

Nonce Following [\[OID4VP, Section 14.1\]](#), our verifier model chooses a fresh random value for each authorization request and includes that value in the request’s `nonce` parameter. However, our understanding of [\[OID4VP, Section 14.1\]](#) is such that the verifier is not mandated to bind this `nonce` to any particular browser session. Consequently, our verifier model does not include such a binding.

Scope/Credential Query Languages We do not model the different credential query languages from [\[OID4VP, Section 5.1\]](#) and we also do not explicitly model the credential selection process on the wallet side. Instead, our wallet model non-deterministically selects one of the credentials from its credential store when processing an authorization request. Recall that the WIM is a possibilistic model, therefore, this simplification is safe over-approximation, as our analysis always includes the case in which the wallet selected the “correct” credential, i.e., one that matches what the verifier requested. Additionally, this modeling subsumes all possible credential choice algorithms.

Request Signing and Encryption Our verifier and wallet models support both signed and unsigned authorization requests. When creating a new request, the verifier model non-deterministically decides whether to sign the request.

We do not model encrypted requests, this is a safe over-approximation: Whatever checks are required on “plain” requests are also required on encrypted ones, i.e., if anything, encrypted requests are more secure.

Expected Origins in Signed Requests As mandated by [\[OID4VP, Appendix A.2\]](#), our verifier model includes the `expected_origins` parameter in signed authorization requests. However, since the specification does not mandate the wallet to actually compare the origin provided by the DC API to this parameter’s contents, our model does not include such a check.

2.7. Authorization Response

In the following, we describe our modeling decisions and assumptions regarding authorization responses.

Signed/Encrypted Responses We do not model the `dc_api.jwt` response mode, i.e., wallets never sign or encrypt authorization responses. Instead, our model always uses the `dc_api` response mode. Note that this is a safe over-approximation, because the verifier needs to perform the same checks on the contents of signed/encrypted and plain responses.

Response Types As agreed upon, in this first deliverable, we only model the `vp_token` response type, i.e., our modeled verifiers only ever request `vp_token` responses, and responses from our wallet model always contain a `vp_token`.

2.8. Further Notable Modeling Details

In the following, we list additional notable details of our model.

- As agreed with the OIDF, the authorization requests in our model do not contain the `transaction_data` parameter.
- As agreed with the OIDF, we also do not model verifier attestation.
- All authorization request parameters are passed directly in the request rather than through a URL pointing to the value (JAR). This approach simplifies the analysis safely, i.e., this is an over-approximation, as these URLs must use the HTTPS scheme [RFC9101, Section 5.2], and an honest verifier always provides a URL under its own control that returns the correct parameters.
- To make the proofs in the WIM simpler and clearer, we do not model the CA infrastructure of the internet. Instead, every party has the public keys for each domain pre-configured in its state. This simplification, of course, means that we assume that the key bindings asserted by a CA are always correct, i.e., we assume that the web PKI works as intended.

2.9. Specification Versions

Initially, it was agreed to analyze draft version 24 of the OID4VP specification in conjunction with the current state of the DC API draft [2]. However, during the modeling phase, relevant parts of the OID4VP specification were updated, including a change to the way wallets choose the audience value for presentations sent over the DC API. Hence, we agreed with the WG chairs to rather use the latest editor's draft as of March 27, 2025, corresponding to commit b0d6812 in the WG's repository [OID4VP]. We note that w.r.t. the texts relevant to our model, this version is identical to draft version 25.¹

3. Notes on the OID4VP Specification

Based on our work with the OID4VP specification, we suggest the working group consider the following changes.

Section 14.1 and DC API [OID4VP, Section 14.1] should be updated to incorporate OID4VP over the DC API. Paragraph 3, for example, says that the audience value must be the client ID but in this case the audience value is always the origin asserted by the DC API.

Expected Origins Parameter [OID4VP, Appendix A.2] introduces a new authentication request parameter, `expected_origins` for signed requests over the DC API. However, the specification does not explicitly require the wallet to verify that the origin asserted by the DC API is included in this set.

Notably, in draft version 24, the wallet used the client ID as the audience in the verifiable presentation. Without verifying whether the asserted origin is within `expected_origins`, a malicious verifier could replay an authentication request from an honest verifier. This would allow the attacker

¹Available at https://openid.net/specs/openid-4-verifiable-presentations-1_0-25.html

to obtain a verifiable presentation containing claims about a legitimate user, which is accepted by an honest verifier as depicted in Figure 1.

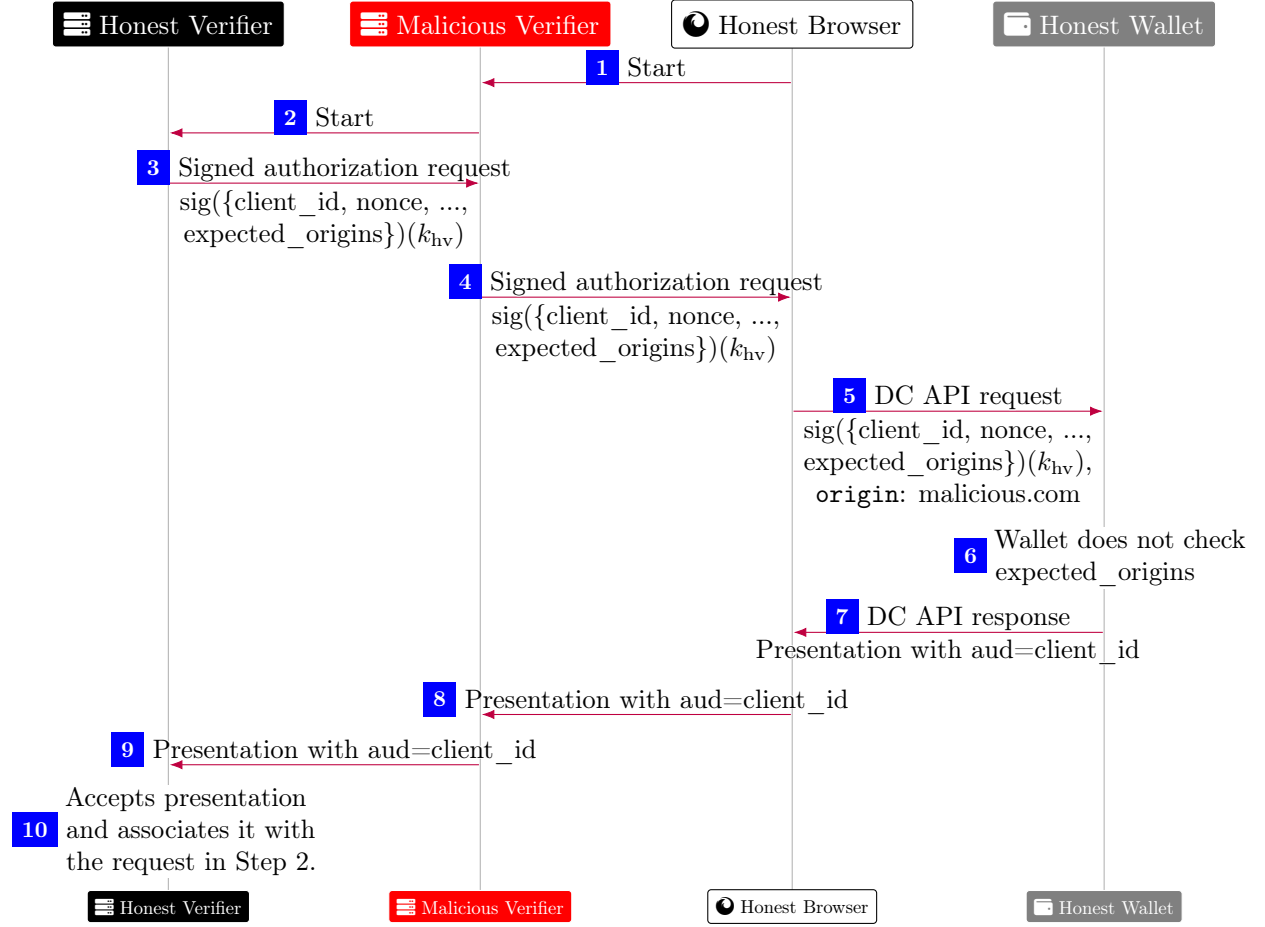


Figure 1: Attack on draft version 24 if `expected_origins` is not validated: The honest verifier associates the honest wallet’s presentation with the session between the honest and the malicious verifier.

4. Informal Security Properties

In the following, we give an informal overview of the security properties that we propose to analyze. We note that these are derived from the agreement/contract between the OIDF and the University of Stuttgart. See [Appendix B](#) for the formalized properties.

4.1. Verifier Authentication

This property captures that when an honest wallet responds to a signed authorization request that contains an honest verifier’s client ID, then that request really did originate from the verifier identified by the aforementioned client ID.

This implies that any verifier metadata from the authorization request that the wallet may show to the user to help them decide whether to approve the request also originated from that verifier.

4.2. Wallet Authorization

With this property, we capture that whenever an honest verifier accepts a presentation, and the wallet that owns the holder key of that credential is honest, then that wallet issued the presentation.

4.3. Claims Unforgeability

On a high level, this property captures that a presentation for a credential by an honest issuer that was created by an honest wallet, and is accepted by an honest verifier, cannot be (successfully) used by an attacker at an honest verifier. In a bit more detail, given that (1) a presentation is signed with an honest wallet's holder key, (2) the credential within that presentation was issued by an honest issuer, (3) an honest verifier accepted the presentation, and (4) all parties in the proximity of the wallet are honest, no attacker-controlled party can convince an honest verifier to have the properties asserted by the credential's claims.

References

- [RFC7617] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. Sept. 2015. DOI: [10.17487/RFC7617](https://doi.org/10.17487/RFC7617). URL: <https://www.rfc-editor.org/info/rfc7617>.
- [RFC9101] N. Sakimura, J. Bradley, and M. B. Jones. *The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)*. RFC 9101. Aug. 2021. DOI: [10.17487/RFC9101](https://doi.org/10.17487/RFC9101). URL: <https://www.rfc-editor.org/info/rfc9101>.
- [OID4VP] O. Terbu, T. Lodderstedt, K. Yasuda, and T. Looker. *OpenID for Verifiable Presentations. Editor's Draft, at commit b0d6812*. OpenID Foundation, Mar. 27, 2025. URL: https://github.com/openid/OpenID4VP/blob/b0d6812/openid-4-verifiable-presentations-1_0.md.
- [1] R. Berjon et al., eds. *HTML5, W3C Recommendation*. Oct. 28, 2014. URL: <http://www.w3.org/TR/html5/>.
- [2] M. Caceres, T. Cappalli, and S. Goto. *Digital Credentials. Editor's Draft*. World Wide Web Consortium, Feb. 20, 2025. URL: <https://wicg.github.io/digital-credentials/>.
- [3] 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>.
- [4] D. Dolev and A. C. Yao. "On the Security of Public-Key Protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208.
- [5] D. Fett. "An Expressive Formal Model of the Web Infrastructure". PhD thesis. 2018.
- [6] D. Fett, P. Hosseyni, and R. Küsters. "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](https://doi.org/10.1109/sp.2019.00067).
- [7] D. Fett, P. Hosseyni, and R. Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-grade API". In: *CoRR* abs/1901.11520 (2019). arXiv: [1901.11520](https://arxiv.org/abs/1901.11520). URL: <http://arxiv.org/abs/1901.11520>.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] D. Fett, R. Küsters, and G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *ACM CCS*. 2016, pp. 1204–1215.
- [12] D. Fett, R. Küsters, and G. Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *CSF*. 2017.
- [13] F. Hauck. "OpenID for Verifiable Credentials: formal security analysis using the Web Infrastructure Model". In: (2023). URL: <http://elib.uni-stuttgart.de/handle/11682/13791>.

- [14] P. Hosseyni, R. Küsters, and T. Würtele. “Formal Security Analysis of the OpenID FAPI 2.0 Family of Protocols: Accompanying a Standardization Process”. In: *ACM Transactions on Privacy and Security* 28.1 (Nov. 2024). DOI: [10.1145/3699716](https://doi.org/10.1145/3699716). URL: <https://doi.org/10.1145/3699716>.
- [15] P. Hosseyni, R. Küsters, and T. Würtele. “Formal Security Analysis of the OpenID FAPI 2.0: Accompanying a Standardization Process”. In: *37th IEEE Computer Security Foundations Symposium (CSF 2024)*. 2024, pp. 589–604. DOI: [10.1109/CSF61375.2024.00002](https://doi.org/10.1109/CSF61375.2024.00002).
- [16] G. Schmitz. “Privacy-Preserving Web Single Sign-On: Formal Security Analysis and Design”. PhD thesis. 2019. URL: <https://publ.sec.uni-stuttgart.de/schmitz-phdthesis-2019.pdf>.

A. OpenID for Verifiable Presentations Web System

The following algorithms model the OpenID for Verifiable Presentations protocol over the Digital Credentials browser API. The model is based on the master’s thesis “OpenID for Verifiable Credentials: Formal Security Analysis using the Web Infrastructure Model” [13].

The OID4VP protocol is modeled in what we call a *Verifiable Presentations Web System with Network Attacker* $\mathcal{V}PWS^n = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$. The system $\mathcal{W} = \text{Hon} \cup \text{Net}$ consists of a network attacker process in **Net** as well as the set $\text{Hon} := \text{Wallets} \cup \text{Verifiers} \cup \text{Issuers} \cup \text{B}$, which contains a finite set of a finite sets of processes for wallets (**Wallets**), verifiers (**Verifiers**), issuers (**Issuers**), and browsers (**B**). These processes are described in more detail in the following sections.

Due to the network attacker in **Net**, $\mathcal{V}PWS^n$ does not contain explicit DNS servers – due to the non-confidential, non-authenticated nature of standard DNS, we just assume that the attacker subsumes all DNS services.

The set of scripts \mathcal{S} and the mapping **script** are shown in Table 1 (see also Appendix C.11).

The set E^0 of initial events is defined as in Definition 60, i.e., containing infinitely many **TRIGGER** events for each IP address.

$s \in \mathcal{S}$	script (s)	Defined in
<i>script_verifier_authentication</i>	script_verifier_authentication	Algorithm 3
<i>script_verifier_index</i>	script_verifier_index	Algorithm 4

Table 1: List of scripts in \mathcal{S} with their string representation and definitions.

A.1. Initialization

This section outlines the initialization process for the Verifiable Presentations Web System with Network Attacker $\mathcal{V}PWS^n$.

A.1.1. Preliminary Definitions

Definition 1 (GETURL). Let **GETURL**(*tree*, *docnonce*) be defined as in Appendix B.1.1 of [5].

Definition 2 (Nearby Predicate). Let **nearby** : $\mathcal{W} \times \mathcal{W} \rightarrow \{\perp, \top\}$ be a reflexive and symmetric predicate that determines whether two processes are nearby.

Definition 3 (Wallets of ID). Let **walletsOfId** : $\text{ID} \rightarrow 2^{\text{Wallets}}$ map every id to a set of wallets.

A.1.2. Addresses and Domain Names

The set **IPs** contains a finite set of addresses for the network attacker in **Net**, every verifier in **Verifiers**, every wallet in **Wallets**, and every browser in **B** such that all these sets are disjoint. By **addr** : $\text{Net} \cup \text{Verifiers} \cup \text{Wallets} \cup \text{B} \rightarrow 2^{\text{IPs}}$ we denote the corresponding assignment from a process to a set of addresses.

The set **Doms** contains a finite set of domains for every verifier in **Verifiers**, every wallet in **Wallets**, every issuer in **Issuers**, and the network attacker in **Net**. The browsers in **B** do not have a domain. By **dom** : $\text{Net} \cup \text{Verifiers} \cup \text{Wallets} \cup \text{Issuers} \rightarrow 2^{\text{Doms}}$, we denote the corresponding assignment from a process to a set of domains (such that these sets are pairwise disjoint).

A.1.3. Identities

Identities consist, similar to email addresses, of a user name and a domain part. For our model, this is defined as follows:

Definition 4 (Identity). An identity id is a term of the form $\langle name, domain \rangle$ with $name \in \mathbb{S}$ and $domain \in \text{Doms}$. Let ID be the finite set of such identities. We say that an id is *governed* by the Dolev-Yao (DY) process to which the domain of the id belongs. This is formally captured by the mappings $\text{governor} : \text{ID} \rightarrow \mathcal{W}, \langle name, domain \rangle \mapsto \text{dom}^{-1}(domain)$ and $\text{ID}^y := \text{governor}^{-1}(y)$.

A.1.4. Keys and Secrets

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

$$\mathcal{K} = N \uplus K_{\text{TLS}} \uplus K_{\text{sign}} \uplus K_{\text{dcapi}}$$

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 $\text{tlskey} : \text{Doms} \rightarrow K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p , we define $\text{tlskeys}^p = \langle \{ \langle d, \text{tlskey}(d) \rangle \mid d \in \text{dom}(p) \} \rangle$ (i.e., a sequence of pairs).
- The set K_{sign} contains the keys that will be used by issuers to sign credentials, verifiers to sign authentication requests, and by wallets to sign presentations. Let $\text{signkey} : \text{Verifiers} \cup \text{Wallets} \cup \text{Issuers} \rightarrow K_{\text{sign}}$ be an injective mapping that assigns a (different) private signing key to every verifier, wallet, and issuer.
- The set K_{dcapi} contains symmetric keys that are shared between nearby processes to exchange encrypted DC API messages.
- Let $\text{dcApiKeys} : \mathcal{W} \times \mathcal{W} \rightarrow 2^{K_{\text{dcapi}}}$ be a mapping that (symmetrically) assigns disjoint sets of keys to pairs of nearby processes. I.e., $\forall p_1, p_2, p_3, p_4 \in \mathcal{W}$ we have all of the following:
 - Symmetry: $\text{dcApiKeys}(p_1, p_2) = \text{dcApiKeys}(p_2, p_1)$
 - Only nearby processes share keys: $\neg \text{nearby}(p_1, p_2) \Rightarrow \text{dcApiKeys}(p_1, p_2) = \emptyset$
 - Disjoint sets: $\text{dcApiKeys}(p_1, p_2) \cap \text{dcApiKeys}(p_3, p_4) \neq \emptyset \Rightarrow \{p_1, p_2\} = \{p_3, p_4\}$

A.2. Issuers

An issuer $i \in \text{Issuers}$ is an instance of the WIM's generic HTTPS web server (see [Appendix C.12](#)), modeled as an atomic DY process (I^i, Z^i, R^i, s_0^i) with the address $I^i := \text{addr}(i)$. The following definition defines the states Z^i of i and the initial state s_0^i of i .

Definition 5. A state $s \in Z^i$ of an issuer i is a term of the form

$$\langle \text{pendingDNS}, \text{pendingRequests}, \text{DNSAddress}, \text{keyMapping}, \text{tlskeys}, \text{corrupt}, \text{signingKey} \rangle$$

with pendingDNS , pendingRequests , DNSAddress , keyMapping , tlskeys , corrupt as in [Definition 61](#), and $\text{signingKey} \in \mathcal{K}$. An initial state s_0^i of i is a state of i with

- $s_0^i.\text{pendingDNS} = \langle \rangle$,

- $s_0^i.\text{pendingRequests} = \langle \rangle$,
- $s_0^i.\text{DNSAddress} \in \text{IPs}$,
- $s_0^i.\text{keyMapping} = \langle \{ \langle d, \text{pub}(\text{tlskey}(d)) \rangle \mid d \in \text{Doms} \} \rangle$,
- $s_0^i.\text{tlskeys} = \text{tlskeys}^i$,
- $s_0^i.\text{corrupt} = \perp$, and
- $s_0^i.\text{signingKey} = \text{signkey}(i)$.

Note that the **Issuers** do not participate in the protocol run and do therefore not override any of the default methods of the generic HTTPS server (see also [Section 2.2](#)).

A.3. Verifiers

An verifier $v \in \text{Verifiers}$ is an instance of the WIM's generic HTTPS web server (see [Appendix C.12](#)), modeled as an atomic DY process (I^v, Z^v, R^v, s_0^v) with the address $I^v := \text{addr}(v)$. The following definition defines the states Z^v of v and the initial state s_0^v of v .

Definition 6. A state $s \in Z^v$ of a verifier v is a term of the form

$$\langle \text{pendingDNS}, \text{pendingRequests}, \text{DNSAddress}, \text{keyMapping}, \text{tlskeys}, \text{corrupt}, \\ \text{issuers}, \text{sessions}, \text{signingKey}, \text{origins}, \text{nonces} \rangle$$

with pendingDNS , pendingRequests , DNSAddress , keyMapping , tlskeys , corrupt as defined in [Definition 61](#), $\text{issuers} \in [\text{Doms} \times \text{pub}(\mathcal{N})]$, $\text{sessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{signingKey} \in \mathcal{N}$, $\text{origins} \in \mathcal{T}_{\mathcal{N}}$, and $\text{nonces} \in \mathcal{T}_{\mathcal{N}}$.

An initial state s_0^v of v is a state of v with

- $s_0^v.\text{pendingDNS} = \langle \rangle$
- $s_0^v.\text{pendingRequests} = \langle \rangle$
- $s_0^v.\text{DNSAddress} \in \text{IPs}$
- $s_0^v.\text{keyMapping} = \langle \{ \langle d, \text{pub}(\text{tlskey}(d)) \rangle \mid d \in \text{Doms} \} \rangle$
- $s_0^v.\text{tlskeys} = \text{tlskeys}^v$
- $s_0^v.\text{corrupt} = \perp$
- $s_0^v.\text{issuers} = \langle \{ \langle d_i, \text{pub}(\text{signkey}(i)) \rangle \mid d_i \in \text{dom}(i) \wedge i \in \text{Issuers} \} \rangle$
- $s_0^v.\text{sessions} = \langle \rangle$
- $s_0^v.\text{signingKey} = \text{signkey}(v)$
- $s_0^v.\text{origins} = \langle \{ \langle d_v, \mathcal{S} \rangle \mid d_v \in \text{dom}(v) \} \rangle$
- $s_0^v.\text{nonces} = \langle \rangle$

The relation of a verifier v is based on the generic HTTPS server as defined in [Appendix C.12](#). The following algorithms 1 to 4 overwrite and extend the generic HTTPS server. Methods that are not overwritten are defined as in [Appendix C.12](#).

Algorithm 1 Relation of a verifier R^v : 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 address of the message.  $s'$  is the current state of the atomic DY process  $v$ .
1: function PROCESS_HTTPS_REQUEST( $m, k, a, f, s'$ )
2:   if  $m.path \equiv /$  then → Verifier's home page (see Algorithm 4)
3:     let  $m' := \text{enc}_s(\langle \text{HTTPResp}, m.\text{nonce}, 200, \langle \rangle, \langle \text{script\_verifier\_index} \rangle \rangle, k)$ 
4:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 


---


    → Generate authorization request


---


5:   else if  $m.path \equiv /start$  then
6:     let  $\text{nonce} := \nu_{\text{nonce}}$  → Randomness for request, see \[OID4VP, Section 14.1\]
7:     let  $s'.\text{nonces} := s'.\text{nonces} + \langle \rangle \text{ nonce}$  → Not bound to browser session, see Section 2.6
8:     let  $\text{parameters} := [\text{response\_type}: \text{vp\_token}]$  → See Section 2.7
9:     let  $\text{parameters}[\text{nonce}] := \text{nonce}$ 
10:    let  $\text{parameters}[\text{response\_mode}] := \text{dc\_api}$  → See Section 2.7
11:    let  $\text{signedRequest} \leftarrow \{\top, \perp\}$  → See Section 2.6
12:    if  $\text{signedRequest} \equiv \top$  then → Create a signed request
13:      let  $\text{protocol} := \text{openid4vp-v1-signed}$ 
14:      let  $\text{parameters}[\text{client\_id}] := m.\text{host}$ 
15:      let  $\text{parameters}[\text{expected\_origins}] := s'.\text{origins}$ 
16:      let  $\text{reqPars} := [\text{request}: \text{sig}(\text{parameters}, s'.\text{signingKey})]$ 
17:    else → Create an unsigned request
18:      let  $\text{protocol} := \text{openid4vp-v1-unsigned}$ 
19:      let  $\text{reqPars} := \text{parameters}$ 
    → Reply with a script that hands the request to the browser's DC API and forwards the resulting
    response to "our" /dc-api-response endpoint (see Algorithm 3 for the script):
20:    let  $\text{scriptState} := \langle \text{protocol}, \text{reqPars} \rangle$ 
21:    let  $m' := \text{enc}_s(\langle \text{HTTPResp}, m.\text{nonce}, 200, \langle \rangle,$ 
    →  $\langle \text{script\_verifier\_authentication}, \text{scriptState} \rangle \rangle, k)$ 
22:    stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 


---


    → Receive authorization response


---


23:   else if  $m.path \equiv /dc-api-response \wedge m.\text{method} \equiv \text{POST}$  then
24:     let  $\text{dcApiResponse} := m.\text{body}$ 
25:     let  $\text{presentation} := \text{dcApiResponse}[\text{vp\_token}]$ 
26:     let  $\text{retVal} := \text{VERIFY\_PRESENTATION}(m, s', \text{presentation})$  → See Algorithm 2.
    → Since VERIFY\_PRESENTATION did not stop,  $v$  logs in the browser that sent the presentation
    (by setting a cookie and associating that cookie with the presentation's claims, i.e.,  $id$ ).
27:     let  $\langle id, iss \rangle, s'$  such that  $\langle \langle id, iss \rangle, s' \rangle \equiv \text{retVal}$  if possible; otherwise stop
28:     let  $\text{serviceTokenID} := \nu_{\text{serviceTokenID}}$ 
29:     let  $s'.\text{sessions}[\text{serviceTokenID}] := \langle id, iss \rangle$ 
30:     let  $\text{url} := \langle \text{URL}, \text{S}, m.\text{host}, /, \langle \rangle, \perp \rangle$  → Send user back to verifier home page.
31:     let  $\text{headers} := \langle \langle \text{Location}, \text{url} \rangle \rangle$ 
32:     let  $\text{headers} := \text{headers} + \langle \rangle \langle \text{Set-Cookie}, [\text{serviceTokenID}: \langle \text{serviceTokenID}, \top, \top, \top \rangle] \rangle$ 
33:     let  $m' := \text{enc}_s(\langle \text{HTTPResp}, m.\text{nonce}, 303, \text{headers}, \langle \rangle \rangle, k)$ 
34:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
35:   stop → Invalid path.

```

Algorithm 2 Relation of a verifier R^v : Function to verify a presentation.

```
1: function VERIFY_PRESENTATION( $m, s', presentation$ )
2:   let  $credential, nonce, aud$  such that
    $\hookrightarrow \langle credential, nonce, aud \rangle \equiv \text{extractmsg}(presentation)$ 
    $\hookrightarrow$  if possible; otherwise stop  $\rightarrow$  See Section 2.3.
3:   let  $origin$  such that  $\langle origin, origin \rangle \equiv aud$  if possible; otherwise stop
4:   if  $origin \notin s'.origins$  then
5:     stop  $\rightarrow$  Check aud claim, see [OID4VP, Section 14.1].
6:   if  $nonce \notin s'.nonces$  then
7:     stop  $\rightarrow$  Check nonce, see [OID4VP, Section 14.1].
8:   let  $s'.nonces := s'.nonces - nonce$   $\rightarrow$  Nonce can only be used once, see [OID4VP, Section 14.1].
    $\rightarrow$  See Section 2.1 for a description of our model's credential format.
9:   let  $id, pubKey$  such that  $\langle id, pubKey \rangle \equiv \text{extractmsg}(credential)$  if possible; otherwise stop
10:  let  $name, iss$  such that  $\langle name, iss \rangle \equiv id$  if possible; otherwise stop
11:  if  $\text{checksig}(presentation, pubKey) \neq \top$  then
12:    stop  $\rightarrow$  Check proof of possession, see [OID4VP, Section 14.1].
13:  if  $\text{checksig}(credential, s'.issuers[iss]) \neq \top$  then
14:    stop  $\rightarrow$  Check credential validity, see Section 2.2.
    $\rightarrow$  If this function returns anything, then all checks passed, i.e., the presentation is accepted by  $v$ .
15:  return  $\langle \langle id, iss \rangle, s' \rangle$ 
```

Algorithm 3 Relation of $script_verifier_authentication$.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets \rangle$
 \rightarrow DC API response from browser to script, POST that response to the script's origin.

```
1: if  $\exists response$  such that  $\langle DC\_API\_RESPONSE, response \rangle \equiv scriptstate$  then
2:   let  $url := \text{GETURL}(tree, docnonce)$ 
3:   let  $url' := \langle URL, S, url.host, /dc-api-response, \langle \rangle, \perp \rangle$ 
4:   let  $command := \langle FORM, url', POST, response, \perp \rangle$ 
    $\rightarrow$  Hand a DC API request to the browser's API.
5: else
6:   let  $command := \langle DCAPI, scriptstate.1, scriptstate.2 \rangle$ 
7: stop  $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$ 
```

Algorithm 4 Relation of $script_verifier_index$.

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

```
1: let  $switch \leftarrow \{oid4vp-flow, link\}$ 
2: if  $switch \equiv oid4vp-flow$  then  $\rightarrow$  Start a new OID4VP flow.
3:   let  $url := \text{GETURL}(tree, docnonce)$ 
4:   let  $url' := \langle URL, S, url.host, /start, \langle \rangle, \perp \rangle$ 
5:   let  $command := \langle HREF, url', \perp, \perp \rangle$ 
6:   stop  $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$ 
7: else  $\rightarrow$  Follow a random link.
8:   let  $protocol \leftarrow \{P, S\}$ 
9:   let  $host \leftarrow \text{Doms}$ 
10:  let  $path \leftarrow S$ 
11:  let  $parameters \leftarrow [S \times S]$ 
12:  let  $fragment \leftarrow S$ 
13:  let  $url := \langle URL, protocol, host, path, parameters, fragment \rangle$ 
14:  let  $command := \langle HREF, url, \perp, \perp \rangle$ 
15:  stop  $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$ 
```

A.4. Wallets

A wallet $w \in \text{Wallets}$ is an instance of the WIM's generic HTTPS web server (see [Appendix C.12](#)), modeled as an atomic DY process (I^w, Z^w, R^w, s_0^w) with the address $I^w := \text{addr}(w)$. The following definition defines the states Z^w of w and the initial state s_0^w of w .

Definition 7. A state $s \in Z^w$ of a wallet w is a term of the form

$$\langle \text{pendingDNS}, \text{pendingRequests}, \text{DNSAddress}, \text{keyMapping}, \text{tlskeys}, \\ \text{corrupt}, \text{credentials}, \text{holderKey}, \text{dcApiKeys}, \text{currentDcApiKeys}, \\ \text{dcApiReqsToWallet}, \text{dcApiRespsFromWallet}, \text{trustedVerifiers} \rangle$$

with pendingDNS , pendingRequests , DNSAddress , keyMapping , tlskeys , corrupt as described in [Definition 61](#), $\text{credentials} \in \mathcal{T}_{\mathcal{N}}$, $\text{holderKey} \in \mathcal{N}$, $\text{dcApiKeys} \in \mathcal{T}_{\mathcal{N}}$, $\text{currentDcApiKeys} \in \mathcal{T}_{\mathcal{N}}$, $\text{dcApiReqsToWallet} \in \mathcal{T}_{\mathcal{N}}$, $\text{dcApiRespsFromWallet} \in \mathcal{T}_{\mathcal{N}}$, $\text{trustedVerifiers} \in \mathcal{T}_{\mathcal{N}}$.

An initial state s_0^w of w is a state of w with

- $s_0^w.\text{pendingDNS} = \langle \rangle$
- $s_0^w.\text{pendingRequests} = \langle \rangle$
- $s_0^w.\text{DNSAddress} \in \text{IPs}$
- $s_0^w.\text{keyMapping} = \langle \{ \langle d, \text{pub}(\text{tlskey}(d)) \rangle \mid d \in \text{Doms} \} \rangle$
- $s_0^w.\text{tlskeys} = \text{tlskeys}^w$
- $s_0^w.\text{corrupt} = \perp$
- $s_0^w.\text{credentials} = \langle \{ \text{credentialOfldForWallet}(w, id) \mid id \in \text{ID} \wedge w \in \text{walletsOfld}(id) \} \rangle$ where $\text{credentialOfldForWallet}(w, id) = \text{sig}(\langle id, \text{pub}(s_0^w.\text{holderKey}) \rangle, \text{signkey}(\text{governor}(id)))$
- $s_0^w.\text{holderKey} = \text{signkey}(w)$
- $s_0^w.\text{dcApiKeys} = \langle \bigcup_{p \in \mathcal{W}} \text{dcApiKeys}(w, p) \rangle$
- $s_0^w.\text{currentDcApiKeys} = \langle \rangle$
- $s_0^w.\text{dcApiReqsToWallet} = \langle \rangle$
- $s_0^w.\text{dcApiRespsFromWallet} = \langle \rangle$
- $s_0^w.\text{trustedVerifiers} = \langle \{ \langle d, \text{pub}(\text{signkey}(v)) \rangle \mid d \in \text{dom}(v) \wedge v \in \text{Verifiers} \} \rangle$

The relation of a wallet w is based on the generic HTTPS server as defined in [Appendix C.12](#). The following algorithms 5 to 6 overwrite and extend the generic HTTPS server. Methods that are not overwritten are defined as in [Appendix C.12](#).

Algorithm 5 Relation of a wallet R^w : Processing non-HTTPS requests.

→ **Process an incoming non-HTTPS, non-TRIGGER message.** m is the incoming message, a is the receiver, f the sender address of the message, and s' is the current state of the atomic DY process w .

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

2: **if** \exists $request, origin, k$ **such that**
 $\hookrightarrow \langle DC_API_REQUEST, protocol, request, origin \rangle \equiv \text{dec}_s(m, k) \wedge \rightarrow m$ is a DC API request encrypted
 $\hookrightarrow k \in s'.dcApiKeys$ **then** $\rightarrow \dots$ using a key of w (see Section 2.4.2).
 3: **let** $s'.dcApiKeys := s'.dcApiKeys - \langle \rangle^* k$ \rightarrow Remove k because each key is only used once.
 4: **let** $dcApiSess := \nu_{dcApiSess}$ \rightarrow Used to map responses to requests.
 5: **let** $s'.currentDcApiKeys := s'.currentDcApiKeys + \langle \rangle \langle dcApiSess, k, f \rangle$
 6: **let** $selectHonestWallet \leftarrow \{\top, \perp\}$ \rightarrow See Section 2.4.4.
 7: **let** $requestToWallet := \langle protocol, request, origin, dcApiSess \rangle$
 8: **if** $selectHonestWallet$ **then** \rightarrow User selected an honest wallet.
 9: **let** $s'.dcApiReqsToWallet := s'.dcApiReqsToWallet + \langle \rangle requestToWallet$
 10: **stop** $\langle \rangle, s'$
 11: **else** \rightarrow User selected an attacker wallet.
 12: **stop** $\langle \langle f, a, requestToWallet \rangle \rangle, s'$

13: **else if** \exists $response, dcApiSess, k, f'$ **such that**
 $\hookrightarrow \langle response, dcApiSess \rangle \equiv m \wedge \langle dcApiSess, k, f' \rangle \in \langle \rangle s'.currentDcApiKeys$ **then**
 \rightarrow Remove entry to ensure that only one response can be send back over the DC API:
 14: **let** $s'.currentDcApiKeys := s'.currentDcApiKeys - \langle \rangle \langle dcApiSess, k, f' \rangle$
 15: **let** $m' := \text{enc}_s(\langle DC_API_RESPONSE, response \rangle, k)$
 16: **stop** $\langle \langle f', a, m' \rangle \rangle, s'$

17: **else if** $m \sim [\text{CRED}: *]$ **then**
 18: **let** $s'.credentials := s'.credentials + \langle \rangle m[\text{CRED}]$
 19: **stop** $\langle \rangle, s'$

20: **stop** \rightarrow Invalid message

Algorithm 6 Relation of a wallet R^w : Processing triggers.

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

```

1: function PROCESS_TRIGGER( $a, s'$ )
2:   let  $action \leftarrow \{\text{PROCESS\_DC\_API\_REQUEST}, \text{SEND\_DC\_API\_RESPONSE}\}$ 
   —Invoke wallet app—


---


3:   if  $action \equiv \text{PROCESS\_DC\_API\_REQUEST}$  then
4:     let  $\langle protocol, request, origin, dcApiSess \rangle \leftarrow s'.dcApiReqsToWallet$  → See Line 9 of Algorithm 5
     → Remove request from sequence so that every request is only processed once:
5:     let  $s'.dcApiReqsToWallet := s'.dcApiReqsToWallet - \langle protocol, request, origin, dcApiSess \rangle$ 
6:     if  $protocol \equiv \text{openid4vp-v1-signed}$  then → Signed authorization request
7:       let  $sig := request[request]$ 
8:       let  $authReq := \text{extractmsg}(sig)$ 
9:       let  $verifier := authReq[client\_id]$ 
10:      let  $pubKey := s'.trustedVerifiers[verifier]$ 
11:      if  $\text{checksig}(sig, pubKey) \neq \top$  then
12:        stop
13:      let  $nonce := authReq[nonce]$ 
14:    else → Unsigned authorization request
15:      let  $nonce := request[nonce]$ 
16:      let  $credential \leftarrow s'.credentials$ 
17:      let  $aud := \langle origin, origin \rangle$ 
18:      let  $presentation := \langle credential, nonce, aud \rangle$ 
19:      let  $vpToken := \text{sig}(presentation, s'.holderKey)$ 
20:      let  $response := [\text{vp\_token}: vpToken]$ 
21:      let  $s'.dcApiRespsFromWallet := s'.dcApiRespsFromWallet + \langle response, dcApiSess \rangle$ 
22:      stop  $\langle \rangle, s'$ 
   —Forward response from wallet app via DC API—


---


23:   else if  $action \equiv \text{SEND\_DC\_API\_RESPONSE}$  then
24:     let  $\langle response, dcApiSess \rangle \leftarrow s'.dcApiRespsFromWallet$ 
25:     let  $k, f$  such that  $\langle dcApiSess, k, f \rangle \in s'.currentDcApiKeys$  if possible; otherwise stop
     → Remove entry to ensure that only one response can be send back over the DC API:
26:     let  $s'.currentDcApiKeys := s'.currentDcApiKeys - \langle dcApiSess, k, f \rangle$ 
27:     let  $m' := \text{enc}_s(\langle \text{DC\_API\_RESPONSE}, response \rangle, k)$ 
28:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 

```

A.5. Web Browser

Web browser processes (i.e., processes $b \in \mathbf{B}$) are modeled as described in [Appendix C.7](#) with the initial state extension defined in [Definition 8](#) and two modifications of the browser algorithms specified in [Algorithm 7](#) and [Algorithm 8](#) to accommodate the DC API.

Definition 8 (Web Browser State Extension). A state $s \in Z_{\text{webbrowser}}$ of a browser b defined in [Definition 50](#) is extended by the following sub term in this work:

$$dcApiKeys, dcApiSessions$$

with $dcApiKeys \in \mathcal{T}_{\mathcal{N}}$, and $dcApiSessions \in \mathcal{T}_{\mathcal{N}}$.

The initial state s_0^b of a web browser b follows the description in [Definition 50](#) with the following additional constraints:

- $s_0^b.\text{keyMapping} = \langle \{ \langle d, \text{pub}(\text{tlskey}(d)) \rangle \mid d \in \text{Doms} \} \rangle$
- $s_0^b.\text{dcApiKeys} = \langle \bigcup_{p \in \mathcal{W}} \text{dcApiKeys}(b, p) \rangle$
- $s_0^b.\text{dcApiSessions} = \langle \rangle$

Algorithm 7 Web Browser Model: Execute a script (extends [Algorithm 15](#)).

```

1: function RUNSCRIPT( $\overline{w}, \overline{d}, a, s'$ )
    :
17:   let docorigin :=  $s'.\overline{d}.\text{origin}$ 
18:   switch command do
19:     case  $\langle \text{DCAPI}, \text{protocol}, \text{request} \rangle \rightarrow$  Process a DC API request from a script (e.g., Algorithm 3)
20:       let  $k \leftarrow s'.\text{dcApiKeys}$   $\rightarrow$  Choose key for DC API request non-deterministically
21:       let  $s'.\text{dcApiKeys} := s'.\text{dcApiKeys} - \langle \rangle k$   $\rightarrow$  Key is only used once, see Section 2.4.2
22:       let  $s'.\text{dcApiSessions} := s'.\text{dcApiSessions} + \langle \rangle \langle k, \overline{d} \rangle$ 
23:       let  $dcApiRequest := \langle \text{DC\_API\_REQUEST}, \text{protocol}, \text{request}, \text{docorigin} \rangle$ 
24:       let  $m' := \text{enc}_s(dcApiRequest, k)$ 
25:       let  $f \leftarrow \text{IPs}$ 
26:       stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
27:     case  $\langle \text{HREF}, \text{url}, \text{hrefwindow}, \text{noreferrer} \rangle$ 
    :

```

Algorithm 8 Web Browser Model: Main Algorithm (extends Algorithm 17).

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

- 1: **let** $s' := s$
- 2: **if** $s.\text{isCorrupted} \neq \perp$ **then**
- 3: **let** $s'.\text{pendingRequests} := \langle m, s.\text{pendingRequests} \rangle \rightarrow$ Collect incoming messages
- 4: **let** $m' \leftarrow d_V(s')$
- 5: **let** $a' \leftarrow \text{IPs}$
- 6: **stop** $\langle \langle a', a, m' \rangle \rangle, s'$
- 7: **if** $m \equiv \text{TRIGGER}$ **then** \rightarrow A special trigger message.
- 8: **let** $\text{switch} \leftarrow \{\text{script}, \text{urlbar}, \text{reload}, \text{forward}, \text{back}\}$
- \vdots
- 73: **let** $s'.\text{pendingDNS} := s'.\text{pendingDNS} - m.\text{nonce}$
- 74: **stop** $\langle \langle m.\text{result}, a, \text{message} \rangle \rangle, s'$
- Hand an incoming DC API response to the script that emitted the corresponding request———
- 75: **else if** $\exists k, \bar{d}, \text{response}$ **such that**
 $\hookrightarrow \langle k, \bar{d} \rangle \in s'.\text{dcApiSessions} \wedge \langle \text{DC_API_RESPONSE}, \text{response} \rangle \equiv \text{dec}_s(m, k)$ **then**
- 76: **let** $s'.\bar{d}.\text{scriptstate} := \langle \text{DC_API_RESPONSE}, \text{response} \rangle \rightarrow$ Pass response to script
- 77: **let** $s'.\text{dcApiSessions} := s'.\text{dcApiSessions} - \langle k, \bar{d} \rangle \rightarrow$ Ensure only one response can be received
- 78: **stop** $\langle \rangle, s'$
- 79: **stop**

B. Formal Security Properties

Before defining our formal security properties, we introduce a few helper definitions.

Definition 9 (Verifier Logged In). Let $P = (S, E, N) \xrightarrow[v \rightarrow E_{\text{out}}]{e_{\text{in}} \rightarrow v} (S', E', N')$ be a processing step in a run ρ of a Verifiable Presentations Web System with Network Attacker \mathcal{VPWS}^n with $v \in \text{Verifiers}$, $id \in \text{ID}$, and $sTID \in \mathcal{T}_{\mathcal{N}}$. We say that v *logged in* id in processing step P of ρ with service token ID $sTID$, if all of the following are true:

- (i) $\exists e_{\text{sch}} \in {}^{\langle \rangle} E_{\text{out}}, m_{\text{sch}}, k \in \mathcal{T}_{\mathcal{N}} : e_{\text{sch}} \sim \langle *, *, \text{enc}_s(m_{\text{sch}}, k) \rangle$ such that $m_{\text{sch}} \in \text{HTTPResponses}$
- (ii) $\exists i \in \mathbb{N} : m_{\text{sch}}.\text{headers}.i \equiv \langle \text{Set-Cookie}, [\text{serviceTokenID}: \langle sTID, \top, \top, \top \rangle] \rangle$
- (iii) $S'(v).\text{sessions}[sTID] \equiv \langle id, id.\text{domain} \rangle$

We then write $\text{loggedIn}_{\rho}^P(v, id, sTID)$.

Definition 10 (Verifier Accepts Presentation). Let $P = (S, E, N) \xrightarrow[v \rightarrow E_{\text{out}}]{e_{\text{in}} \rightarrow v} (S', E', N')$ be a processing step in a run ρ of a Verifiable Presentations Web System with Network Attacker \mathcal{VPWS}^n with $v \in \text{Verifiers}$, $id \in \text{ID}$, and $sTID, pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$. We say that v *accepts a presentation* $pres$ signed with k_{holder} for id in processing step P of ρ with service token ID $sTID$, if all of the following are true:

- (i) $\exists e_{\text{sch}} \in {}^{\langle \rangle} E_{\text{out}}, m_{\text{sch}}, k \in \mathcal{T}_{\mathcal{N}} : e_{\text{sch}} \sim \langle *, *, \text{enc}_s(m_{\text{sch}}, k) \rangle$ such that $m_{\text{sch}} \in \text{HTTPResponses}$
- (ii) $\exists i \in \mathbb{N} : m_{\text{sch}}.\text{headers}.i \equiv \langle \text{Set-Cookie}, [\text{serviceTokenID}: \langle sTID, \top, \top, \top \rangle] \rangle$
- (iii) $S'(v).\text{sessions}[sTID] \equiv \langle id, id.\text{domain} \rangle$
- (iv) $e_{\text{in}} \sim \langle *, *, \text{enc}_a(\langle \langle \text{HTTPReq}, *, *, *, *, *, [\text{vp_token}: pres] \rangle, k \rangle, \text{pub}(*)) \rangle$
- (v) $pres \sim \text{sig}(\langle \text{sig}(\langle id, \text{pub}(k_{\text{holder}}) \rangle, *) \rangle, *, *, k_{\text{holder}})$

We then write $\text{acceptsPresentation}_\rho^P(v, id, sTID, pres, k_{\text{holder}})$.

Lemma 1 (loggedIn implies acceptsPresentation). Let $P = (S, E, N) \xrightarrow[v \rightarrow E_{\text{out}}]{e_{\text{in}} \rightarrow v} (S', E', N')$ be a processing step in a run ρ of a Verifiable Presentations Web System with Network Attacker \mathcal{VPWS}^n with $v \in \text{Verifiers}$, $id \in \text{ID}$, and $sTID \in \mathcal{T}_{\mathcal{N}}$, such that $\text{loggedIn}_\rho^P(v, id, sTID)$. If v is honest in S' , then $\exists pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$ such that $\text{acceptsPresentation}_\rho^P(v, id, sTID, pres, k_{\text{holder}})$.

PROOF. Statements (i), (ii), and (iii) in Definition 10 are identical to the corresponding statements in Definition 9, i.e., follow immediately from $\text{loggedIn}_\rho^P(v, id, sTID)$.

(A) **Statement (iv).**

- (A.i) Due to (i) and (ii), v must emit an HTTPS response in P , and that response must contain a Set-Cookie header with a cookie with name `serviceTokenID`. The only place in which a verifier such as v outputs such an event is in Line 34 of Algorithm 1, after creating the corresponding header in Line 31 of Algorithm 1. To even reach this line, Algorithm 1 must be called, which only ever happens in Line 9 of Algorithm 27. Reaching that line implies that all conditions in Lines 7f. of Algorithm 27 were met, and in particular, $e_{\text{in}} \sim \langle *, *, \text{enc}_a(\langle \langle \text{HTTPReq}, *, *, *, *, *, *, body \rangle, k_{\text{req}} \rangle, \text{pub}(*)) \rangle$ for some $body, k_{\text{req}} \in \mathcal{T}_{\mathcal{N}}$.
- (A.ii) Again to reach Line 31 of Algorithm 1, the execution of `VERIFY_PRESENTATION` (Algorithm 2) during P as called in Line 26 of Algorithm 1 must reach the end of Algorithm 2. In particular, none of the **stops** in that algorithm are reached during P . Hence, due to Lines 24f. of Algorithm 1 and Line 2 of Algorithm 2, the body of the message processed in P , i.e., $body$, must match the pattern `[vp_token: *]`.
- (A.iii) Furthermore, the key k used to encrypt the response emitted in P (see (i)) is applied in Line 33 of Algorithm 1 and the value used there originates from Line 9 of Algorithm 27, i.e., is the second part of e_{in} 's decrypted payload, in other words: $k \equiv k_{\text{req}}$.

Therefore, we have (iv).

(B) **Statement (v).** Recall (A.ii) from above: None of the **stops** in `VERIFY_PRESENTATION` can be reached during P . In the following, let $pres' := body[\text{vp_token}]$, i.e., $pres'$ is the third argument given to `VERIFY_PRESENTATION` in the call in Line 26 of Algorithm 1 during P .

- (B.i) Due to Line 25 of Algorithm 1, $pres' = pres$.
- (B.ii) Due to Line 2 of Algorithm 2 not ending in **stop**, we have $pres' \sim \text{sig}(\langle cred, *, * \rangle, k_{pres'})$ for some $cred, k_{pres'} \in \mathcal{T}_{\mathcal{N}}$ (see also Definition 17).
- (B.iii) Due to Line 9 of Algorithm 2 not ending in **stop**, we also have $cred \sim \text{sig}(\langle id', pubKey' \rangle, *)$ for some $id', pubKey' \in \mathcal{T}_{\mathcal{N}}$.
- (B.iv) Due to Line 11 of Algorithm 2 not leading to the **stop** in the next Line, and due to Definition 17, we get $pres' \sim \text{sig}(\langle \text{sig}(\langle id', \text{pub}(k_{pres'}) \rangle, *), *, * \rangle, k_{pres'})$, i.e., $\text{pub}(k_{pres'}) = pubKey'$ from above. Or, just named differently, $k_{\text{holder}} = k_{pres'}$.
- (B.v) Due to (iii) and the fact that the value for $sTID$ is always a fresh nonce (see Line 28 of Algorithm 1, and Line 29 of Algorithm 1 is the only place there a verifier such as v adds or changes elements in its `sessions` state subterm), the id stored in `sessions` must be the first element of the first element of the return value of `VERIFY_PRESENTATION` call during P . That element of the return value originates from Line 9 of Algorithm 2, i.e., we can infer $id = id'$.

Hence, we have (v).

Overall, this gives us $\text{acceptsPresentation}_\rho^P(v, id, sTID, pres, k_{\text{holder}})$ for some $pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$. ■

Definition 11 (Wallet Creates Presentation). Let $Q = (S, E, N) \xrightarrow[w \rightarrow E_{\text{out}}]{e_{\text{in}} \rightarrow w} (S', E', N')$ be a processing step in a run ρ of a Verifiable Presentations Web System with Network Attacker \mathcal{VPWS}^n with $w \in \text{Wallets}$, and $\text{pres} \in \mathcal{T}_{\mathcal{N}}$. We say that w creates presentation pres in processing step Q of ρ , if all of the following are true:

- (i) $e_{\text{in}} \sim \langle *, *, \text{TRIGGER} \rangle$
- (ii) During Q , w selects the `PROCESS_DC_API_REQUEST` action in Line 2 of Algorithm 6
- (iii) The term resulting from the signature operation in Line 19 of Algorithm 6 during Q is pres

We then write $\text{createsPresentation}_\rho^Q(w, \text{pres})$.

B.1. Verifier Authentication

See Section 4.1 for an informal description of this property.

Definition 12 (Verifier Authentication). We say that a Verifiable Presentations Web System with Network Attacker \mathcal{VPWS}^n is secure w.r.t. verifier authentication iff for every run ρ of \mathcal{VPWS}^n , every configuration (S^*, E^*, N^*) in ρ , every wallet $w \in \text{Wallets}$ that is honest in S^* , every processing step $P = (S, E, N) \xrightarrow[w \rightarrow E_{\text{out}}]{e_{\text{in}} \rightarrow w} (S', E', N')$ prior to (S^*, E^*, N^*) in ρ , if

- (1) during P , w received a trigger message, i.e. $e_{\text{in}} \sim \langle *, *, \text{TRIGGER} \rangle$,
- (2) selected the `PROCESS_DC_API_REQUEST` action in Line 2 of Algorithm 6,
- (3) chose a term $\langle \text{protocol}, \text{request}, \text{origin}, \text{dcApiSess} \rangle$ in Line 4 of Algorithm 6 such that $\text{protocol} \equiv \text{openid4vp-v1-signed}$, and
- (4) finished P with the stop in Line 22 of Algorithm 6,

then, with $\text{cid} := \text{extractmsg}(\text{request}[\text{request}])[\text{client_id}]$,

- (I) $\text{client_id} \in \text{extractmsg}(\text{request}[\text{request}])$, $\text{cid} \in \text{Doms}$, and
- (II) if $v := \text{dom}^{-1}(\text{cid})$ is honest in S , then there is a processing step Q prior to P in ρ in which v executed Line 16 of Algorithm 1, and the result of the signature operation there was request .

B.2. Wallet Authorization

See Section 4.2 for an informal description of this property.

Definition 13 (Wallet Authorization). We say that a Verifiable Presentations Web System with Network Attacker \mathcal{VPWS}^n is secure w.r.t. wallet authorization iff for every run ρ of \mathcal{VPWS}^n , every configuration (S^*, E^*, N^*) in ρ , every verifier $v \in \text{Verifiers}$ that is honest in S^* , every processing step $P = (S, E, N) \xrightarrow[v \rightarrow E_{\text{out}}]{e_{\text{in}} \rightarrow v} (S', E', N')$ prior to (S^*, E^*, N^*) in ρ , every identity $\text{id} \in \text{ID}$, all terms $sTID \in \mathcal{T}_{\mathcal{N}}$, if we have

- (1) $\text{loggedIn}_\rho^P(v, \text{id}, sTID)$ (which, by Lemma 1 and the fact that v honest in S^* implies v honest in S' , implies $\text{acceptsPresentation}_\rho^P(v, \text{id}, sTID, \text{pres}, k_{\text{holder}})$ for some $\text{pres}, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$), and
- (2) $\exists w \in \text{Wallets}$ that is honest in S^* , such that $k_{\text{holder}} \equiv s_0^w.\text{holderKey}$,

then there is a processing step Q in ρ prior to P , such that $\text{createsPresentation}_\rho^Q(w, \text{pres})$.

B.3. Claims Unforgeability

See [Section 4.3](#) for an informal description of this property.

Definition 14 (Claims Unforgeability). We say that a Verifiable Presentations Web System with Network Attacker \mathcal{VPWS}^n is secure w.r.t. *claims unforgeability* iff for every run ρ of \mathcal{VPWS}^n , every configuration (S^*, E^*, N^*) in ρ , every verifier $v \in \text{Verifiers}$ that is honest in S^* , every processing step $P = (S, E, N) \xrightarrow[v \rightarrow E_{\text{out}}]{e_{\text{in}} \rightarrow v} (S', E', N')$ prior to (S^*, E^*, N^*) in ρ , every identity $id \in \text{ID}$, if we have all of the following:

- (1) $\text{loggedIn}_\rho^P(v, id, sTID)$ for some $sTID \in \mathcal{T}_{\mathcal{N}}$ (which, by [Lemma 1](#) and the fact that v honest in S^* implies v honest in S' , implies $\text{acceptsPresentation}_\rho^P(v, id, sTID, pres, k_{\text{holder}})$ for some $pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$),
- (2) all $w' \in \text{walletsOfId}(id)$ are honest in S' ,
- (3) $iss := \text{dom}^{-1}(id.\text{domain})$ is honest in S' , and
- (4) for $w \in \text{Wallets}$ such that $k_{\text{holder}} \equiv s_0^w.\text{holderKey}$, or, if no such wallet process exists, $w = \text{Net}$, all processes nearby w are honest in S^* , i.e., all processes in $\{p \mid p \in \mathcal{W} \wedge \text{nearby}(p, w)\}$,

then the attacker cannot derive $sTID$ in S^* , i.e., $sTID \notin d_\emptyset(S^*(\text{attacker}))$.

C. Technical Definitions

Here, we provide technical definitions of the WIM. These follow the descriptions in [5, 6, 8–12].

C.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 15 (Signature). In the case of the WIM, the signature Σ consists of the following pairwise disjoint sets:

Constants $\mathcal{C} = \mathbb{S} \cup \text{IPs} \cup \{\perp, \top, \diamond\}$ 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: $\text{pub}(\cdot)$, $\text{enc}_a(\cdot, \cdot)$, $\text{dec}_a(\cdot, \cdot)$, $\text{enc}_s(\cdot, \cdot)$, $\text{dec}_s(\cdot, \cdot)$, $\text{sig}(\cdot, \cdot)$, $\text{checksig}(\cdot, \cdot)$, $\text{mac}(\cdot, \cdot)$, $\text{checkmac}(\cdot, \cdot)$, $\text{extractmsg}(\cdot)$, $\text{hash}(\cdot)$.

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

Projection Symbols to access sequence elements: $\pi_i(\cdot)$ for all $i \in \mathbb{N}_0$. Note that formally, projection symbols are also function symbols.

Definition 16 (Nonces and Terms). Given this signature, we define $X = \{x_1, x_2, \dots\}$ to be an infinite set of variables, and \mathcal{N} to be an infinite set of constants (*nonces*) such that Σ, 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 $t_1, \dots, t_n \in \mathcal{T}_N(X)$, then $f(t_1, \dots, t_n) \in \mathcal{T}_N(X)$.

Definition 17 (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:

$$\text{dec}_a(\text{enc}_a(x, \text{pub}(y)), y) = x \quad (1)$$

$$\text{dec}_s(\text{enc}_s(x, y), y) = x \quad (2)$$

$$\text{checksig}(\text{sig}(x, y), \text{pub}(y)) = \top \quad (3)$$

$$\text{extractmsg}(\text{sig}(x, y)) = x \quad (4)$$

$$\text{checkmac}(\text{mac}(x, y), y) = \top \quad (5)$$

$$\text{extractmsg}(\text{mac}(x, y)) = x \quad (6)$$

$$\pi_i(\langle x_1, \dots, x_n \rangle) = x_i \text{ if } 1 \leq i \leq n \quad (7)$$

$$\pi_j(\langle x_1, \dots, x_n \rangle) = \diamond \text{ if } j \notin \{1, \dots, n\} \quad (8)$$

$$\pi_j(t) = \diamond \text{ if } t \text{ is not a sequence} \quad (9)$$

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 $\pi_1(\text{dec}_a(\text{enc}_a(\langle \mathbf{a}, \mathbf{b} \rangle, \text{pub}(k)), k)) \equiv \mathbf{a}$.

Definition 18 (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 $\text{pub}(k)$ are messages, where k typically models a private key and $\text{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\text{enc}_a(\langle a, b, c \rangle, \text{pub}(k))$ is interpreted to be the message $\langle a, b, c \rangle$ (the sequence of constants a, b, c) encrypted by the public key $\text{pub}(k)$.

Definition 19 (Events and Protoevents). An *event* (over IPs and \mathcal{M}) is a term of the form $\langle a, f, m \rangle$, for $a, f \in \text{IPs}$ 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}^ν are called *protoevents* and are denoted \mathcal{E}^ν . By $2^{\mathcal{E}}$ (or $2^{\mathcal{E}^\nu}$, 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 20 (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 17. For a term t , we denote its normal form as $t \downarrow$.

Definition 21 (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 $t \sim patterns$ 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 $\langle \top, 42 \rangle \sim p$, $\langle \perp, 42 \rangle \not\sim p$, and

$$\langle \langle \perp, \top \rangle, \langle \top, 23 \rangle, \langle a, b \rangle, \langle \top, \perp \rangle \rangle | p = \langle \langle \top, 23 \rangle, \langle \top, \perp \rangle \rangle .$$

Definition 22 (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 23 (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:

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

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 $M = \{m_1, \dots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \dots, m_n \rangle$. The order of the elements does not matter; one is chosen arbitrarily.

Definition 24 (Dictionaries). A *dictionary over X and Y* is a term of the form

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

where $k_1, \dots, k_n \in X$, $v_1, \dots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \dots, n\}$, an *element* of the dictionary with key k_i and value v_i . We often write $[k_1 : v_1, \dots, k_n : v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$. 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 25 (Operations on Dictionaries). Let $z = [k_1 : v_1, k_2 : v_2, \dots, k_n : v_n]$ be a dictionary with unique keys, i.e., $\forall i, j : k_i \neq k_j$. In addition, let t and v be terms. We define the following operations:

- $t \in z \iff \exists i \in \{1, \dots, n\} : k_i = t$
- $z[t] := \begin{cases} v_i & \text{if } \exists k_i \in z : t = k_i \\ \langle \rangle & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- $z - t := \begin{cases} [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1}, \dots, k_n : v_n] & \text{if } \exists k_i \in z : t = k_i \\ z & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- 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 $\langle t, x \rangle \in^\diamond z$, 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 $t = \langle t_1, \dots, t_n \rangle$, 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 26 (Pointers). A *pointer* is a sequence of non-negative integers. We write $\tau.\bar{p}$ for the application of the pointer \bar{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 $\bar{p} = \langle 3, 1 \rangle$, the subterm of τ at the position \bar{p} is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3, 1 \rangle = \tau.3.\bar{p} = \tau.3.3.1 = e$.

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 $\langle host, protocol \rangle$ and o is an *Origin* term, then we can write $o.protocol$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

Note that this “replacement” of a string (the subterm name) with an integer (the pointer value) happens as early as possible.

Definition 27 (Concatenation of Sequences). For a sequence $a = \langle a_1, \dots, a_i \rangle$ and a sequence $b = \langle b_1, b_2, \dots \rangle$, we define the *concatenation* as $a \cdot b := \langle a_1, \dots, a_i, b_1, b_2, \dots \rangle$.

Definition 28 (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 occurrence of that element in X is removed.

C.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.

C.2.1. URLs

Definition 29. A *URL* is a term of the form

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

with $\text{protocol} \in \{\text{P}, \text{S}\}$ (for **p**lain (HTTP) and **s**ecure (HTTPS)), a domain $\text{host} \in \text{Doms}$, $\text{path} \in \mathbb{S}$, $\text{parameters} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, and $\text{fragment} \in \mathcal{T}_{\mathcal{N}}$. 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 \perp . We sometimes also write $\text{URL}_{\text{path}}^{\text{host}}$ to denote the URL $\langle \text{URL}, \text{S}, \text{host}, \text{path}, \langle \rangle, \perp \rangle$.

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 26):

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

C.2.2. Origins

Definition 30. An *origin* is a term of the form $\langle \text{host}, \text{protocol} \rangle$ with $\text{host} \in \text{Doms}$ and $\text{protocol} \in \{\text{P}, \text{S}\}$. We write **Origins** for the set of all origins.

Example 5. For example, $\langle \text{F00}, \text{S} \rangle$ is the HTTPS origin for the domain F00, while $\langle \text{BAR}, \text{P} \rangle$ is the HTTP origin for the domain BAR.

C.2.3. Cookies

Definition 31. A *cookie* is a term of the form $\langle \text{name}, \text{content} \rangle$ where $\text{name} \in \mathcal{T}_{\mathcal{N}}$, and content is a term of the form $\langle \text{value}, \text{secure}, \text{session}, \text{httpOnly} \rangle$ where $\text{value} \in \mathcal{T}_{\mathcal{N}}$, $\text{secure}, \text{session}, \text{httpOnly} \in \{\top, \perp\}$. 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., $\text{name}.1$) the *prefix* of the name. We write **Cookies** for the set of all cookies and Cookies^V 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.² 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 [3]) of a cookie is set (i.e., name consists of two parts and $\text{name}.1 \equiv \text{__Host}$), 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 $\langle \text{name}, \text{value} \rangle$.

²Note that *secure* cookies can be set over unencrypted connections (c.f. RFC 6265).

C.2.4. HTTP Messages

Definition 32. An *HTTP request* is a term of the form shown in (10). An *HTTP response* is a term of the form shown in (11).

$$\langle \text{HTTPReq}, \text{nonce}, \text{method}, \text{host}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \quad (10)$$

$$\langle \text{HTTPResp}, \text{nonce}, \text{status}, \text{headers}, \text{body} \rangle \quad (11)$$

The components are defined as follows:

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

We write HTTPRequests/HTTPResponses for the set of all HTTP requests or responses, respectively.

Example 6 (HTTP Request and Response).

$$r := \langle \text{HTTPReq}, n_1, \text{POST}, \text{example.com}, /show, \langle \langle \text{index}, 1 \rangle \rangle, [\text{Origin} : \langle \text{example.com}, \mathbb{S} \rangle], \langle \text{foo}, \text{bar} \rangle \rangle \quad (12)$$

$$s := \langle \text{HTTPResp}, n_1, 200, \langle \langle \text{Set-Cookie}, \langle \langle \text{SID}, \langle n_2, \perp, \perp, \top \rangle \rangle \rangle \rangle, \langle \text{somescript}, x \rangle \rangle \quad (13)$$

An HTTP POST request for the URL <http://example.com/show?index=1> is shown in (12), with an Origin header and a body that contains $\langle \text{foo}, \text{bar} \rangle$. A possible response is shown in (13), 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 C.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 33. An *encrypted HTTP request* is of the form $\text{enc}_a(\langle m, k' \rangle, k)$, where $k \in \text{terms}$, $k' \in \mathcal{N}$, and $m \in \text{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\text{enc}_s(m', k')$, where $m' \in \text{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses HTTPSRequests or 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.

$$\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}})) \quad (14)$$

$$\text{enc}_s(s, k') \quad (15)$$

The term (14) shows an encrypted request (with r as in (12)). It is encrypted using the public key $\text{pub}(k_{\text{example.com}})$. The term (15) is a response (with s as in (13)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (14).

C.2.5. DNS Messages

Definition 34. A *DNS request* is a term of the form $\langle \text{DNSResolve}, \text{domain}, \text{nonce} \rangle$ where $\text{domain} \in \text{Doms}$, $\text{nonce} \in \mathcal{N}$. We call the set of all DNS requests DNSRequests .

Definition 35. A *DNS response* is a term of the form $\langle \text{DNSResolved}, \text{domain}, \text{result}, \text{nonce} \rangle$ with $\text{domain} \in \text{Doms}$, $\text{result} \in \text{IPs}$, $\text{nonce} \in \mathcal{N}$. We call the set of all DNS responses 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.

C.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 36 (Generic Atomic Processes and Systems). A (*generic*) *atomic process* is a tuple

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

where $I^p \subseteq \text{IPs}$, $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^\nu} \times \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (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 (η_1, η_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 37 (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 sequence³ (e_1, e_2, \dots) of events waiting to be delivered, and N is an infinite sequence of nonces (n_1, n_2, \dots) .

³Here: Not in the sense of terms as defined earlier.

Definition 38 (Processing Steps). A *processing step* of the system \mathcal{P} is of the form

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

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

with

1. $((e_{\text{in}}, S(p)), (E_{\text{out}}^\nu, s^\nu)) \in R^p$ and $a \in I^p$,
2. $E_{\text{out}} = E_{\text{out}}^\nu[\eta_1/v_1, \dots, \eta_i/v_i]$,
3. $S'(p) = s^\nu[\eta_1/v_1, \dots, \eta_i/v_i]$ 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 occurrences 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 39 (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 $((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \dots)$ such that $S^0(p) = s_0^p$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ 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 $P = (S, E, N) \rightarrow (S', E', N')$ 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 $\langle a, a, \text{TRIGGER} \rangle$ for each $a \in \text{IPs}$, interleaved by address.

C.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 40 (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 $m_1, \dots, m_n \in M$ with $n \geq 0$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \dots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \dots, m_n/x_n]$. 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 $\{\text{enc}_a(\langle a, b, c \rangle, \text{pub}(k)), k\}$, i.e., $a \in d_\emptyset(\{\text{enc}_a(\langle a, b, c \rangle, \text{pub}(k)), k\})$.

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 41 (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 $E, s' \in d_{V_{\text{process}}}(\{e, s\})$.

C.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 42 (Atomic Attacker Process). An (atomic) attacker process for a set of sender addresses $A \subseteq \text{IPs}$ 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 $E = \langle \langle a_1, f_1, m_1 \rangle, \dots, \langle a_n, f_n, m_n \rangle \rangle$ with $n \in \mathbb{N}$, $a_1, \dots, a_n \in \text{IPs}$, $f_1, \dots, f_n \in A$, $m_1, \dots, m_n \in d_{V_{\text{process}}}(\{e, s\})$. The initial state of an (atomic) attacker process is empty, i.e., $s_0 := \langle \rangle$.

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.

C.6. Notations for Functions and Algorithms

When describing algorithms, we use the following notations:

C.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** $\langle \text{Constant}, x, y \rangle \equiv t$ **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 **Constant** $\equiv \pi_1(t)$ and if $|\langle \text{Constant}, x, y \rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

C.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.

C.6.3. Stop without output

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

C.6.4. Placeholders

In several places throughout the algorithms we use placeholders to generate “fresh” nonces as described in our communication model (see Definition 16). Table 2 shows a list of some of the placeholders, generally denoted by ν with some subscript to distinguish between multiple fresh values.

C.6.5. Abbreviations for URLs and Origins

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

C.7. Browsers

Here, we present the formal model of browsers.

C.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).

Placeholder	Usage
ν_1	Algorithm 17, new window nonces
ν_2	Algorithm 17, new HTTP request nonce
ν_3	Algorithm 17, lookup key for pending HTTP requests entry
ν_4	Algorithm 15, new HTTP request nonce (multiple lines)
ν_5	Algorithm 15, new subwindow nonce
ν_6	Algorithm 16, new HTTP request nonce
ν_7	Algorithm 16, new document nonce
ν_8	Algorithm 12, lookup key for pending DNS entry
ν_9	Algorithm 9, new window nonce
ν_{10}, \dots	Algorithm 15, replacement for placeholders in script output

Table 2: List of placeholders used in browser algorithms.

Definition 43 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 44 (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 λ_1, \dots which the browser will replace by (otherwise unused) placeholders ν_1, \dots 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 45 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$.

C.7.2. Web Browser State

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

Definition 46. A *window* is a term of the form $w = \langle \text{nonce}, \text{documents}, \text{opener} \rangle$ with $\text{nonce} \in \mathcal{N}$, $\text{documents} \subset {}^\diamond \text{Documents}$ (defined below), $\text{opener} \in \mathcal{N} \cup \{\perp\}$ where $d.\text{active} = \top$ for exactly one $d \in {}^\diamond \text{documents}$ 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 $w.\text{activedocument}$ 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., $b.\text{opener} = a.\text{nonce}$.

Definition 47. A *document* d is a term of the form

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

where $nonce \in \mathcal{N}$, $location \in \text{URLs}$, $headers \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, $referrer \in \text{URLs} \cup \{\perp\}$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $subwindows \subset^{\langle \rangle} \text{Windows}$, $active \in \{\top, \perp\}$. 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 **Documents** for the set of all documents. For a document term d we write $d.\text{origin}$ to denote the origin of the document, i.e., the term $\langle d.\text{location}.\text{host}, d.\text{location}.\text{protocol} \rangle \in \text{Origins}$.

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

Definition 48. For two window terms w and w' we write

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

if $w \in^{\langle \rangle} w'.\text{activedocument.subwindows}$. 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 49. A reference for a normal HTTP(S) request is a sequence of the form $\langle \text{REQ}, nonce \rangle$, where $nonce$ is a window reference. A reference for a XMLHttpRequest is a sequence of the form $\langle \text{XHR}, nonce, xhrreference \rangle$, 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 24.

Definition 50. 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:

- $windows \subset^{\langle \rangle} \text{Windows}$ 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).
- $secrets \in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$ 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 **Cookies** modeling cookies that are stored for specific domains.
- $localStorage \in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$ stores the data saved by scripts using the localStorage API (separated by origins).
- $sessionStorage \in [OR \times \mathcal{T}_{\mathcal{N}}]$ for $OR := \{\langle o, r \rangle \mid o \in \text{Origins}, r \in \mathcal{N}\}$ similar to localStorage, but the data in sessionStorage is additionally separated by top-level windows.
- $keyMapping \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$ 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).
- $DNSaddress \in \text{IPs}$ 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.
- $isCorrupted \in \{\perp, \text{FULLCORRUPT}, \text{CLOSECORRUPT}\}$ 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).

C.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 pointers⁴ to windows in the window list $s.\text{windows}$ 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 51. For a browser state s we denote by *Subwindows*(s) the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in \langle \rangle s.\text{windows}$ there is a $\bar{p} \in \text{Subwindows}(s)$ such that $s.\bar{p} = w$. (2) For all $\bar{p} \in \text{Subwindows}(s)$, the active document d of the window $s.\bar{p}$ and every subwindow w of d there is a pointer $\bar{p}' \in \text{Subwindows}(s)$ such that $s.\bar{p}' = w$.

Given a browser state s , the set *Docs*(s) of pointers to active documents is the minimal set such that for every $\bar{p} \in \text{Subwindows}(s)$ with $s.\bar{p}.\text{activedocument} \neq \langle \rangle$, there exists a pointer $\bar{p}' \in \text{Docs}(s)$ with $s.\bar{p}' = s.\bar{p}.\text{activedocument}$.

By $\text{Subwindows}^+(s)$ and $\text{Docs}^+(s)$ 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.

⁴Recall the definition of a pointer in Definition 26.

Definition 52. Let s be a browser state and d a document. By $\text{Clean}(s, d)$ we denote the term that equals $s.\text{windows}$ 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 53. For a sequence of cookies (with pairwise different names) *oldcookies*, a sequence of cookies *newcookies*, and a string $\text{protocol} \in \{\text{P}, \text{S}\}$, the set $\text{CookieMerge}(\text{oldcookies}, \text{newcookies}, \text{protocol})$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\text{content.httpOnly} \equiv \top$ or where $(c.\text{name}.1 \equiv __\text{Host}) \wedge ((\text{protocol} \equiv \text{P}) \vee (c.\text{secure} \equiv \perp))$. For any $c, c' \in {}^\diamond \text{newcookies}$, $c.\text{name} \equiv c'.\text{name}$, 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 $c_{\text{old}} \in {}^\diamond \text{oldcookies}$, $c_{\text{new}} \in {}^\diamond \text{newcookies}$, $c_{\text{old}}.\text{name} \equiv c_{\text{new}}.\text{name}$, add c_{new} to m if $c_{\text{old}}.\text{content.httpOnly} \equiv \perp$ and add c_{old} to m otherwise. The result of $\text{CookieMerge}(\text{oldcookies}, \text{newcookies}, \text{protocol})$ 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 54. For a sequence of cookies (with pairwise different names) *oldcookies*, a cookie c , and a string $\text{protocol} \in \{\text{P}, \text{S}\}$ (denoting whether the HTTP response was received from an insecure or a secure origin), the sequence $\text{AddCookie}(\text{oldcookies}, c, \text{protocol})$ is defined by the following algorithm: Let $m := \text{oldcookies}$. If $(c.\text{name}.1 \equiv __\text{Host}) \wedge \neg((\text{protocol} \equiv \text{S}) \wedge (c.\text{secure} \equiv \top))$, then return m , else: Remove any c' from m that has $c.\text{name} \equiv c'.\text{name}$. 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 55. The set $\text{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \text{Subwindows}(s')$ 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 $\overline{w}' \in \text{Subwindows}(s')$ the following is true:

- If $s'.\overline{w}'.\text{activedocument.origin} \equiv s'.\overline{w}.\text{activedocument.origin}$ (i.e., the active documents in \overline{w} and \overline{w}' are same-origin), then $\overline{w}' \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w}' \wedge \nexists \overline{w}'' \in \text{Subwindows}(s') \text{ with } s'.\overline{w}' \xrightarrow{\text{childof}^*} s'.\overline{w}''$ (\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 $\exists \overline{p} \in \text{Subwindows}(s')$ such that $s'.\overline{w}' \xrightarrow{\text{childof}^+} s'.\overline{p}$
 $\wedge s'.\overline{p}.\text{activedocument.origin} = s'.\overline{w}.\text{activedocument.origin}$ (\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 $\exists \overline{p} \in \text{Subwindows}(s')$ such that $s'.\overline{w}'.\text{opener} = s'.\overline{p}.\text{nonce} \wedge \overline{p} \in \overline{W}$ (\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 9 Web Browser Model: Determine window for navigation.

```
1: function GETNAVIGABLEWINDOW( $\bar{w}$ ,  $window$ ,  $noreferrer$ ,  $s'$ )
2:   if  $window \equiv \_BLANK$  then  $\rightarrow$  Open a new window when  $\_BLANK$  is used
3:     if  $noreferrer \equiv \top$  then
4:       let  $w' := \langle \nu_9, \langle \rangle, \perp \rangle$ 
5:     else
6:       let  $w' := \langle \nu_9, \langle \rangle, s'.\bar{w}.nonce \rangle$ 
7:       let  $s'.windows := s'.windows + \langle \rangle w'$ 
         $\hookrightarrow$  and let  $w'$  be a pointer to this new element in  $s'$ 
8:       return  $w'$ 
9:   let  $w' \leftarrow \text{NavigableWindows}(\bar{w}, s')$  such that  $s'.w'.nonce \equiv window$ 
         $\hookrightarrow$  if possible; otherwise return  $\bar{w}$ 
10:  return  $w'$ 
```

Algorithm 10 Web Browser Model: Determine same-origin window.

```
1: function GETWINDOW( $\bar{w}$ ,  $window$ ,  $s'$ )
2:   let  $w' \leftarrow \text{Subwindows}(s')$  such that  $s'.w'.nonce \equiv window$ 
         $\hookrightarrow$  if possible; otherwise return  $\bar{w}$ 
3:   if  $s'.w'.activatedocument.origin \equiv s'.\bar{w}.activatedocument.origin$  then
4:     return  $w'$ 
5:   return  $\bar{w}$ 
```

Functions

- The function GETNAVIGABLEWINDOW (Algorithm 9) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\bar{w}$ 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 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, \bar{w} is returned instead (the script navigates its own window).
- The function GETWINDOW (Algorithm 10) 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 11) 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 11 Web Browser Model: Cancel pending requests for given window.

```
1: function CANCELNAV( $reference$ ,  $s'$ )
2:   remove all  $\langle reference, req, url, key, f \rangle$  from  $s'.pendingRequests$  for any  $req, url, key, f$ 
3:   remove all  $\langle x, \langle reference, message, url \rangle \rangle$  from  $s'.pendingDNS$ 
         $\hookrightarrow$  for any  $x, message, url$ 
4:   return  $s'$ 
```

Algorithm 12 Web Browser Model: Prepare headers, do DNS resolution, save message.

```
1: function HTTP_SEND(reference, message, url, origin, referrer, referrerPolicy, a, s')
2:   if message.host  $\in \langle \rangle$  s'.sts then
3:     let url.protocol := S
4:   let cookies :=  $\langle \{ \langle c.name, c.content.value \rangle \mid c \in \langle \rangle s'.cookies[message.host] \}$ 
    $\hookrightarrow \wedge (c.content.secure \equiv \top \implies (url.protocol \equiv S)) \rangle \rangle$ 
5:   let message.headers[Cookie] := cookies
6:   if origin  $\neq \perp$  then
7:     let message.headers[Origin] := origin
8:   if referrerPolicy  $\equiv$  no-referrer then
9:     let referrer :=  $\perp$ 
10:  if referrer  $\neq \perp$  then
11:    if referrerPolicy  $\equiv$  origin then
12:      let referrer :=  $\langle URL, referrer.protocol, referrer.host, /, \langle \rangle, \perp \rangle$ 
       $\rightarrow$  Referrer stripped down to origin.
13:    let referrer.fragment :=  $\perp$ 
       $\rightarrow$  Browsers do not send fragment identifiers in the Referer header.
14:    let message.headers[Referer] := referrer
15:  let s'.pendingDNS[ $\nu_8$ ] :=  $\langle reference, message, url \rangle$ 
16:  stop  $\langle \langle s'.DNSAddress, a, \langle DNSResolve, message.host, \nu_8 \rangle \rangle \rangle, s'$ 
```

Algorithm 13 Web Browser Model: Navigate a window backward.

```
1: function NAVBACK( $\overline{w'}$ , s')
2:   if  $\exists \bar{j} \in \mathbb{N}, \bar{j} > 1$  such that s'. $\overline{w'}$ .documents. $\bar{j}$ .active  $\equiv \top$  then
3:     let s'. $\overline{w'}$ .documents. $\bar{j}$ .active :=  $\perp$ 
4:     let s'. $\overline{w'}$ .documents. $(\bar{j} - 1)$ .active :=  $\top$ 
5:     let s' := CANCELNAV(s'. $\overline{w'}$ .nonce, s')
6:   stop  $\langle \rangle, s'$ 
```

Algorithm 14 Web Browser Model: Navigate a window forward.

```
1: function NAVFORWARD( $\overline{w'}$ , s')
2:   if  $\exists \bar{j} \in \mathbb{N}$  such that s'. $\overline{w'}$ .documents. $\bar{j}$ .active  $\equiv \top$ 
    $\hookrightarrow \wedge s'.\overline{w'}$ .documents. $(\bar{j} + 1) \in \text{Documents}$  then
3:     let s'. $\overline{w'}$ .documents. $\bar{j}$ .active :=  $\perp$ 
4:     let s'. $\overline{w'}$ .documents. $(\bar{j} + 1)$ .active :=  $\top$ 
5:     let s' := CANCELNAV(s'. $\overline{w'}$ .nonce, s')
6:   stop  $\langle \rangle, s'$ 
```

Algorithm 15 Web Browser Model: Execute a script.

```
1: function RUNSCRIPT( $\bar{w}, \bar{d}, a, s'$ )
2:   let  $tree := \text{Clean}(s', s'.\bar{d})$ 
3:   let  $cookies := \langle \{ \langle c.name, c.content.value \rangle \mid c \in {}^\langle \rangle s'.cookies[s'.\bar{d}.origin.host] \}$ 
    $\hookrightarrow \wedge c.content.httpOnly \equiv \perp$ 
    $\hookrightarrow \wedge (c.content.secure \equiv \top \implies (s'.\bar{d}.origin.protocol \equiv S)) \rangle \rangle$ 
4:   let  $tlw \leftarrow s'.windows$  such that  $tlw$  is the top-level window containing  $\bar{d}$ 
5:   let  $sessionStorage := s'.sessionStorage[\langle s'.\bar{d}.origin, tlw.nonce \rangle]$ 
6:   let  $localStorage := s'.localStorage[s'.\bar{d}.origin]$ 
7:   let  $secrets := s'.secrets[s'.\bar{d}.origin]$ 
8:   let  $R := \text{script}^{-1}(s'.\bar{d}.script)$  if possible; otherwise stop
9:   let  $in := \langle tree, s'.\bar{d}.nonce, s'.\bar{d}.scriptstate, s'.\bar{d}.scriptinputs, cookies,$ 
    $\hookrightarrow localStorage, sessionStorage, s'.ids, secrets \rangle$ 
10:  let  $state' \leftarrow \mathcal{T}_{\mathcal{H}}(V_{\text{process}}), cookies' \leftarrow Cookies', localStorage' \leftarrow \mathcal{T}_{\mathcal{H}}(V_{\text{process}}),$ 
    $\hookrightarrow sessionStorage' \leftarrow \mathcal{T}_{\mathcal{H}}(V_{\text{process}}), command \leftarrow \mathcal{T}_{\mathcal{H}}(V_{\text{process}}),$ 
    $\hookrightarrow out := \langle state', cookies', localStorage', sessionStorage', command \rangle$ 
    $\hookrightarrow$  such that  $out := out^\lambda[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$  with  $(in, out^\lambda) \in R$ 
11:  let  $s'.cookies[s'.\bar{d}.origin.host] :=$ 
    $\hookrightarrow \langle \text{CookieMerge}(s'.cookies[s'.\bar{d}.origin.host], cookies', s'.\bar{d}.origin.protocol) \rangle$ 
12:  let  $s'.localStorage[s'.\bar{d}.origin] := localStorage'$ 
13:  let  $s'.sessionStorage[\langle s'.\bar{d}.origin, tlw.nonce \rangle] := sessionStorage'$ 
14:  let  $s'.\bar{d}.scriptstate := state'$ 
15:  let  $referrer := s'.\bar{d}.location$ 
16:  let  $referrerPolicy := s'.\bar{d}.headers[ReferrerPolicy]$ 
17:  let  $docorigin := s'.\bar{d}.origin$ 
18:  switch  $command$  do
19:    case  $\langle HREF, url, hrefwindow, norereferrer \rangle$ 
20:      let  $\bar{w}' := \text{GETNAVIGABLEWINDOW}(\bar{w}, hrefwindow, norereferrer, s')$ 
21:      let  $reference := \langle REQ, s'.\bar{w}'.nonce \rangle$ 
22:      let  $req := \langle \text{HTTPReq}, \nu_4, GET, url.host, url.path, url.parameters, \langle \rangle, \langle \rangle \rangle$ 
23:      if  $norereferrer \equiv \top$  then
24:        let  $referrerPolicy := norereferrer$ 
25:        let  $s' := \text{CANCELNAV}(reference, s')$ 
26:        call  $\text{HTTP\_SEND}(reference, req, url, \perp, referrer, referrerPolicy, a, s')$ 
27:    case  $\langle IFRAME, url, window \rangle$ 
28:      if  $window \equiv \_SELF$  then
29:        let  $\bar{w}' := \bar{w}$ 
30:      else
31:        let  $\bar{w}' := \text{GETWINDOW}(\bar{w}, window, s')$ 
32:        let  $req := \langle \text{HTTPReq}, \nu_4, GET, url.host, url.path, url.parameters, \langle \rangle, \langle \rangle \rangle$ 
33:        let  $w' := \langle \nu_5, \langle \rangle, \perp \rangle$ 
34:        let  $s'.\bar{w}'.activatedocument.subwindows := s'.\bar{w}'.activatedocument.subwindows + {}^\langle \rangle w'$ 
35:        call  $\text{HTTP\_SEND}(\langle REQ, \nu_5 \rangle, req, url, \perp, referrer, referrerPolicy, a, s')$ 
```

— This algorithm is continued on the next page. —

```

36:   case  $\langle \text{FORM}, url, method, data, hrefwindow \rangle$ 
37:     if  $method \notin \{\text{GET}, \text{POST}\}$  then
38:       stop
39:     let  $\overline{w'} := \text{GETNAVIGABLEWINDOW}(\overline{w}, hrefwindow, \perp, s')$ 
40:     let  $reference := \langle \text{REQ}, s'.\overline{w'}.nonce \rangle$ 
41:     if  $method = \text{GET}$  then
42:       let  $body := \langle \rangle$ 
43:       let  $parameters := data$ 
44:       let  $origin := \perp$ 
45:     else
46:       let  $body := data$ 
47:       let  $parameters := url.parameters$ 
48:       let  $origin := docorigin$ 
49:     let  $req := \langle \text{HTTPReq}, \nu_4, method, url.host, url.path, parameters, \langle \rangle, body \rangle$ 
50:     let  $s' := \text{CANCELNAV}(reference, s')$ 
51:     call  $\text{HTTP\_SEND}(reference, req, url, origin, referrer, referrerPolicy, a, s')$ 
52:   case  $\langle \text{SETSCRIPT}, window, script \rangle$ 
53:     let  $\overline{w'} := \text{GETWINDOW}(\overline{w}, window, s')$ 
54:     let  $s'.\overline{w'}.activedocument.script := script$ 
55:     stop  $\langle \rangle, s'$ 
56:   case  $\langle \text{SETSCRIPTSTATE}, window, scriptstate \rangle$ 
57:     let  $\overline{w'} := \text{GETWINDOW}(\overline{w}, window, s')$ 
58:     let  $s'.\overline{w'}.activedocument.scriptstate := scriptstate$ 
59:     stop  $\langle \rangle, s'$ 
60:   case  $\langle \text{XMLHTTPREQUEST}, url, method, data, xhrreference \rangle$ 
61:     if  $method \in \{\text{CONNECT}, \text{TRACE}, \text{TRACK}\} \vee xhrreference \notin V_{\text{process}} \cup \{\perp\}$  then
62:       stop
63:     if  $url.host \neq docorigin.host \vee url.protocol \neq docorigin.protocol$  then
64:       stop
65:     if  $method \in \{\text{GET}, \text{HEAD}\}$  then
66:       let  $data := \langle \rangle$ 
67:       let  $origin := \perp$ 
68:     else
69:       let  $origin := docorigin$ 
70:     let  $req := \langle \text{HTTPReq}, \nu_4, method, url.host, url.path, url.parameters, \langle \rangle, data \rangle$ 
71:     let  $reference := \langle \text{XHR}, s'.\overline{d}.nonce, xhrreference \rangle$ 
72:     call  $\text{HTTP\_SEND}(reference, req, url, origin, referrer, referrerPolicy, a, s')$ 
73:   case  $\langle \text{BACK}, window \rangle$ 
74:     let  $\overline{w'} := \text{GETNAVIGABLEWINDOW}(\overline{w}, window, \perp, s')$ 
75:     call  $\text{NAVBACK}(\overline{w'}, s')$ 
76:   case  $\langle \text{FORWARD}, window \rangle$ 
77:     let  $\overline{w'} := \text{GETNAVIGABLEWINDOW}(\overline{w}, window, \perp, s')$ 
78:     call  $\text{NAVFORWARD}(\overline{w'}, s')$ 
79:   case  $\langle \text{CLOSE}, window \rangle$ 
80:     let  $\overline{w'} := \text{GETNAVIGABLEWINDOW}(\overline{w}, window, \perp, s')$ 
81:     remove  $s'.\overline{w'}$  from the sequence containing it
82:     stop  $\langle \rangle, s'$ 
83:   case  $\langle \text{POSTMESSAGE}, window, message, origin \rangle$ 
84:     let  $\overline{w'} \leftarrow \text{Subwindows}(s')$  such that  $s'.\overline{w'}.nonce \equiv window$ 
85:     if  $\exists \bar{j} \in \mathbb{N}$  such that  $s'.\overline{w'}.documents.\bar{j}.active \equiv \top$ 
       $\hookrightarrow \wedge (origin \neq \perp \implies s'.\overline{w'}.documents.\bar{j}.origin \equiv origin)$  then
86:       let  $s'.\overline{w'}.documents.\bar{j}.scriptinputs := s'.\overline{w'}.documents.\bar{j}.scriptinputs$ 
       $\hookrightarrow + \langle \rangle \langle \text{POSTMESSAGE}, s'.\overline{w'}.nonce, docorigin, message \rangle$ 
87:     stop  $\langle \rangle, s'$ 
88:   case else
89:     stop

```

Algorithm 16 Web Browser Model: Process an HTTP response.

```
1: function PROCESSRESPONSE(response, reference, request, requestUrl, a, f, s')
2:   if Set-Cookie  $\in$  response.headers then
3:     for each  $c \in {}^\diamond \text{response.headers}[\text{Set-Cookie}], c \in \text{Cookies}$  do
4:       let s'.cookies [request.host]
          $\hookrightarrow := \text{AddCookie}(s'.\text{cookies}[\text{request.host}], c, \text{requestUrl.protocol})$ 
5:   if Strict-Transport-Security  $\in$  response.headers  $\wedge$  requestUrl.protocol  $\equiv$  S then
6:     let s'.sts  $:= s'.sts + {}^\diamond \text{request.host}$ 
7:   if Referer  $\in$  request.headers then
8:     let referrer  $:= \text{request.headers}[\text{Referer}]$ 
9:   else
10:    let referrer  $:= \perp$ 
11:   if Location  $\in$  response.headers  $\wedge$  response.status  $\in \{303, 307\}$  then
12:     let url  $:= \text{response.headers}[\text{Location}]$ 
13:     if url.fragment  $\equiv \perp$  then
14:       let url.fragment  $:= \text{requestUrl.fragment}$ 
15:     let method'  $:= \text{request.method}$ 
16:     let body'  $:= \text{request.body}$ 
17:     if Origin  $\in$  request.headers
        $\hookrightarrow \wedge \text{request.headers}[\text{Origin}] \neq \diamond$ 
        $\hookrightarrow \wedge (\langle \text{url.host}, \text{url.protocol} \rangle \equiv \langle \text{request.host}, \text{requestUrl.protocol} \rangle$ 
        $\hookrightarrow \vee \langle \text{request.host}, \text{requestUrl.protocol} \rangle \equiv \text{request.headers}[\text{Origin}])$  then
18:       let origin  $:= \text{request.headers}[\text{Origin}]$ 
19:     else
20:       let origin  $:= \diamond$ 
21:     if response.status  $\equiv 303 \wedge \text{request.method} \notin \{\text{GET}, \text{HEAD}\}$  then
22:       let method'  $:= \text{GET}$ 
23:       let body'  $:= \langle \rangle$ 
24:     if  $\exists \bar{w} \in \text{Subwindows}(s')$  such that s'.w.nonce  $\equiv \pi_2(\text{reference})$  then  $\rightarrow$  Do not redirect XHRs.
25:       let req  $:= \langle \text{HTTPReq}, v_6, \text{method}', \text{url.host}, \text{url.path}, \text{url.parameters}, \langle \rangle, \text{body}' \rangle$ 
26:       let referrerPolicy  $:= \text{response.headers}[\text{ReferrerPolicy}]$ 
27:       call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s')
28:     else
29:       stop  $\langle \rangle, s'$ 
```

This algorithm is continued on the next page.

```

30:  switch  $\pi_1(\text{reference})$  do
31:    case REQ
32:      let  $\bar{w} \leftarrow \text{Subwindows}(s')$  such that  $s'.\bar{w}.\text{nonce} \equiv \pi_2(\text{reference})$  if possible;
         $\hookrightarrow$  otherwise stop  $\rightarrow$  normal response
33:      if  $\text{response.body} \not\sim \langle *, * \rangle$  then
34:        stop  $\langle \rangle, s'$ 
35:      let  $\text{script} := \pi_1(\text{response.body})$ 
36:      let  $\text{scriptstate} := \pi_2(\text{response.body})$ 
37:      let  $d := \langle \nu_7, \text{requestUrl}, \text{response.headers}, \text{referrer}, \text{script}, \text{scriptstate}, \langle \rangle, \langle \rangle, \top \rangle$ 
38:      if  $s'.\bar{w}.\text{documents} \equiv \langle \rangle$  then
39:        let  $s'.\bar{w}.\text{documents} := \langle d \rangle$ 
40:      else
41:        let  $\bar{i} \leftarrow \mathbb{N}$  such that  $s'.\bar{w}.\text{documents}.\bar{i}.\text{active} \equiv \top$ 
42:        let  $s'.\bar{w}.\text{documents}.\bar{i}.\text{active} := \perp$ 
43:        remove  $s'.\bar{w}.\text{documents}.\bar{i} + 1$  and all following documents
           $\hookrightarrow$  from  $s'.\bar{w}.\text{documents}$ 
44:        let  $s'.\bar{w}.\text{documents} := s'.\bar{w}.\text{documents} + \langle \rangle d$ 
45:      stop  $\langle \rangle, s'$ 
46:    case XHR
47:      let  $\bar{w} \leftarrow \text{Subwindows}(s'), \bar{d}$  such that  $s'.\bar{d}.\text{nonce} \equiv \pi_2(\text{reference})$ 
         $\hookrightarrow \wedge s'.\bar{d} = s'.\bar{w}.\text{activedocument}$  if possible; otherwise stop
         $\rightarrow$  process XHR response
48:      let  $\text{headers} := \text{response.headers} - \text{Set-Cookie}$ 
49:      let  $s'.\bar{d}.\text{scriptinputs} := s'.\bar{d}.\text{scriptinputs} + \langle \rangle$ 
         $\langle \text{XMLHTTPREQUEST}, \text{headers}, \text{response.body}, \pi_3(\text{reference}) \rangle$ 
50:      stop  $\langle \rangle, s'$ 

```

- The function `HTTP_SEND` (Algorithm 12) 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 $s'.\text{pendingDNS}$ until the DNS resolution finishes. *reference* is a reference as defined in Definition 49. *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 13) and `NAVFORWARD` (Algorithm 14), 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 15) performs a script execution step of the script in the document $s'.\bar{d}$ (which is part of the window $s'.\bar{w}$). 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 16) 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 49. *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 56. The pair $((\langle a, f, m \rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 17 (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).

C.8. Definition of Web Browsers

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

Definition 57 (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 58 (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 50) with the following properties:

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

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

C.9. Helper Functions

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

Algorithm 17 Web Browser Model: Main Algorithm.

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

- 1: **let** $s' := s$
- 2: **if** $s.isCorrupted \neq \perp$ **then**
- 3: **let** $s'.pendingRequests := \langle m, s.pendingRequests \rangle$ \rightarrow Collect incoming messages
- 4: **let** $m' \leftarrow d_V(s')$
- 5: **let** $a' \leftarrow IPs$
- 6: **stop** $\langle \langle a', a, m' \rangle \rangle, s'$
- 7: **if** $m \equiv \text{TRIGGER}$ **then** \rightarrow A special trigger message.
- 8: **let** $switch \leftarrow \{\text{script}, \text{urlbar}, \text{reload}, \text{forward}, \text{back}\}$
- 9: **if** $switch \equiv \text{script}$ **then** \rightarrow Run some script.
- 10: **let** $\bar{w} \leftarrow \text{Subwindows}(s')$ **such that** $s'.\bar{w}.documents \neq \langle \rangle$
 \hookrightarrow **if possible; otherwise stop** \rightarrow Pointer to some window.
- 11: **let** n **such that** $s'.\bar{w}.n.active \equiv \top$ **if possible; otherwise stop**
- 12: **let** $\bar{d} := \bar{w} + \langle \rangle n$
- 13: **call** $\text{RUNSCRIPT}(\bar{w}, \bar{d}, a, s')$
- 14: **else if** $switch \equiv \text{urlbar}$ **then** \rightarrow Create some new request.
- 15: **let** $newwindow \leftarrow \{\top, \perp\}$
- 16: **if** $newwindow \equiv \top$ **then** \rightarrow Create a new window.
- 17: **let** $windownonce := \nu_1$
- 18: **let** $w' := \langle windownonce, \langle \rangle, \perp \rangle$
- 19: **let** $s'.windows := s'.windows + \langle \rangle w'$
- 20: **else** \rightarrow Use existing top-level window.
- 21: **let** $tlw \leftarrow \mathbb{N}$ **such that** $s'.tlw.documents \neq \langle \rangle$
 \hookrightarrow **if possible; otherwise stop** \rightarrow Pointer to some top-level window.
- 22: **let** $windownonce := s'.tlw.nonce$
- 23: **let** $protocol \leftarrow \{P, S\}$
- 24: **let** $host \leftarrow \text{Doms}$
- 25: **let** $path \leftarrow \mathbb{S}$
- 26: **let** $fragment \leftarrow \mathbb{S}$
- 27: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
- 28: **let** $url := \langle \text{URL}, protocol, host, path, parameters, fragment \rangle$
- 29: **let** $req := \langle \text{HTTPReq}, \nu_2, \text{GET}, host, path, parameters, \langle \rangle, \langle \rangle \rangle$
- 30: **call** $\text{HTTP_SEND}(\langle \text{REQ}, windownonce \rangle, req, url, \perp, \perp, \perp, a, s')$
- 31: **else if** $switch \equiv \text{reload}$ **then** \rightarrow Reload some document.
- 32: **let** $\bar{w} \leftarrow \text{Subwindows}(s')$ **such that** $s'.\bar{w}.documents \neq \langle \rangle$
 \hookrightarrow **if possible; otherwise stop** \rightarrow Pointer to some window.
- 33: **let** $url := s'.\bar{w}.activedocument.location$
- 34: **let** $req := \langle \text{HTTPReq}, \nu_2, \text{GET}, url.host, url.path, url.parameters, \langle \rangle, \langle \rangle \rangle$
- 35: **let** $referrer := s'.\bar{w}.activedocument.referrer$
- 36: **let** $s' := \text{CANCELNAV}(s'.\bar{w}.nonce, s')$
- 37: **call** $\text{HTTP_SEND}(\langle \text{REQ}, s'.\bar{w}.nonce \rangle, req, url, \perp, referrer, \perp, a, s')$
- 38: **else if** $switch \equiv \text{forward}$ **then**
- 39: **let** $\bar{w} \leftarrow \text{Subwindows}(s')$ **such that** $s'.\bar{w}.documents \neq \langle \rangle$
 \hookrightarrow **if possible; otherwise stop** \rightarrow Pointer to some window.
- 40: **call** $\text{NAVFORWARD}(\bar{w}, s')$
- 41: **else if** $switch \equiv \text{back}$ **then**
- 42: **let** $\bar{w} \leftarrow \text{Subwindows}(s')$ **such that** $s'.\bar{w}.documents \neq \langle \rangle$
 \hookrightarrow **if possible; otherwise stop** \rightarrow Pointer to some window.
- 43: **call** $\text{NAVBACK}(\bar{w}, s')$
- 44: **else if** $m \equiv \text{FULLCORRUPT}$ **then** \rightarrow Request to corrupt browser
- 45: **let** $s'.isCorrupted := \text{FULLCORRUPT}$
- 46: **stop** $\langle \rangle, s'$
- 47: **else if** $m \equiv \text{CLOSECORRUPT}$ **then** \rightarrow Close the browser
- 48: **let** $s'.secrets := \langle \rangle$
- 49: **let** $s'.windows := \langle \rangle$
- 50: **let** $s'.pendingDNS := \langle \rangle$

\rightarrow This algorithm is continued on the next page. \leftarrow

```

51:  let  $s'.pendingRequests := \langle \rangle$ 
52:  let  $s'.sessionStorage := \langle \rangle$ 
53:  let  $s'.cookies \subset \langle \rangle$  Cookies such that
     $\hookrightarrow (c \in \langle \rangle s'.cookies) \iff (c \in \langle \rangle s.cookies \wedge c.content.session \equiv \perp)$ 
54:  let  $s'.isCorrupted := \text{CLOSECORRUPT}$ 
55:  stop  $\langle \rangle, s'$ 
56: else if  $\exists \langle reference, request, url, key, f \rangle \in \langle \rangle s'.pendingRequests$  such that
     $\hookrightarrow \pi_1(\text{dec}_s(m, key)) \equiv \text{HTTPResp}$  then  $\rightarrow$  Encrypted HTTP response
57:   let  $m' := \text{dec}_s(m, key)$ 
58:   if  $m'.nonce \neq request.nonce$  then
59:     stop
60:   remove  $\langle reference, request, url, key, f \rangle$  from  $s'.pendingRequests$ 
61:   call  $\text{PROCESSRESPONSE}(m', reference, request, url, a, f, s')$ 
62: else if  $\pi_1(m) \equiv \text{HTTPResp} \wedge \exists \langle reference, request, url, \perp, f \rangle \in \langle \rangle s'.pendingRequests$  such that
     $\hookrightarrow m.nonce \equiv request.nonce$  then  $\rightarrow$  Plain HTTP Response
63:   remove  $\langle reference, request, url, \perp, f \rangle$  from  $s'.pendingRequests$ 
64:   call  $\text{PROCESSRESPONSE}(m, reference, request, url, a, f, s')$ 
65: else if  $m \in \text{DNSResponses}$  then  $\rightarrow$  Successful DNS response
66:   if  $m.nonce \notin s.pendingDNS \vee m.result \notin \text{IPs}$ 
     $\hookrightarrow \vee m.domain \neq s.pendingDNS[m.nonce].request.host$  then
67:     stop
68:   let  $\langle reference, message, url \rangle := s.pendingDNS[m.nonce]$ 
69:   if  $url.protocol \equiv S$  then
70:     let  $s'.pendingRequests := s'.pendingRequests$ 
     $\hookrightarrow + \langle \rangle \langle reference, message, url, \nu_3, m.result \rangle$ 
71:     let  $message := \text{enc}_a(\langle message, \nu_3 \rangle, s'.keyMapping[message.host])$ 
72:   else
73:     let  $s'.pendingRequests := s'.pendingRequests$ 
     $\hookrightarrow + \langle \rangle \langle reference, message, url, \perp, m.result \rangle$ 
74:   let  $s'.pendingDNS := s'.pendingDNS - m.nonce$ 
75:   stop  $\langle \langle m.result, a, message \rangle \rangle, s'$ 
76: stop

```

Algorithm 18 Function to retrieve an unhandled input message for a script.

```

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

```

Algorithm 19 Function to extract the first script input message matching a specific pattern.

```

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

```

CHOOSEINPUT (Algorithm 18) 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 $\text{CHOOSEINPUT}(s', \text{scriptinputs})$ 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 19) Similar to the function CHOOSEINPUT above, we define the function $\text{CHOOSEFIRSTINPUTPAT}$. 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 $\text{PARENTWINDOW}(\text{tree}, \text{docnonce})$ 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 \perp .

PARENTDOCNONCE The function $\text{PARENTDOCNONCE}(\text{tree}, \text{docnonce})$ 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 $\text{SUBWINDOWS}(\text{tree}, \text{docnonce})$ returns $\langle \rangle$. Otherwise, let $\langle \text{docnonce}, \text{location}, \langle \rangle, \text{referrer}, \text{script},$

$scriptstate, scriptinputs, subwindows, active$ 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.

C.10. DNS Servers

Definition 59. 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

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

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 20.

C.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 20 Relation of a DNS server R^d .

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

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

Definition 60. A *Web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ 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 $p \in \text{Web} \cup \text{Net}$ is an attacker process for some set of sender addresses $A \subseteq \text{IPs}$. For a Web attacker $p \in \text{Web}$, 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 $p' \neq p, p' \in \text{Hon} \cup \text{Web}$. 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 $p \in \text{Hon}$ 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 **script**(s).

Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \text{TRIGGER} \rangle$ 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 .

C.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 61 (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: $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \text{IPs}$ (containing the IP address of a DNS server), $keyMapping \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$ (containing a mapping from domains to public keys), $tlskeys \in [\text{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $corrupt \in \mathcal{T}_{\mathcal{N}}$ (either \perp if the server is not corrupted, or an arbitrary term otherwise).

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 21–25, a generic helper function to merge dictionaries in Algorithm 26, and the main relation in Algorithm 27.

Algorithm 21 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'.pendingDNS[ $\nu_{n0}$ ] :=  $\langle \text{reference}, \text{message} \rangle$ 
3:   stop  $\langle \langle \text{s'}.DNSAddress, a, \langle \text{DNSResolve}, \text{message}.host, \nu_{n0} \rangle \rangle \rangle, s'$ 

```

Algorithm 22 Generic HTTPS Server Model: Default HTTPS response handler.

```

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

```

Algorithm 23 Generic HTTPS Server Model: Default trigger event handler.

```

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

```

Algorithm 24 Generic HTTPS Server Model: Default HTTPS request handler.

```

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

```

Algorithm 25 Generic HTTPS Server Model: Default handler for other messages.

```

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

```

Algorithm 26 Generic HTTPS Server Model: Helper function to merge dictionaries.

→ Merge two dictionaries into one, making sure that there are no duplicate keys (stops the current processing step if there are duplicate keys).

```

1: function MERGE_DICTS(dict1, dict2)
2:   for key ∈  $\mathcal{T}_{\mathcal{N}}$  do
3:     if key ∈ dict2 then
4:       if key ∈ dict1 then
5:         stop → Duplicate key.
6:       let dict1[key] := dict2[key]
7:   return dict1

```

Algorithm 27 Generic HTTPS Server Model: Main relation of a generic HTTPS server

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

```
1: let  $s' := s$ 
2: if  $s'.\text{corrupt} \neq \perp \vee m \equiv \text{CORRUPT}$  then
3:   let  $s'.\text{corrupt} := \langle \langle a, f, m \rangle, s'.\text{corrupt} \rangle$ 
4:   let  $m' \leftarrow d_V(s')$ 
5:   let  $a' \leftarrow \text{IPs}$ 
6:   stop  $\langle \langle a', a, m' \rangle \rangle, s'$ 
7: if  $\exists m_{\text{dec}}, k, k', \text{inDomain}$  such that  $\langle m_{\text{dec}}, k \rangle \equiv \text{dec}_a(m, k') \wedge \langle \text{inDomain}, k' \rangle \in {}^{\langle \rangle} s.\text{tlskeys}$  then
8:   let  $n, \text{method}, \text{path}, \text{parameters}, \text{headers}, \text{body}$  such that
       $\hookrightarrow \langle \text{HTTPReq}, n, \text{method}, \text{inDomain}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$ 
       $\hookrightarrow$  if possible; otherwise stop
9:   call  $\text{PROCESS\_HTTPS\_REQUEST}(m_{\text{dec}}, k, a, f, s')$ 
10: else if  $m \in \text{DNSResponses}$  then  $\rightarrow$  Successful DNS response
11:   if  $m.\text{nonce} \notin s.\text{pendingDNS} \vee m.\text{result} \notin \text{IPs}$ 
       $\hookrightarrow \vee m.\text{domain} \neq s.\text{pendingDNS}[m.\text{nonce}].2.\text{host}$  then
12:     stop
13:     let  $\text{reference} := s.\text{pendingDNS}[m.\text{nonce}].1$ 
14:     let  $\text{request} := s.\text{pendingDNS}[m.\text{nonce}].2$ 
15:     let  $s'.\text{pendingRequests} := s'.\text{pendingRequests} + {}^{\langle \rangle} \langle \text{reference}, \text{request}, \nu_{n1}, m.\text{result} \rangle$ 
16:     let  $\text{message} := \text{enc}_a(\langle \text{request}, \nu_{n1} \rangle, s'.\text{keyMapping}[\text{request}.\text{host}])$ 
17:     let  $s'.\text{pendingDNS} := s'.\text{pendingDNS} - m.\text{nonce}$ 
18:     stop  $\langle \langle m.\text{result}, a, \text{message} \rangle \rangle, s'$ 
19: else if  $\exists \langle \text{reference}, \text{request}, \text{key}, f \rangle \in {}^{\langle \rangle} s'.\text{pendingRequests}$ 
       $\hookrightarrow$  such that  $\pi_1(\text{dec}_s(m, \text{key})) \equiv \text{HTTPResp}$  then  $\rightarrow$  Encrypted HTTP response
20:   let  $m' := \text{dec}_s(m, \text{key})$ 
21:   if  $m'.\text{nonce} \neq \text{request}.\text{nonce}$  then
22:     stop
23:   if  $m' \notin \text{HTTPResponses}$  then
24:     call  $\text{PROCESS\_OTHER}(m, a, f, s')$ 
25:   remove  $\langle \text{reference}, \text{request}, \text{key}, f \rangle$  from  $s'.\text{pendingRequests}$ 
26:   call  $\text{PROCESS\_HTTPS\_RESPONSE}(m', \text{reference}, \text{request}, a, f, s')$ 
27: else if  $m \equiv \text{TRIGGER}$  then  $\rightarrow$  Process was triggered
28:   call  $\text{PROCESS\_TRIGGER}(a, s')$ 
29: else
30:   call  $\text{PROCESS\_OTHER}(m, a, f, s')$ 
31: stop
```

C.13. General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [9].

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

Definition 62 (Emitting Events). Given an atomic process p , an event e , and a finite run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \dots)$ we say that p emits e iff there is a processing step in ρ of the form

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

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

Definition 63. We say that a term t is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \rightarrow s_{i+1}$ of a run $\rho = (s_0, s_1, \dots)$) 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 64. We say that a set of processes P leaks a term t (in a processing step $s_i \rightarrow s_{i+1}$) to a set of processes P' if there exists a message m that is emitted (in $s_i \rightarrow s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \rightarrow s_{i+1}$. 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 65. We say that a DY process p created a message m in a processing step

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

of a run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ 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

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

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

Definition 66. 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 16).

Definition 67. 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 68. 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 $t = f(t_1, \dots, t_n)$, for $f \in \Sigma$ and $t_i \in \mathcal{T}_{\mathcal{N}}(X)$ ($i \in \{1, \dots, n\}$), then $f = \text{pub}$ or for all t_i , k appears only as a public key in t_i .

Definition 69. We say that a *script initiated a request* r if a browser triggered the script (in Line 10 of Algorithm 15) 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 70. 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 27).

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

Lemma 2. If in the processing step $s_i \rightarrow s_{i+1}$ of a run ρ of \mathcal{WS} an honest browser b

- (I) emits an HTTPS request of the form

$$m = \text{enc}_a(\langle \text{req}, k \rangle, \text{pub}(k'))$$

(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 $s_j \rightarrow s_{j+1}$ where the browser b leaks k to $\mathcal{W} \setminus \{u, b\}$ there is a processing step $s_h \rightarrow s_{h+1}$ 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 $\text{pub}(k')$ in the browsers' keymapping $s_0.\text{keyMapping}$ (in its initial state).
- (4) If b accepts a response (say, m') to m in a processing step $s_j \rightarrow s_{j+1}$ 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 $\text{pub}(x)$, 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 $s_j \rightarrow s_{j+1}$ 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 65ff. of Algorithm 17) 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 47ff. of Algorithm 17). 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 17), a host header is always contained in HTTP requests by browsers. From Line 71 of Algorithm 17 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 2, 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., $\nexists s' = (S', E') \in \rho: k \in d_{N^p}(S'(p))$).

Lemma 3. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).windows$ with the origin $\langle dom, S \rangle$ where $dom \in \text{Domain}$, and $S_i(b).keyMapping[dom] \equiv \text{pub}(k)$ with $k \in \mathcal{K}$ 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 16) 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 16. 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 70 (or Line 73 which we can exclude as we will see later) of Algorithm 17. 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 71 of Algorithm 17 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 $S_i(b).keyMapping[dom] = S_0(b).keyMapping[dom] \equiv \text{pub}(k)$ 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 2 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 4. If in a processing step $s_i \rightarrow s_{i+1}$ of a run ρ of \mathcal{WS} an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, S \rangle$ where $S_i(b).keyMapping[dom] \equiv \text{pub}(k)$ with $k \in \mathcal{K}$ 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 16) 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 12), 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 15
- Line 72 of Algorithm 15
- Line 27 of Algorithm 16

■

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 3 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 16 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 2, but is applied to the generic HTTPS server (instead of the Web browser).

Lemma 5. If in the processing step $s_i \rightarrow s_{i+1}$ 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

$$m = \text{enc}_a(\langle req, k \rangle, \text{pub}(k'))$$

(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 *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 $\text{enc}_a(\langle \text{req}', k_2 \rangle, \text{pub}(k'))$, 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 $\text{enc}_a(\langle \text{req}', k_2 \rangle, \text{pub}(k'))$, 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 $s_j \rightarrow s_{j+1}$ where some process leaks k to $\mathcal{W} \setminus \{u, s\}$, there is a processing step $s_h \rightarrow s_{h+1}$ 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 $\text{pub}(k')$ in $S^0(s).\text{keyMapping}$ (i.e., in the initial state of s).
- (4) If s accepts a response (say, m') to m in a processing step $s_j \rightarrow s_{j+1}$ 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 2:

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 $\text{pub}(x)$, 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 $s_j \rightarrow s_{j+1}$ 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 27:

- The message emitted in Line 3 of Algorithm 21 has a different message structure
- As s is honest, it does not send the message of Line 6 of Algorithm 27
- 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 27, is only stored in the *pendingRequests* subterm of the state of s , i.e., in $S^{i+1}(s).\text{pendingRequests}$. Other than that, s only accesses this value in Line 19 of Algorithm 27, 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 27 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 27, 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 21, where a DNS request is sent, and in Line 18 of Algorithm 27, 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 27 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' .