Technical Report

Formal Security Analysis of the OpenID for Verifiable Presentations Specification

Final Report on Deliverable A.1(B)

Fabian Hauck Pedram Hosseyni Ralf Küsters Tim Würtele

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

Institute of Information Security – University of Stuttgart, Germany

July 15, 2025

In Deliverable A.1(A), we presented a formal model of the *OpenID for Verifiable Presentations* specification in conjunction with the *Digital Credentials API* and identified and formalized relevant security properties. This report, Deliverable A.1(B), builds on that foundation by presenting formal proofs for those security properties. These proofs were completed successfully for the original model and security properties, confirming the security of the protocol within the bounds of the mathematical assumptions and formal modeling, i.e., no new vulnerabilities were identified during the verification process.

1. Introduction

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, 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. Section 3 contains informal descriptions of the security properties that we analyze. The appendices contain the full formal model, the formalized security properties, and the security proofs. 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.

In our report for Deliverable A.1(A), we suggested some improvements to the OID4VP specification, which the working group incorporated. For completeness, Appendix D contains these suggestions and the corresponding pull requests that introduce the changes to the specification.

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., $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$), nonces, and constants. The nonces are considered to be infinite-entropy random values, which means they can never be guessed, and must instead be learned, e.g., from received messages. 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 18 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 Creden-

tials" [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

When we created the model for Deliverable A.1(A), the DC API specification [2] was a very early draft with many details missing. Consequently, besides the "usual" modeling decisions, we made 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 an early draft stage, the WebIDL definition of the `DigitalCredential` interface in [2, Section 4.7] 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 a 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 we analyze 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 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. Informal Security Properties

In the following, we give an informal overview of the security properties that we analyze. We note that these are derived from the agreement/contract between the OIDF and the University of Stuttgart. We refer to Appendix B for the formalized properties and Appendix C for the full formal proofs.

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

3.1. 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.

3.2. 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.

3.3. 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. 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. 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. Kusters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019. DOI: 10.1109/sp.2019.00067.

[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. 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. 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.

[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* $\mathcal{VPWS}^{\mathrm{n}} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$. The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker process in Net as well as the set $\mathsf{Hon} := \mathsf{Wallets} \cup \mathsf{Verifiers} \cup \mathsf{Issuers} \cup \mathsf{B}$, 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, $\mathcal{VPWS}^{\mathrm{n}}$ 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 E.11).

The set E^0 of initial events is defined as in Definition 61, i.e., containing infinitely many `TRIGGER` events for each IP address.

$s \in \mathcal{S}$	script(s)	Defined in
R^{att}	`att_script`	Definition 46
script_verifier_authentication	`script_verifier_authentication`	Algorithm 3
script_verifier_index	`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 $\mathcal{VPWS}^{\mathrm{n}}$.

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 $\mathsf{nearby} : \mathcal{W} \times \mathcal{W} \to \{\bot, \top\}$ be a reflexive and symmetric predicate that determines whether two processes are nearby.

Definition 3 (Wallets of ID). Let $\mathsf{walletsOfId} : \mathsf{ID} \to 2^{\mathsf{Wallets}}$ 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 $\mathsf{addr} : \mathsf{Net} \cup \mathsf{Verifiers} \cup \mathsf{Wallets} \cup \mathsf{B} \to 2^{\mathsf{IPs}}$ 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 $\mathsf{dom} : \mathsf{Net} \cup \mathsf{Verifiers} \cup \mathsf{Wallets} \cup \mathsf{Issuers} \to 2^{\mathsf{Doms}}$, 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 $domain \in \mathsf{Doms}$. 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 $\mathsf{governor} : \mathsf{ID} \to \mathcal{W}, \langle name, domain \rangle \longmapsto \mathsf{dom}^{-1}(domain)$ and $\mathsf{ID}^y := \mathsf{governor}^{-1}(y)$.

A.1.4. Keys and Secrets

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

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

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 $\mathsf{tlskey} : \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p, we define $tlskeys^p = \langle \{ \langle d, \mathsf{tlskey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle$ (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 $\mathsf{signkey} : \mathsf{Verifiers} \cup \mathsf{Wallets} \cup \mathsf{Issuers} \to K_{\text{sign}}$ 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 $\mathsf{dcApiKeys} : \mathcal{W} \times \mathcal{W} \to 2^{K_{\text{dcapi}}}$ 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: $\mathsf{dcApiKeys}(p_1, p_2) = \mathsf{dcApiKeys}(p_2, p_1)$
  - Only nearby processes share keys: $\neg\mathsf{nearby}(p_1, p_2) \Rightarrow \mathsf{dcApiKeys}(p_1, p_2) = \emptyset$
  - Disjoint sets: $\mathsf{dcApiKeys}(p_1, p_2) \cap \mathsf{dcApiKeys}(p_3, p_4) \neq \emptyset \Rightarrow \{p_1, p_2\} = \{p_3, p_4\}$

A.2. Issuers

An issuer $i \in \mathsf{Issuers}$ is an instance of the WIM's generic HTTPS web server (see Appendix E.12), modeled as an atomic DY process (I^i, Z^i, R^i, s_0^i) with the address $I^i := \mathsf{addr}(i)$. 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

$$\langle pendingDNS, pendingRequests, DNSaddress, keyMapping, tlskeys, corrupt, signingKey \rangle$$

with $pendingDNS$, $pendingRequests$, $DNSaddress$, $keyMapping$, $tlskeys$, $corrupt$ as in Definition 62, and $signingKey \in \mathcal{N}$. An initial state s_0^i of i is a state of i with

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

- $s_0^i.\texttt{pendingRequests} = \langle\rangle$,
- $s_0^i.\texttt{DNSaddress} \in \mathsf{IPs}$,
- $s_0^i.\texttt{keyMapping} = \langle\{\langle d, \mathsf{pub}(\mathsf{tlskey}(d))\rangle \mid d \in \mathsf{Doms}\}\rangle$,
- $s_0^i.\texttt{tlskeys} = \mathit{tlskeys}^i$,
- $s_0^i.\texttt{corrupt} = \bot$, and
- $s_0^i.\texttt{signingKey} = \mathsf{signkey}(i)$.

Note that the $\mathsf{Issuers}$ 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 $v \in \mathsf{Verifiers}$ is an instance of the WIM's generic HTTPS web server (see Appendix E.12), modeled as an atomic DY process (I^v, Z^v, R^v, s_0^v) with the address $I^v := \mathsf{addr}(v)$. 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

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

with $\mathit{pendingDNS}$, $\mathit{pendingRequests}$, $\mathit{DNSaddress}$, $\mathit{keyMapping}$, $\mathit{tlskeys}$, $\mathit{corrupt}$ as defined in Definition 62, $\mathit{issuers} \in [\mathsf{Doms} \times \mathsf{pub}(\mathcal{N})]$, $\mathit{sessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{signingKey} \in \mathcal{N}$, $\mathit{origins} \in \mathcal{T}_{\mathcal{N}}$, and $\mathit{nonces} \in \mathcal{T}_{\mathcal{N}}$.
An initial state s_0^v of v is a state of v with

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

The relation of a verifier v is based on the generic HTTPS server as defined in Appendix E.12. The following algorithms 1 to 4 overwrite and extend the generic HTTPS server. Methods that are not overwritten are defined as in Appendix E.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 ≡ / **then** → Verifier's home page (see Algorithm 4)
3: **let** m' := $\mathsf{enc_s}(\langle$HTTPResp, m.nonce, 200, $\langle\rangle$, $\langle$script_verifier_index$\rangle\rangle, k)$
4: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'

Generate authorization request

5: **else if** m.path ≡ /start **then**
6: **let** *nonce* := ν_{nonce} → Randomness for request, see [OID4VP, Section 14.1]
7: **let** s'.nonces := s'.nonces $+^{\langle\rangle}$ *nonce* → Not bound to browser session, see Section 2.6
8: **let** *parameters* := [response_type: vp_token] → See Section 2.7
9: **let** *parameters*[nonce] := *nonce*
10: **let** *parameters*[response_mode] := dc_api → See Section 2.7
11: **let** *signedRequest* ← $\{\top, \bot\}$ → See Section 2.6
12: **if** *signedRequest* ≡ $\top$ **then** → Create a signed request
13: **let** *protocol* := openid4vp-v1-signed
14: **let** *parameters*[client_id] := m.host
15: **let** *parameters*[expected_origins] := s'.origins
16: **let** *reqPars* := [request: sig(*parameters*, s'.signingKey)]
17: **else** → Create an unsigned request
18: **let** *protocol* := openid4vp-v1-unsigned
19: **let** *reqPars* := *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** *scriptState* := $\langle$*protocol*, *reqPars*$\rangle$
21: **let** m' := $\mathsf{enc_s}(\langle$HTTPResp, m.nonce, 200, $\langle\rangle$, ↪ $\langle$script_verifier_authentication, *scriptState*$\rangle\rangle, k)$
22: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'

Receive authorization response

23: **else if** m.path ≡ /dc-api-response ∧ m.method ≡ POST **then**
24: **let** *dcApiResponse* := m.body
25: **let** *presentation* := *dcApiResponse*[vp_token]
26: **let** *retVal* := VERIFY_PRESENTATION($m, s', presentation$) → 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** $\langle\langle id, iss\rangle, s'\rangle \equiv$ *retVal* **if possible; otherwise stop**
28: **let** *serviceTokenID* := $\nu_{\text{serviceTokenID}}$
29: **let** s'.sessions[*serviceTokenID*] := $\langle id, iss\rangle$
30: **let** *url* := $\langle$URL, S, m.host, /, $\langle\rangle, \bot\rangle$ → Send user back to verifier home page.
31: **let** *headers* := $\langle\langle$Location, *url*$\rangle\rangle$
32: **let** *headers* := *headers* $+^{\langle\rangle}$ $\langle$Set-Cookie, [serviceTokenID: $\langle$*serviceTokenID*, $\top, \top, \top\rangle$]$\rangle$
33: **let** m' := $\mathsf{enc_s}(\langle$HTTPResp, m.nonce, 303, *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.

function VERIFY_PRESENTATION(m, s', *presentation*)
  let *credential*, *nonce*, *aud* **such that**
  ↪ $\langle$*credential*, *nonce*, *aud*$\rangle \equiv$ extractmsg(*presentation*)
  ↪ **if possible; otherwise stop** → See Section 2.3.
  let *origin* **such that** $\langle$`origin`, *origin*$\rangle \equiv$ *aud* **if possible; otherwise stop**
  if *origin* $\notin^{\langle\rangle}$ s'.`origins` **then**
    stop → Check aud claim, see [OID4VP, Section 14.1].
  if *nonce* $\notin^{\langle\rangle}$ s'.`nonces` **then**
    stop → Check nonce, see [OID4VP, Section 14.1].
  let s'.`nonces` := s'.`nonces` $-^{\langle\rangle}$ *nonce* → Nonce can only be used once, see [OID4VP, Section 14.1].
  → See Section 2.1 for a description of our model's credential format.
  let *id*, *pubKey* **such that** $\langle$*id*, *pubKey*$\rangle \equiv$ extractmsg(*credential*) **if possible; otherwise stop**
  let *name*, *iss* **such that** $\langle$*name*, *iss*$\rangle \equiv$ *id* **if possible; otherwise stop**
  if checksig(*presentation*, *pubKey*) $\not\equiv \top$ **then**
    stop → Check proof of possession, see [OID4VP, Section 14.1].
  if checksig(*credential*, s'.`issuers`[*iss*]) $\not\equiv \top$ **then**
    stop → Check credential validity, see Section 2.2.
  → If this function returns anything, then all checks passed, i.e., the presentation is accepted by v.
  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*$\rangle$
——————DC API response from browser to script, POST that response to the script's origin.——————
if $\exists$ *response* **such that** $\langle$`DC_API_RESPONSE`, *response*$\rangle \equiv$ *scriptstate* **then**
  let *url* := GETURL(*tree*, *docnonce*)
  let *url′* := $\langle$`URL`, `S`, *url*.`host`, /`dc-api-response`, $\langle\rangle, \bot\rangle$
  let *command* := $\langle$`FORM`, *url′*, `POST`, *response*, $\bot\rangle$
————————————————Hand a DC API request to the browser's API.————————————————
else
  let *command* := $\langle$`DCAPI`, *scriptstate*.1, *scriptstate*.2$\rangle$
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*$\rangle$
let *switch* ← {`oid4vp-flow`, `link`}
if *switch* ≡ `oid4vp-flow` **then** → Start a new OID4VP flow.
  let *url* := GETURL(*tree*, *docnonce*)
  let *url′* := $\langle$`URL`, `S`, *url*.`host`, /`start`, $\langle\rangle, \bot\rangle$
  let *command* := $\langle$`HREF`, *url′*, $\bot, \bot\rangle$
  stop $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$
else → Follow a random link.
  let *protocol* ← {`P`, `S`}
  let *host* ← Doms
  let *path* ← $\mathbb{S}$
  let *parameters* ← $[\mathbb{S} \times \mathbb{S}]$
  let *fragment* ← $\mathbb{S}$
  let *url* := $\langle$`URL`, *protocol*, *host*, *path*, *parameters*, *fragment*$\rangle$
  let *command* := $\langle$`HREF`, *url*, $\bot, \bot\rangle$
  stop $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$

A.4. Wallets

A wallet $w \in \mathsf{Wallets}$ is an instance of the WIM's generic HTTPS web server (see Appendix E.12), modeled as an atomic DY process (I^w, Z^w, R^w, s_0^w) with the address $I^w := \mathsf{addr}(w)$. 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

$$\langle pendingDNS, pendingRequests, DNSaddress, keyMapping, tlskeys, \\ corrupt, credentials, holderKey, dcApiKeys, currentDcApiKeys, \\ dcApiReqsToWallet, dcApiRespsFromWallet, trustedVerifiers\rangle$$

with $pendingDNS$, $pendingRequests$, $DNSaddress$, $keyMapping$, $tlskeys$, $corrupt$ as described in Definition 62, $credentials \in \mathcal{T}_{\mathcal{N}}$, $holderKey \in \mathcal{N}$, $dcApiKeys \in \mathcal{T}_{\mathcal{N}}$, $currentDcApiKeys \in \mathcal{T}_{\mathcal{N}}$, $dcApiReqsToWallet \in \mathcal{T}_{\mathcal{N}}$, $dcApiRespsFromWallet \in \mathcal{T}_{\mathcal{N}}$, $trustedVerifiers \in \mathcal{T}_{\mathcal{N}}$.

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

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

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

——————————Receive a DC API request and hand it to a wallet——————————

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

——————Receive a DC API response (from an attacker wallet, see Section 2.4.4)——————

13: **else if** $\exists$ *response*, *dcApiSess*, k, f' **such that**
↪ $\langle$*response*, *dcApiSess*$\rangle \equiv m \wedge \langle$*dcApiSess*, k, $f'\rangle \in^{\langle\rangle} s'.$`currentDcApiKeys` **then**
→ 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' := $\mathsf{enc_s}(\langle$`DC_API_RESPONSE`, *response*$\rangle, k)$
16: **stop** $\langle\langle f', a, m'\rangle\rangle, s'$

———Endpoint for the attacker to add terms to the credential store of the wallet (see Section 2.2)———

17: **else if** $m \sim$ [`CRED`: $*$] **then**
18: **let** $s'.$`credentials` := $s'.$`credentials` $+^{\langle\rangle}$ m[`CRED`]
19: **stop** $\langle\rangle, s'$
20: **stop** → 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** $action \leftarrow \{$`PROCESS_DC_API_REQUEST`, `SEND_DC_API_RESPONSE`$\}$
——————————Invoke wallet app——————————
3: **if** $action \equiv$ `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` $-^{\langle\rangle} \langle protocol, request, origin, dcApiSess \rangle$
6: **if** $protocol \equiv$ `openid4vp-v1-signed` **then** → Signed authorization request
7: **let** $sig := request$[`request`]
8: **let** $authReq :=$ extractmsg(sig)
9: **let** $verifier := authReq$[`client_id`]
10: **let** $pubKey := s'.$`trustedVerifiers`[$verifier$]
11: **if** checksig($sig, pubKey$) $\not\equiv \top$ **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** $vpToken :=$ sig($presentation, s'.$`holderKey`)
20: **let** $response :=$ [`vp_token`: $vpToken$]
21: **let** $s'.$`dcApiRespsFromWallet` $:= s'.$`dcApiRespsFromWallet` $+^{\langle\rangle} \langle response, dcApiSess \rangle$
22: **stop** $\langle\rangle, s'$
——————————Forward response from wallet app via DC API——————————
23: **else if** $action \equiv$ `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` $-^{\langle\rangle} \langle dcApiSess, k, f \rangle$
27: **let** $m' :=$ $\mathsf{enc_s}(\langle$`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 $b \in \mathsf{B}$) are modeled as described in Appendix E.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 51 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 51 with the following additional constraints:

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

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

```
 1: function RUNSCRIPT(w̄, d̄, a, s′)
                ⋮
17:     let docorigin := s′.d̄.origin
18:     switch command do
19:         case ⟨DCAPI, protocol, request⟩   → Process a DC API request from a script (e.g., Algorithm 3)
20:             if docorigin.protocol ≢ S then
21:                 stop
22:             let k ← s′.dcApiKeys   → Choose key for DC API request non-deterministically
23:             let s′.dcApiKeys := s′.dcApiKeys −^⟨⟩ k   → Key is only used once, see Section 2.4.2
24:             let s′.dcApiSessions := s′.dcApiSessions +^⟨⟩ ⟨k, s′.d̄.nonce⟩
25:             let dcApiRequest := ⟨DC_API_REQUEST, protocol, request, docorigin⟩
26:             let m′ := enc_s(dcApiRequest, k)
27:             let f ← IPs
28:             stop ⟨⟨f, a, m′⟩⟩, s′
29:         case ⟨HREF, url, hrefwindow, noreferrer⟩
                ⋮
```

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

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

let $s' := s$
if $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
  let $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests}\rangle$ → Collect incoming messages
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle, s'$
if $m \equiv \texttt{TRIGGER}$ **then** → A special trigger message.
  let *switch* $\leftarrow \{\texttt{script}, \texttt{urlbar}, \texttt{reload}, \texttt{forward}, \texttt{back}\}$
  ⋮
  let $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
  stop $\langle\langle m.\texttt{result}, a, \mathit{message}\rangle\rangle, s'$
———Hand an incoming DC API response to the script that emitted the corresponding request———
else if $\exists\ k, \mathit{docnonce}, \mathit{response}$ **such that** $\hookrightarrow \langle k, \mathit{docnonce}\rangle \in s'.\texttt{dcApiSessions} \wedge \langle \texttt{DC\_API\_RESPONSE}, \mathit{response}\rangle \equiv \mathsf{dec_s}(m, k)$ **then**
  let $\overline{d}$ **such that** $\overline{d} \in \mathsf{Docs}^+(s') \wedge s'.\overline{d}.\texttt{nonce} \equiv \mathit{docnonce}$ **if possible; otherwise stop**
  let $s'.\overline{d}.\texttt{scriptstate} := \langle \texttt{DC\_API\_RESPONSE}, \mathit{response}\rangle$ → Pass response to script
  let $s'.\texttt{dcApiSessions} := s'.\texttt{dcApiSessions} -^{\langle\rangle} \langle k, \mathit{docnonce}\rangle$ → Ensure only one response can be received
  stop $\langle\rangle, s'$
stop

B. Formal Security Properties

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

Definition 9 (Verifier Logged In). Let $P = (S, E, N) \xrightarrow[v \to E_{\text{out}}]{e_{\text{in}} \to v} (S', E', N')$ be a processing step in a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\text{n}}$ with $v \in \mathsf{Verifiers}$, $id \in \mathsf{ID}$, 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) $\exists e_{\text{sch}} \in^{\langle\rangle} E_{\text{out}}, m_{\text{sch}}, k \in \mathcal{T}_{\mathcal{N}} : e_{\text{sch}} \sim \langle *, *, \mathsf{enc_s}(m_{\text{sch}}, k)\rangle$ such that $m_{\text{sch}} \in \mathsf{HTTPResponses}$

(ii) $\exists i \in \mathbb{N} : m_{\text{sch}}.\texttt{headers}.i \equiv \langle \texttt{Set-Cookie}, [\texttt{serviceTokenID}\colon \langle sTID, \top, \top, \top\rangle]\rangle$

(iii) $S'(v).\texttt{sessions}[sTID] \equiv \langle id, id.\texttt{domain}\rangle$

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

Definition 10 (Verifier Accepts Presentation). Let $P = (S, E, N) \xrightarrow[v \to E_{\text{out}}]{e_{\text{in}} \to v} (S', E', N')$ be a processing step in a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\text{n}}$ with $v \in \mathsf{Verifiers}$, $id \in \mathsf{ID}$, and $sTID, \mathit{pres}, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$. 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) $\exists e_{\text{sch}} \in^{\langle\rangle} E_{\text{out}}, m_{\text{sch}}, k \in \mathcal{T}_{\mathcal{N}} : e_{\text{sch}} \sim \langle *, *, \mathsf{enc_s}(m_{\text{sch}}, k)\rangle$ such that $m_{\text{sch}} \in \mathsf{HTTPResponses}$

(ii) $\exists i \in \mathbb{N} : m_{\text{sch}}.\texttt{headers}.i \equiv \langle \texttt{Set-Cookie}, [\texttt{serviceTokenID}\colon \langle sTID, \top, \top, \top\rangle]\rangle$

(iii) $S'(v).\texttt{sessions}[sTID] \equiv \langle id, id.\texttt{domain}\rangle$

(iv) $e_{\text{in}} \sim \langle *, *, \mathsf{enc_a}(\langle\langle \texttt{HTTPReq}, *, *, *, *, *, *, [\texttt{vp\_token}\colon \mathit{pres}]\rangle, k\rangle, \mathsf{pub}(*))\rangle$

(v) $pres \sim \mathsf{sig}(\langle \mathsf{sig}(\langle id, \mathsf{pub}(k_{\text{holder}})\rangle, \mathsf{signkey}(\mathsf{governor}(id))), *, \langle \texttt{origin}, \langle d_v, \mathsf{S}\rangle\rangle\rangle, k_{\text{holder}})$ with $d_v \in \mathsf{dom}(v)$

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

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

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

(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, $e_{\text{in}} \sim \langle *, *, \mathsf{enc_a}(\langle\langle \texttt{HTTPReq}, *, *, *, *, *, *, *, body\rangle, k_{\text{req}}\rangle, \mathsf{pub}(*))\rangle$ 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 **stop**s 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 $[\texttt{vp\_token}: *]$.

(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 **stop**s in VERIFY_PRESENTATION can be reached during P. In the following, let $pres' := body[\texttt{vp\_token}]$, 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 Lines 2ff. of Algorithm 2 not ending in **stop**, we have $pres' \sim \mathsf{sig}(\langle cred, *, \langle \texttt{origin}, \langle d_v, \mathsf{S}\rangle\rangle\rangle, k_{pres'})$ for some $cred, k_{pres'} \in \mathcal{T}_{\mathcal{N}}$ and $d_v \in \mathsf{dom}(v)$ (see also Definition 18).

(B.iii) Due to Lines 9f. of Algorithm 2 not ending in **stop** and Line 13 of Algorithm 2 not leading to the **stop** in the next Line, we also have $cred \sim \mathsf{sig}(\langle id', pubKey'\rangle, \mathsf{signkey}(\mathsf{governor}(id')))$ 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 18, we get $pres' \sim \mathsf{sig}(\langle \mathsf{sig}(\langle id', \mathsf{pub}(k_{pres'})\rangle, \mathsf{signkey}(\mathsf{governor}(id'))), *, \langle \texttt{origin}, \langle d_v, \mathsf{S}\rangle\rangle\rangle, k_{pres'})$, i.e., $\mathsf{pub}(k_{pres'}) = pubKey'$ 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 $\mathsf{acceptsPresentation}_{\rho}^{P}(v, id, sTID, pres, k_{\text{holder}})$ for some $pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$. ∎

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

(i) $e_{\text{in}} \sim \langle *, *, \texttt{TRIGGER} \rangle$

(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 $\mathsf{createsPresentation}_{\rho}^{Q}(w, pres)$.

B.1. Verifier Authentication

See Section 3.1 for an informal description of this property.

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

(1) during P, w received a trigger message, i.e, $e_{\text{in}} \sim \langle *, *, \texttt{TRIGGER} \rangle$,

(2) selected the `PROCESS_DC_API_REQUEST` action in Line 2 of Algorithm 6,

(3) chose a term $\langle protocol, request, origin, dcApiSess \rangle$ in Line 4 of Algorithm 6 such that $protocol \equiv$ `openid4vp-v1-signed`, and

(4) finished P with the stop in Line 22 of Algorithm 6,

then, with $cid := \mathsf{extractmsg}(request[\texttt{request}])[\texttt{client\_id}]$,

(I) $\texttt{client\_id} \in \mathsf{extractmsg}(request[\texttt{request}])$, $cid \in \mathsf{Doms}$, and

(II) if $v := \mathsf{dom}^{-1}(cid)$ 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 $request[\texttt{request}]$.

B.2. Wallet Authorization

See Section 3.2 for an informal description of this property.

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

(1) $\mathsf{loggedIn}^P_\rho(v, id, sTID)$ (which, by Lemma 1 and the fact that v honest in S^* implies v honest in S', implies $\mathsf{acceptsPresentation}^P_\rho(v, id, sTID, pres, k_{\text{holder}})$ for some $pres, k_{\text{holder}} \in \mathcal{T}_\mathcal{N}$), and

(2) $\exists w \in \mathsf{Wallets}$ that is honest in S^*, such that $k_{\text{holder}} \equiv s_0^w.\texttt{holderKey}$,

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

B.3. Claims Unforgeability

See Section 3.3 for an informal description of this property.

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

(1) $\mathsf{loggedIn}^P_\rho(v, id, sTID)$ for some $sTID \in \mathcal{T}_\mathcal{N}$,

(2) all $w' \in \mathsf{walletsOfId}(id)$ are honest in $S^{p'}$, and

(3) $iss := \mathsf{dom}^{-1}(id.\texttt{domain})$ is honest in $S^{p'}$,

then all of the following are true:

(I) $\exists pres, k_{\text{holder}} \in \mathcal{T}_\mathcal{N}$ such that $\mathsf{acceptsPresentation}^P_\rho(v, id, sTID, pres, k_{\text{holder}})$, and

(II) $\exists w \in \mathsf{walletsOfId}(id)$ such that $\mathsf{createsPresentation}^Q_\rho(w, pres)$ and $k_{\text{holder}} \equiv s_0^w.\texttt{holderKey}$ for some processing step Q prior to P, and

(III) if all processes nearby w are honest in S^*, i.e., all processes in $\{p \mid p \in \mathcal{W} \wedge \mathsf{nearby}(p, w)\}$, then the attacker cannot derive $sTID$ in S^*, i.e., $sTID \notin d_\emptyset(S^*(\mathsf{attacker}))$.

C. Formal Security Proof

Lemma 2 (Attacker cannot derive holder key of honest wallet). In a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\text{n}}$ we have that for every configuration (S^n, E^n, N^n) with $n \in \mathbb{N}$ in ρ and every $w \in \mathsf{Wallets}$ that is honest in S^n, it holds that $S^0(w).\texttt{holderKey}$ is not derivable by the attacker, i.e., $S^0(w).\texttt{holderKey} \notin d_\emptyset(S^n(\mathsf{attacker}))$.

Proof. There are only two lines where $S^0(w).\texttt{holderKey}$ is ever used. One is in Line 19 of Algorithm 6 where the key is used in a signing operation and the other one is Line 4 of Algorithm 27 which is only executed if w is corrupted. Since we know that w is honest, it is sufficient to only look into the first case.

In Line 19 of Algorithm 6 w signs a term with the sign function. From the equational theory defined in Definition 18 it follows that it is impossible to extract the signing key from a signature. The only two operations on signatures are $\mathsf{checksig}$ which only returns $\top$ if it is given the public key of the signing key and the other one is $\mathsf{extractmsg}$ which returns the message encapsulated in the signature but not the signing key.

Furthermore, the holder key of the wallets are initialized via the $\mathsf{signkey}$ function which is an injective mapping from processes to the set of nonces K_{sign} and this set is only used by the $\mathsf{signkey}$ function. $S^0(w).\texttt{holderKey}$ is also contained in $S^0(w).\texttt{credentials}$ wrapped in the pub function. From the equational theory defined in Definition 18, it follows that $S^0(w).\texttt{holderKey}$ cannot

be extracted from the pub function and therefore it does not leak by sending a credential from $S^0(w).\texttt{credentials}$ to the network. This means the attacker can only learn $S^0(w).\texttt{holderKey}$ by corrupting w. ∎

Lemma 3 (Attacker cannot derive signing key of honest verifier). In a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\mathrm{n}}$ we have that for every configuration (S^n, E^n, N^n) with $n \in \mathbb{N}$ in ρ and every $v \in \mathsf{Verifiers}$ that is honest in S^n, it holds that $S^0(v).\texttt{signingKey}$ is not derivable by the attacker, i.e., $S^0(v).\texttt{signingKey} \notin d_\emptyset(S^n(\mathsf{attacker}))$.

Proof. There are only two lines where $S^0(v).\texttt{signingKey}$ is ever used. One is in Line 16 of Algorithm 1 where the key is used in a signing operation and the other one is Line 4 of Algorithm 27 which is only executed if v is corrupted. Since we know that v is honest, it is sufficient to only look into the first case.

In Line 16 of Algorithm 1 v signs a term with the sign function. From the equational theory defined in Definition 18 it follows that it is impossible to extract the signing key from a signature. The only two operations on signatures are checksig which only returns $\top$ if it is given the public key of the signing key and the other one is extractmsg which returns the message encapsulated in the signature but not the signing key.

Furthermore, the signing key of the verifiers are initialized via the signkey function which is an injective mapping from processes to the set of nonces K_{sign} and this set is only used by the signkey function. Additionally, the signkey function is also used in the initial state of wallets with a verifier as the argument inside a pub function. From the equational theory defined in Definition 18, it follows that the private key cannot be extracted from the pub function. This means the attacker can only learn $S^0(v).\texttt{signingKey}$ by corrupting v. ∎

Lemma 4 (Attacker cannot derive signing key of honest issuer). In a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\mathrm{n}}$ we have that for every configuration (S^n, E^n, N^n) with $n \in \mathbb{N}$ in ρ and every $i \in \mathsf{Issuers}$ that is honest in S^n, it holds that $S^0(i).\texttt{signingKey}$ is not derivable by the attacker, i.e., $S^0(i).\texttt{signingKey} \notin d_\emptyset(S^n(\mathsf{attacker}))$.

Proof. The only line where an issuer ever reads $S^0(i).\texttt{signingKey}$ from its state is Line 4 of Algorithm 27. This line is only executed if a corrupt message has been received previously ($m \equiv \texttt{CORRUPT}$). Since we know that i is honest there is no way i leaks its signing key.

Furthermore, $S^0(i).\texttt{signingKey}$ is initialized with the injective signkey function that maps processes to the set of nonces K_{sign} and this set is only used by the signkey function. The signkey function with an issuer as the argument is also used in the initial state of verifiers (i.e., *issuers* state) and wallets (i.e., *credentials* state). In the former, it is the input to the pub function, and in the latter, it is the signing key input to the sig function. From the equational theory defined in Definition 18, it follows that the private key can neither be extracted from the pub function nor the sig function when used as the signing key. This means the attacker can only learn $S^0(i).\texttt{signingKey}$ by corrupting i. ∎

C.1. Wallet Authorization

Lemma 5 (Wallet Authorization Holds – Proof of Definition 13). We say that a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\mathrm{n}}$ *is secure w.r.t. wallet authorization* iff for every run ρ of $\mathcal{VPWS}^{\mathrm{n}}$, every configuration (S^*, E^*, N^*) in ρ, every verifier $v \in \mathsf{Verifiers}$ that is honest in S^*, every processing step $P = (S, E, N) \xrightarrow[v \to E_{\text{out}}]{e_{\text{in}} \to v} (S', E', N')$ prior to (S^*, E^*, N^*) in ρ, every identity $id \in \mathsf{ID}$, all terms $sTID \in \mathcal{T}_{\mathcal{N}}$, if we have

(1) $\mathsf{loggedIn}^P_\rho(v, id, sTID)$ (which, by Lemma 1 and the fact that v honest in S^* implies v honest in S', implies $\mathsf{acceptsPresentation}^P_\rho(v, id, sTID, pres, k_{\text{holder}})$ for some $pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$), and

(2) $\exists w \in \mathsf{Wallets}$ that is honest in S^*, such that $k_{\text{holder}} \equiv s_0^w.\texttt{holderKey}$,

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

Proof. From (1) and Lemma 1, we have $\mathsf{acceptsPresentation}^P_\rho(v, id, sTID, pres, k_{\text{holder}})$ for some $pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$. The $\mathsf{acceptsPresentation}^P_\rho$ predicate says that $pres$ is a term that is signed with k_{holder} ((v) in Definition 10), i.e., $pres \sim \mathsf{sig}(*, k_{\text{holder}})$. From (2) we know that there exists an honest wallet $w \in \mathsf{Wallets}$ such that $k_{\text{holder}} \equiv s_0^w.\texttt{holderKey}$. Since an honest wallet never changes the contents of its `holderKey` state subterm, w can indeed derive k_{holder} in all configurations in ρ up to and including S^*.

Furthermore, because $s_0^{w'}.\texttt{holderKey}$ (for any $w' \in \mathsf{Wallets}$) is initialized with the injective $\mathsf{signkey}$ function (see Definition 7 and Appendix A.1.4), we know that there exists only one such wallet in the initial state, i.e., w. We also know that this w's state contains k_{holder} only in the `holderKey` state subterm and as part of credentials (Definition 7) – but in a credential, k_{holder} always appears as $\mathsf{pub}(k_{\text{holder}})$, from which k_{holder} cannot be derived (Definition 18). The fact that w is honest allows us to apply Lemma 2 which gives us (together with the fact that the `holderKey` state subterm of an honest wallet never changes) that $s_0^w.\texttt{holderKey}$ is not derivable by the attacker.

However, only a process that can derive k_{holder} could have created $pres$, because none of the initial states of any process contains a term that matches $\mathsf{sig}(*, k_{\text{holder}})$ (see Definitions 5, 6, 7, 8, and 43).

Hence, we can conclude that w must have signed $pres$. An honest wallet such as w only uses the key stored in `holderKey` in Line 19 of Algorithm 6; hence, $pres$ must originate from there, giving us (iii) of Definition 11. Let Q' be a processing step in which w created $pres$ (there might be multiple such processing steps). Note that at least one of these steps Q' must be prior to P in ρ, otherwise, $pres$ could not have been part of a message processed by v in P (see Definition 10). In the following, we consider such a Q' that is prior to P in ρ.

Of course, during Q', Line 19 of Algorithm 6 must have been executed. Because Algorithm 6 is only ever called in Line 28 of Algorithm 27, the message processed by w in Q' must be a `TRIGGER` message, which gives us (i) of Definition 11 for Q'.

Likewise, reaching Line 19 of Algorithm 6 in Q' is only possible if w selected the `PROCESS_DC_API_REQUEST` action in Line 2 of Algorithm 6, hence, we have (ii) of Definition 11 for Q'.

This means we have proven all conditions of Definition 11 for Q'. I.e., we choose $Q = Q'$ and have $\mathsf{createsPresentation}^Q_\rho(w, pres)$ with Q prior to P in ρ. ∎

C.2. Verifier Authentication

Lemma 6 (Verifier Authentication Holds – Proof of Definition 12). We say that a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\text{n}}$ *is secure w.r.t. verifier authentication* iff for every run ρ of $\mathcal{VPWS}^{\text{n}}$, every configuration (S^*, E^*, N^*) in ρ, every wallet $w \in \mathsf{Wallets}$ that is honest in S^*, every processing step $P = (S, E, N) \xrightarrow[w \to E_{\text{out}}]{e_{\text{in}} \to w} (S', E', N')$ prior to (S^*, E^*, N^*) in ρ, if

(1) during P, w received a trigger message, i.e, $e_{\text{in}} \sim \langle *, *, \texttt{TRIGGER} \rangle$,

(2) selected the `PROCESS_DC_API_REQUEST` action in Line 2 of Algorithm 6,

(3) chose a term $\langle protocol, request, origin, dcApiSess \rangle$ in Line 4 of Algorithm 6 such that $protocol \equiv$ `openid4vp-v1-signed`, and

(4) finished P with the stop in Line 22 of Algorithm 6,

then, with $cid := \mathsf{extractmsg}(request[\texttt{request}])[\texttt{client\_id}]$,

(I) $\texttt{client\_id} \in \mathsf{extractmsg}(request[\texttt{request}])$, $cid \in \mathsf{Doms}$, and

(II) if $v := \mathsf{dom}^{-1}(cid)$ 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 $request[\texttt{request}]$.

Proof. Due to (1) to (4), we have that w must have executed Lines 7ff. of Algorithm 6 during P. In particular, the check in Line 11 of Algorithm 6 must have been successful (due to (4)). This means we have $\mathsf{checksig}(request[\texttt{request}], pubKey) \equiv \top$ with $pubKey \equiv S(w).\texttt{trustedVerifiers}[cid]$ (Line 10 of Algorithm 6) and $cid \equiv \mathsf{extractmsg}(request[\texttt{request}])[\texttt{client\_id}]$ (Line 9 of Algorithm 6).

Because of the successful check in Line 11 of Algorithm 6 and the equational theory (Definition 18) for signature verification, we know that $pubKey \sim \mathsf{pub}(*)$. In particular, $pubKey$ cannot be $\langle\rangle$ (see Definition 26), hence, $cid \in S(w).\texttt{trustedVerifiers}$.

Since w as an honest wallet never changes its `trustedVerifiers` state subterm, we have that $S(w).\texttt{trustedVerifiers}[cid] \equiv S^0(w).\texttt{trustedVerifiers}[cid]$. From the initialization (see Definition 7) of $S^0(w).\texttt{trustedVerifiers}$ we know that every entry in `trustedVerifiers` is a pair $\langle d, \mathsf{pub}(\mathsf{signkey}(v))\rangle$ with $d \in \mathsf{dom}(v)$ and $v \in \mathsf{Verifiers}$. This implies two things: First of all, because $\langle\rangle \notin S^0(w).\texttt{trustedVerifiers}$ and $pubKey \not\equiv \langle\rangle$, we have $cid \not\equiv \langle\rangle$, which, due to Definition 26, implies $\texttt{client\_id} \in \mathsf{extractmsg}(request[\texttt{request}])$. Second, due to $cid \in S(w).\texttt{trustedVerifiers}$, we also have $cid \in \mathsf{Doms}$. Hence, we get condition (I).

Due to how $S^0(w).\texttt{trustedVerifiers}$ is initialized and the successful check in Line 11 of Algorithm 6, there must be a process $v \in \mathsf{Verifiers}$, such that $v \in \mathsf{Verifiers}$, $d \in \mathsf{dom}(v)$, and $pubKey \equiv \mathsf{pub}(\mathsf{signkey}(v))$. Because condition (II) considers an honest v, we can apply Lemma 3 for $\mathsf{signkey}(v)$, because Definition 6 tells us that $S^0(v).\texttt{signingKey} \equiv \mathsf{signkey}(v)$. Hence, we have that only v can derive $\mathsf{signkey}(v)$ in S (and all prior processing steps).

Due to $pubKey \equiv \mathsf{pub}(\mathsf{signkey}(v))$, and because none of the initial states of any process contains a term that matches $\mathsf{sig}(t, \mathsf{signkey}(v))$ where $t[\texttt{client\_id}] \equiv cid$ (see Definitions 5, 6, 7, 8, and 43), the term $request[\texttt{request}]$ must have been created (and emitted) by v in some processing step Q' prior to P in ρ (there might be multiple such processing steps). Now, the only place in which a verifier uses the key stored in $S^0(v).\texttt{signingKey}$ is in Line 16 of Algorithm 1, where a signature is created.

Hence, $request[\texttt{request}]$ must originate from that line, or, in other words: If we set $Q = Q'$ for one of the Q' from above, 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 that signature operation was $request[\texttt{request}]$. Hence, we have condition (II). ∎

C.3. Claims Unforgeability

Definition 15 (Wallet Receives DC API Request). Let $K = (S, E, N) \xrightarrow[w \to E_{\text{out}}]{e_{\text{in}} \to w} (S', E', N')$ be a processing step in a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\text{n}}$ with $w \in \mathsf{Wallets}$, $origin \in \mathcal{T}_{\mathcal{N}}$, and $k \in \mathcal{N}$. We say that w received an DC API request in processing step K of ρ, if all of the following is true:

(i) $e_{\text{in}} \sim \mathsf{enc_s}(\langle \texttt{DC\_API\_REQUEST}, *, *, origin\rangle, k)$

(ii) $k \in S^0(w).\texttt{dcApiKeys}$

(iii) in K, w chooses $\top$ for *selectHonestWallet* in Line 6 of Algorithm 5

We then write $\mathsf{receiveDcApiRequest}_{\rho}^{K}(w, \mathit{origin}, k)$.

Lemma 7 (DC API Request precedes DC API Response). In a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\mathrm{n}}$ we have that for every $w \in \mathsf{Wallets}$, every nonce $\mathit{dcApiSess} \in \mathcal{N}$, every term $\mathit{pres} \in \mathcal{T}_{\mathcal{N}}$, and every processing step $R = (S, E, N) \to (S', E', N')$ in which w terminates in Line 28 of Algorithm 6 and chooses $\langle[\texttt{vp\_token} : \mathit{pres}], \mathit{dcApiSess}\rangle \in^{\langle\rangle} S(w).\texttt{dcApiRespsFromWallet}$ in Line 24 of Algorithm 6, if w is honest in S', then there exists a processing step K in ρ prior to R such that $\mathsf{receiveDcApiRequest}_{\rho}^{K}(w, \mathit{origin}, k)$ with $\mathit{origin} := \mathsf{extractmsg}(\mathit{pres}).3.2$ and $k \in \mathcal{T}_{\mathcal{N}}$ being the first value chosen in Line 25 of Algorithm 6 in processing step R. Furthermore, when executing Line 4 of Algorithm 5 in K, w chooses the value $\mathit{dcApiSess}$.

Proof. We now trace the path that $\mathit{dcApiSess}$ takes.

(A) From the preconditions, we know that $\langle[\texttt{vp\_token} : \mathit{pres}], \mathit{dcApiSess}\rangle$ is stored in $S(w).\texttt{dcApiRespsFromWallet}$. Furthermore, $S(w).\texttt{currentDcApiKeys}$ contains the sequence $\langle \mathit{dcApiSess}, k, f\rangle$, for some value $f \in \mathcal{T}_{\mathcal{N}}$.

(B) Initially, the `currentDcApiKeys` state subterm is empty (see Definition 7). The only location in which the wallet adds values to this state subterm is Line 5 of Algorithm 5, with the first value being a freshly chosen nonce (see Line 4 of Algorithm 5). Therefore, there exists exactly one processing step N in which w chooses $\mathit{dcApiSess}$ in Line 4 of Algorithm 5 and stores the sequence $\langle \mathit{dcApiSess}, k, f\rangle$ into the `currentDcApiKeys` state subterm. Note that the key k is determined in Line 2 of Algorithm 5.

(C) Similarly, the `dcApiRespsFromWallet` state subterms of wallets are initially also empty (see Definition 7). Thus, the sequence was added to the state subterm by w in some processing step Q. The wallet only adds values to the `dcApiRespsFromWallet` subterm in Line 21 of Algorithm 6.

(D) In Q, the wallet takes both the $\mathit{dcApiSess}$ value and the audience of the presentation, i.e., $\mathsf{extractmsg}(\mathit{pres}).3.2$, from the `dcApiReqsToWallet` state subterm in Line 4 of Algorithm 6.

(E) The `dcApiReqsToWallet` state subterm is initially empty (see Definition 7). Thus, there is some processing step $K' = (S'', E'', N'') \xrightarrow[w \to E_{\mathrm{out}}]{e_{\mathrm{in}} \to w} (S''', E''', N''')$ prior to Q in which w executes Line 9 of Algorithm 5, as this is the only location where a wallet adds values to this state subterm. Note that the *origin* value stored into the `dcApiReqsToWallet` entry is determined in Line 2 of Algorithm 5.

(F) $K' = N$, as the value $\mathit{dcApiSess}$ is the same nonce chosen in Line 4 of Algorithm 5.

(G) Hence, to reach Line 9 of Algorithm 5 in K', w must have:

(G.i) Chosen $\top$ for *selectHonestWallet* in Line 6 of Algorithm 5.

(G.ii) Chosen $\mathit{dcApiSess}$ in Line 4 of Algorithm 5.

(G.iii) Passed $m \equiv \mathsf{enc_s}(\langle\texttt{DC\_API\_REQUEST}, \mathit{protocol}', \mathit{request}', \mathit{origin}\rangle, k)$ to PROCESS_OTHER with $\mathit{protocol}' \in \mathbb{S}$ and $\mathit{request}' \in \mathcal{T}_{\mathcal{N}}$. Note that the *origin* and key k are the same values as in the statement of the lemma. Furthermore, $k \in S''(w).\texttt{dcApiKeys}$, see the condition in Line 2 of Algorithm 5.

(G.iv) To call PROCESS_OTHER with m as the first argument in Line 30 of Algorithm 27 w must have received $e_{\mathrm{in}} \sim \mathsf{enc_s}(\langle\texttt{DC\_API\_REQUEST}, \mathit{protocol}', \mathit{request}', \mathit{origin}\rangle, k)$.

Since terms in the `dcApiKeys` state subterm are only read (Line 2 of Algorithm 5) or deleted (Line 3 of Algorithm 5) we know that when reading a term it is one that was initialized in $S^0(w).\texttt{dcApiKeys}$. Now, with K set to K', *protocol* set to *protocol'*, and *request* set to *request'*, we have $\mathsf{receiveDcApiRequest}_\rho^K(w, origin, k)$. ∎

Lemma 8 (Browser send DC API Request). For every run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\mathrm{n}}$, every $w \in \mathsf{Wallets}$ that is honest in S^n, every term $k \in^{\langle\rangle} S^0(w).\texttt{dcApiKeys}$, if we have that

(1) every $p \in \mathcal{W}$ with $\mathsf{nearby}(p, w) \equiv \top$ is honest in S^n and

(2) there exists a processing step $K = (S, E, N) \xrightarrow[w \to E_{\text{out}}]{e_{\text{in}} \to w} (S', E', N')$ in ρ such that $\mathsf{receiveDcApiRequest}_\rho^K(w, origin, k)$ with $origin \in \mathcal{T}_{\mathcal{N}}$,

then it holds that k is only derivable by w and exactly one $b \in \mathsf{B}$ which is honest in S^n, i.e., k is not derivable by the attacker ($k \notin d_\emptyset(S^n(\mathsf{attacker}))$).

Proof. From Definition 7 we know that $S^0(w).\texttt{dcApiKeys} \equiv \left\langle \bigcup_{p \in \mathcal{W}} \mathsf{dcApiKeys}(w, p) \right\rangle$. The $\mathsf{dcApiKeys}$ function maps two processes to disjoint subsets of K_{dcapi} (see Appendix A.1.4). Nonces in this set are only distributed via the $\mathsf{dcApiKeys}$ mapping, which is only used when initializing states. We now consider the two processes that initially obtain the value k:

(A) **Wallet w:** Initially, regarding the wallet process w, the key k is only stored in the `dcApiKeys` state subterm. The wallet process accesses this state subterm only when deleting values in Line 3 of Algorithm 5 and obtaining a key in Line 2 of Algorithm 5. The former does not reveal any DC API keys but the later one does.

Tracing k after executing Line 2 of Algorithm 5 in some processing step, we can see that k is removed from the `dcApiKeys` state subterm and stored in the `currentDcApiKeys` state subterm in Line 5 of Algorithm 5, as part of a sequence.

A wallet accesses the `currentDcApiKeys` state subterm only in Lines 13ff. of Algorithm 5 and Lines 25ff. of Algorithm 6 (besides Line 5 of Algorithm 5). In both cases, the key is removed from `currentDcApiKeys` (see Line 14 of Algorithm 5 and Line 26 of Algorithm 6), but used (only) as a symmetric encryption key (see Line 15 of Algorithm 5 and Line 27 of Algorithm 6). Due to the definition of the equational theory (Definition 18) we know that there is no function to extract the key from a ciphertext.

(B) **Process $p \in \mathcal{W}$:** Looking into the initial state of $\mathsf{Issuers}$ (Definition 5), $\mathsf{Verifiers}$ (Definition 6), $\mathsf{Wallets}$ (Definition 7), and B (Definition 8) we can see that only $\mathsf{Wallets}$ and B have keys initialized with $\mathsf{dcApiKeys}$. Due to the disjointness of the $\mathsf{dcApiKeys}$ function and the definition of states in a $\mathcal{VPWS}^{\mathrm{n}}$ we know that k is either in the initial state of at most two wallet processes, or in one initial state of a browser and a wallet.

Taking into account the second pre-condition, we know that w receives the term $\langle$`DC_API_REQUEST`, *protocol*, *request*, *origin*$\rangle$ encrypted with k. Since a wallet never encrypts such a term with a key from its `dcApiKeys` state subterm, we know that the request was created by some $b \in \mathsf{B}$ by executing Lines 25ff. of Algorithm 7 in some processing step prior to K.

Due to the fact that the $\mathsf{dcApiKeys}$ function does only distribute the same key to two processes that are near each other we have $\mathsf{nearby}(w, b)$. Using the pre-condition (1) gives us that b must be honest in S^n.

We now have to prove that b does not leak k to the attacker. From Definition 8 we know that only $S^0(b).\texttt{dcApiKeys} \equiv \left\langle \bigcup_{p \in \mathcal{W}} \mathsf{dcApiKeys}(b, p) \right\rangle$ contains keys distributed by $\mathsf{dcApiKeys}$ in

the initial state of a browser. Looking into Algorithm 7 and Algorithm 8 gives us that the state *dcApiKeys* is only ever used in Lines 22f. of Algorithm 7. These lines must have been executed in some processing step $J = (S'', E'', N'') \xrightarrow[b \to E_{\text{out}}]{e_{\text{in}} \to b} (S''', E''', N''')$ in ρ prior to K to create $\mathsf{enc_s}(\langle \texttt{DC\_API\_REQUEST}, \mathit{protocol}, \mathit{request}, \mathit{origin} \rangle, k)$ in Line 25 of Algorithm 7. In Line 22 of Algorithm 7, b chooses k, and in Line 23 of Algorithm 7, k is removed from $S'''(b).\texttt{dcApiKeys}$. That means that every key stored in $S^0(b).\texttt{dcApiKeys}$ is only used once and there is never something added to the `dcApiKeys` state subterm, but only removed. Tracing the usage of k after Line 23 of Algorithm 7 shows that k is stored in $S'''(b).\texttt{dcApiSessions}$ in Line 24 of Algorithm 7 and used to symmetrically encrypt the DC API request in Line 26 of Algorithm 7, i.e., $\mathsf{enc_s}(*, k)$. Due to the definition of the equational theory (Definition 18) we know that there is no function to extract the key from a ciphertext.

We now have to trace what happens to the key stored in $S'''(b).\texttt{dcApiSessions}$. The *dcApiSessions* state is used in three lines: In Line 24 of Algorithm 7, new sessions are added to the state. This means nothing leaks from the state. In Line 76 of Algorithm 8, a session is retrieved from the state. k contained in the session is only used to decrypt a term in this line. The last line using the *dcApiSessions* state is Line 79 of Algorithm 8. In this line the session containing k is removed from the state. Since every $k \in S^0(b).\texttt{dcApiKeys}$ is only once added to *dcApiSessions* there are no occurences of k in *dcApiSessions* anymore. Due to the fact that k is never contained in any other part of the state and b is honest (otherwise states could leak in Line 4 of Algorithm 27), we conclude that k cannot leak to the attacker. ∎

Lemma 9 (Verifier Script is not Changed in the Browser). In a run ρ of a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\text{n}}$ we have that for every configuration (S^*, E^*, N^*) in ρ and every $b \in \mathsf{B}$ that is honest in S^*, every $\mathit{docnonce} \in \mathcal{N}$ and there exists $\bar{d} \in \mathsf{Docs}^+(S^*(b))$ with $S^*(b).\bar{d}.\texttt{nonce} \equiv \mathit{docnonce}$, every configuration (S', E', N') prior to (S^*, E^*, N^*) in which there exists $\bar{d}' \in \mathsf{Docs}^+(S'(b))$ with $S'(b).\bar{d}.\texttt{nonce} \equiv \mathit{docnonce}$, and the following is true:

(1) $S'(b).\bar{d}'.\texttt{script} \equiv \texttt{script\_verifier\_authentication}$

(2) the process $v := \mathsf{dom}^{-1}(S^*(b).\bar{d}.\texttt{location.host})$ is honest in S^*

(3) $S'(b).\bar{d}'.\texttt{location.protocol} \equiv \texttt{S}$

it holds that

(I) $S^*(b).\bar{d}.\texttt{location} \equiv S'(b).\bar{d}'.\texttt{location}$

(II) $S^*(b).\bar{d}.\texttt{headers} \equiv S'(b).\bar{d}'.\texttt{headers}$

(III) $S^*(b).\bar{d}.\texttt{referrer} \equiv S'(b).\bar{d}'.\texttt{referrer}$

(IV) $S^*(b).\bar{d}.\texttt{script} \equiv S'(b).\bar{d}'.\texttt{script}$

Proof. We now consider any possible location in which the browser modifies documents stored in its state and show that, with the given preconditions, the `location`, `headers`, `referrer`, and the `script` values of $S'(b).\bar{d}'$ are never modified.

(A) Line 78 of Algorithm 8 changes $S'(b).\bar{d}'.\texttt{scriptstate}$.

(B) Line 3 of Algorithm 13 changes $S'(b).\bar{d}'.\texttt{active}$.

(C) Line 4 of Algorithm 13 changes $S'(b).\bar{d}'.\texttt{active}$.

(D) Line 3 of Algorithm 14 changes $S'(b).\bar{d}'.\texttt{active}$.

(E) Line 4 of Algorithm 14 changes $S'(b).\bar{d}'.\texttt{active}$.

(F) Line 42 of Algorithm 16 changes $S'(b).\bar{d}'.\texttt{active}$.

(G) Line 14 of Algorithm 15 changes $S'(b).\bar{d}'.\texttt{scriptstate}$.

(H) Line 34 of Algorithm 15 changes $S'(b).\bar{d}'.\texttt{subwindows}$ by adding a window. However, this window does not contain any documents.

(I) Line 54 of Algorithm 15 changes $S'(b).\bar{d}'.\texttt{script}$.
From the precondition we know that the process $v := \mathsf{dom}^{-1}(S^*(b).\bar{d}.\texttt{location.host})$ is honest in S^*, $S'(b).\bar{d}'.\texttt{location.protocol} \equiv \texttt{S}$, and $S'(b).\bar{d}'.\texttt{script} \equiv \texttt{script\_verifier\_authentication}$. Since $S'(b).\bar{d}'.\texttt{location}$ is never changed we know that it will be the same in the future, i.e., $S'(b).\bar{d}'.\texttt{location} \equiv S^*(b).\bar{d}.\texttt{location}$.

Looking into the code where a SETSCRIPT command is executed we can see that the script in the active document of window $\bar{w}'$ is modified (Line 54 of Algorithm 15) and $\bar{w}'$ is selected with the GETWINDOW function in Line 53 of Algorithm 15. The GETWINDOW function has the current window $\bar{w}$ as an argument and does only return a different window if its active document runs under the same origin, i.e., $S'(b).\bar{w}'.\texttt{activedocument.origin} \equiv S'(b).\bar{w}.\texttt{activedocument.origin}$ (Line 3 of Algorithm 10).

Due to the fact that $S^*(b).\bar{d}.\texttt{location}$ is an HTTPS origin and v is honest in S^* only a script from v can change the script in $S^*(b).\bar{d}.\texttt{script}$. Looking into the definition of a $\mathcal{VPWS}^{\mathrm{n}}$ we can see that the only honest party returning the script *script_verifier_authentication* is a verifier. Further looking into the definition of a verifier we can see that only Line 2 of Algorithm 1 returns the script *script_verifier_index* and Line 21 of Algorithm 1 returns the script *script_verifier_authentication*.

From the relation of *script_verifier_index* and *script_verifier_authentication* we can see that they do not terminate with a SETSCRIPT command. This means that $S'(b).\bar{d}'.\texttt{script}$ never changes and equals $S^*(b).\bar{d}.\texttt{script}$.

(J) Line 58 of Algorithm 15 changes $S'(b).\bar{d}'.\texttt{scriptstate}$.

(K) Line 86 of Algorithm 15 changes $S'(b).\bar{d}'.\texttt{scriptinputs}$.

(L) In Lines 39 and 44 of Algorithm 16, a new document is created and added to some window. However, this document has a freshly chosen document nonce, see Line 37 of Algorithm 16.

(M) Line 49 of Algorithm 16 changes $S'(b).\bar{d}'.\texttt{scriptinputs}$. ∎

Lemma 10 (Claims Unforgeability Holds - Proof of Definition 14). We say that a Verifiable Presentations Web System with Network Attacker $\mathcal{VPWS}^{\mathrm{n}}$ *is secure w.r.t. claims unforgeability* iff for every run ρ of $\mathcal{VPWS}^{\mathrm{n}}$, every configuration (S^*, E^*, N^*) in ρ, every verifier $v \in \mathsf{Verifiers}$ that is honest in S^*, every processing step $P = (S^p, E^p, N^p) \xrightarrow[v \to E_{\mathrm{out}}]{e_{\mathrm{in}} \to v} (S^{p'}, E^{p'}, N^{p'})$ prior to (S^*, E^*, N^*) in ρ, every identity $id \in \mathsf{ID}$, if we have all of the following:

(1) $\mathsf{loggedIn}^P_\rho(v, id, sTID)$ for some $sTID \in \mathcal{T}_{\mathcal{N}}$,

(2) all $w' \in \mathsf{walletsOfId}(id)$ are honest in $S^{p'}$, and

(3) $iss := \mathsf{dom}^{-1}(id.\texttt{domain})$ is honest in $S^{p'}$,

then all of the following are true:

(I) $\exists pres, k_{\text{holder}} \in \mathcal{T}_{\mathcal{N}}$ such that $\mathsf{acceptsPresentation}^{P}_{\rho}(v, id, sTID, pres, k_{\text{holder}})$, and

(II) $\exists w \in \mathsf{walletsOfId}(id)$ such that $\mathsf{createsPresentation}^{Q}_{\rho}(w, pres)$ and $k_{\text{holder}} \equiv s_0^w.\texttt{holderKey}$ for some processing step Q prior to P, and

(III) if all processes nearby w are honest in S^*, i.e., all processes in $\{p \mid p \in \mathcal{W} \wedge \mathsf{nearby}(p, w)\}$, then the attacker cannot derive $sTID$ in S^*, i.e., $sTID \notin d_{\emptyset}(S^*(\mathsf{attacker}))$.

Proof. (A) **Postcondition (I):** Follows immediately from Lemma 1 and the fact that v honest in S^* implies v honest in $S^{p'}$.

(B) **Postcondition (II):**

We prove this property by starting from the predicate $\mathsf{acceptsPresentation}^{P}_{\rho}(v, id, sTID, pres, k_{\text{holder}})$ (see postcondition (I)), and show that, provided the issuer is honest and all $w' \in \mathsf{walletsOfId}(id)$ are honest, there exists exactly one wallet $w \in \mathsf{walletsOfId}(id)$ that could have generated $pres$.

From (I) we know that $\mathsf{checksig}(\mathsf{sig}(\langle id, \mathsf{pub}(k_{\text{holder}})\rangle, \mathsf{signkey}(\mathsf{governor}(id))), S^p(v).\texttt{issuers}[id.\texttt{domain}]) \equiv \top$ in Line 13 of Algorithm 2. The definition of $\mathsf{governor}$ and (3) gives us $iss \equiv \mathsf{dom}^{-1}(id.\texttt{domain}) \equiv \mathsf{governor}(id)$ is honest in $S^{p'}$. The term $\mathsf{sig}(\langle id, \mathsf{pub}(k_{\text{holder}})\rangle, \mathsf{signkey}(\mathsf{governor}(id)))$ can only be created by a process that can derive $\mathsf{signkey}(\mathsf{governor}(id)) \equiv \mathsf{signkey}(iss)$. Since iss is honest in $S^{p'}$ we can apply Lemma 4 which says that the attacker cannot derive $\mathsf{signkey}(iss)$. Furthermore, due to the injectivity of the $\mathsf{signkey}$ function there is no other process than iss who can derive $\mathsf{signkey}(iss)$.

Looking into the definition of issuers we can see that an honest issuer never creates any signatures. Only the initial state of wallets contain terms signed with $\mathsf{signkey}(iss)$. The set of all terms signed with $\mathsf{signkey}(iss)$ and containing id follows from the initialization of wallets and is:

$$C_{iss}^{id} := \{\mathsf{sig}(\langle id, \mathsf{pub}(S^0(w').\texttt{holderKey})\rangle, \mathsf{signkey}(iss)) | w' \in \mathsf{walletsOfId}(id)\}$$

We can now draw the following conclusions:

$$\begin{aligned}
pres &\sim \mathsf{sig}(\langle \mathsf{sig}(\langle id, \mathsf{pub}(k_{\text{holder}})\rangle, \mathsf{signkey}(\mathsf{governor}(id))), *, *\rangle, k_{\text{holder}}) && (1)\\
&\sim \mathsf{sig}(\langle \mathsf{sig}(\langle id, \mathsf{pub}(k_{\text{holder}})\rangle, \mathsf{signkey}(iss)), *, *\rangle, k_{\text{holder}}) && (2)\\
&\sim \mathsf{sig}(\langle c, *, *\rangle, k_{\text{holder}}) \text{ with } c \in C_{iss}^{id} && (3)\\
&\sim \mathsf{sig}(\langle \mathsf{sig}(\langle id, \mathsf{pub}(S^0(w').\texttt{holderKey})\rangle, \mathsf{signkey}(iss)), *, *\rangle, k_{\text{holder}}) && (4)\\
&\sim \mathsf{sig}(\langle \mathsf{sig}(\langle id, \mathsf{pub}(S^0(w').\texttt{holderKey})\rangle, \mathsf{signkey}(iss)), *, *\rangle, S^0(w').\texttt{holderKey}) && (5)
\end{aligned}$$

(1) Follows directly from $\mathsf{acceptsPresentation}^{P}_{\rho}(v, id, sTID, pres, k_{\text{holder}})$.

(2) Follows from $iss \equiv \mathsf{dom}^{-1}(id.\texttt{domain}) \equiv \mathsf{governor}(id)$.

(3) Follows from the fact that the inner signature is signed with $\mathsf{signkey}(iss)$ and from above only a term in C_{iss}^{id} is signed with this key and contains id.

(4) Replace c by its definition with some $w' \in \mathsf{walletsOfId}(id)$

(5) Follows from Line 11 of Algorithm 2 because there a verifier uses $\mathsf{pub}(S^0(w').\texttt{holderKey})$ to check the signature of $pres$. This means we have $\mathsf{pub}(S^0(w').\texttt{holderKey}) \equiv \mathsf{pub}(k_{\text{holder}})$ and due to the equational theory defined in Definition 18 we can conclude that $S^0(w').\texttt{holderKey} \equiv k_{\text{holder}}$.

The term $\mathsf{sig}(\langle \mathsf{sig}(\langle id, \mathsf{pub}(S^0(w').\texttt{holderKey})\rangle, \mathsf{signkey}(iss)), *, *\rangle, S^0(w').\texttt{holderKey})$ could have only been created by someone who can derive $S^0(w').\texttt{holderKey}$. From the initialization of wallets we know that $S^0(w').\texttt{holderKey} \equiv \mathsf{signkey}(w')$. Since the $\mathsf{signkey}$ function is an injective mapping every wallet has their unique *holderKey*. This means there exists only one honest wallet $w' \equiv w$ that can sign the above presentation. From (2) we know that all $w' \in \mathsf{walletsOfId}(id)$ are honest in S' which lets us apply Lemma 2 that gives us that $S^0(w).\texttt{holderKey}$ is not derivable by the attacker.

We know that the honest wallet w created *pres*. This must have append in a processing step $Q = (S^q, E^q, N^q) \xrightarrow[w \to E_{\text{out}}]{e_{\text{in}} \to w} (S^{q'}, E^{q'}, N^{q'})$ prior to P. From looking into the definition of wallets we can see that only in Line 19 of Algorithm 6 a wallet signs a term such as *pres* with $S(w).\texttt{holderKey}$ and since a wallet never changes its *holderKey* we have that $S(w).\texttt{holderKey} \equiv S^0(w).\texttt{holderKey}$. To reach this line `PROCESS_DC_API_REQUEST` must have been chosen for *action* in Line 2 of Algorithm 6 and before that the wallet process must have received a trigger message (i.e. $e_{\text{in}} \sim \langle *, *, \texttt{TRIGGER}\rangle$).

(C) **Postcondition (III):**

We prove this property by starting from the predicate $\mathsf{createsPresentation}^Q_\rho(w, pres)$ (see postcondition (II)) and tracing the path that *pres* takes to reach the verifier. *pres* must reach the v because we have $\mathsf{acceptsPresentation}^P_\rho(v, id, sTID, pres, k_{\text{holder}})$ as a precondition (see postcondition (I)). To do so, we assume as a precondition that all processes in the vicinity of w are honest, i.e., all $p \in \mathcal{W}$ such that $\mathsf{nearby}(p, w)$ holds are honest (see postcondition (III)).

Due to $\mathsf{createsPresentation}^Q_\rho(w, pres)$ we know that *pres* is created in Line 19 of Algorithm 6. Tracing the path of *pres* we can see that it is only stored in the form $\langle[\texttt{vp\_token}, pres], *\rangle$ in $S^{q'}(w).\texttt{dcApiRespsFromWallet}$ in the processing step $Q = (S^q, E^q, N^q) \xrightarrow[w \to E_{\text{out}}]{e_{\text{in}} \to w} (S^{q'}, E^{q'}, N^{q'})$ and not send out to the network in Line 22 of Algorithm 6.

The only line where w reads the state *dcApiRespsFromWallet* is Line 24 of Algorithm 6. w will not read *dcApiRespsFromWallet* in Line 4 of Algorithm 27 because we know that it is honest at least until $S^{p'}$ of processing step $P = (S^p, E^p, N^p) \xrightarrow[v \to E_{\text{out}}]{e_{\text{in}} \to v} (S^{p'}, E^{p'}, N^{p'})$. Since we know that in the processing step P v receives *pres* we can conclude that there exists at least one processing step $R = (S^r, E^r, N^r) \xrightarrow[w \to E_{\text{out}}]{e_{\text{in}} \to w} (S^{r'}, E^{r'}, N^{r'})$ after Q and prior to P in which w:

(C.i) selects $\langle[\texttt{vp\_token}, pres], dcApiSess\rangle \in S^r(w).\texttt{dcApiRespsFromWallet}$ in Line 24 of Algorithm 6 with $dcApiSess \in \mathcal{N}$ and

(C.ii) terminates in Line 28 of Algorithm 6 with $E_{\text{out}} \sim \langle\langle *, *, \mathsf{enc_s}(\langle \texttt{DC\_API\_RESPONSE}, [\texttt{vp\_token}, pres]\rangle, k)\rangle\rangle$ with $k \in \mathcal{N}$.

The key k is chosen based on *dcApiSess* from $\langle dcApiSess, k, *\rangle \in S^r(w).\texttt{dcApiRespsFromWallet}$ in Line 25 of Algorithm 6. Due to Lemma 7 we have $\mathsf{receiveDcApiRequest}^K_\rho(w, origin, k)$ in some processing step $K = (S^k, E^k, N^k) \xrightarrow[w \to E_{\text{out}}]{e_{\text{in}} \to w} (S^{k'}, E^{k'}, N^{k'})$ prior to Q with $origin = \mathsf{extractmsg}(pres).3.2$. Furthermore, in K, w chooses the value *dcApiSess* in Line 4 of Algorithm 5.

The postcondition of Lemma 7 together with all processes nearby w being honest in $S^{p'}$ (III) lets us apply Lemma 8 that gives us that only w and some honest $b \in \mathsf{B}$ can derive k in $S^{p'}$. Hence, only w and b can decrypt the term $m' := \mathsf{enc_s}(\langle \texttt{DC\_API\_RESPONSE}, [\texttt{vp\_token}, pres]\rangle, k)$ (at least until $S^{p'}$). Due to the fact that w does not process a term of the form m' and v receives *pres* in P there has to be at least one processing step in b that processes m'.

From $\mathsf{acceptsPresentation}^{P}_{\rho}(v, id, sTID, pres, k_{\text{holder}})$ (I) we know that $pres \equiv \mathsf{sig}(\langle \mathsf{sig}(\langle id, \mathsf{pub}(k_{\text{holder}})\rangle, *), *, \langle \texttt{origin}, origin\rangle\rangle, k_{\text{holder}})$ with $origin.\texttt{host} \in \mathsf{dom}(v)$.

The only line where b creates a DC API request of the form $\mathsf{enc_s}(\langle \texttt{DC\_API\_REQUEST}, *, *, origin\rangle, k)$ is Line 26 of Algorithm 7. This line is only reached if b executed a script that used the $\langle \texttt{DCAPI}, *, *\rangle$ command (Line 19 of Algorithm 7). From Line 25 of Algorithm 7 we can see that the *origin* is the origin under which the script is running.

Due to the fact that:

(C.i) $origin.\texttt{host} \in \mathsf{dom}(v)$

(C.ii) b is honest in S^*

(C.iii) the DC API script command can only be executed by an HTTPS origin (Line 20 of Algorithm 7)

we know that only a script running under an HTTPS origin of v (i.e., $\langle d_v, \texttt{S}\rangle$ with $d_v \in \mathsf{dom}(v)$) can send the DC API request.

To send the DC API request in some processing step $J = (S^j, E^j, N^j) \xrightarrow[b \to E_{\text{out}}]{e_{\text{in}} \to b} (S^{j'}, E^{j'}, N^{j'})$ b must execute some script of v and terminate in Line 28 of Algorithm 7. In this processing step, b also writes the document nonce ($S(b).\bar{d}.nonce$) of the document in which the script is executed together with the key $k \in S^j(b).\texttt{dcApiKeys}$ into $S^{j'}(b).\texttt{dcApiSessions}$ in Line 24 of Algorithm 7. Looking into the definition of a verifier we can see that only the script *script_verifier_authentication* terminates with a DC API script command created in Line 6 of Algorithm 3.

From above we know that the DC API response m' must be decrypted by b. The only line where a browser decrypts a term of the form $\mathsf{enc_s}(\langle \texttt{DC\_API\_RESPONSE}, [\texttt{vp\_token}, pres]\rangle, k)$ with a key that is originally retrieved from its `dcApiKeys` state subterm is Line 76 of Algorithm 8. Let $T = (S^t, E^t, N^t) \xrightarrow[b \to E_{\text{out}}]{e_{\text{in}} \to b} (S^{t'}, E^{t'}, N^{t'})$ be a processing step in which b decrypts the DC API response.

Retrieving some $\bar{d}'$ in Line 77 of Algorithm 8 is successful because of $\mathsf{acceptsPresentation}^{P}_{\rho}(v, id, sTID, pres, k_{\text{holder}})$ which means that *pres* does have to be process further which only happens after this line because b is honest in S^*. This means that b terminates in the processing step T in Line 80 of Algorithm 8.

Due to the fact that k and *docnonce* are stored together in Line 24 of Algorithm 7, retrieved together in Line 76 of Algorithm 8, and that sequences stored in the `dcApiSessions` state subterm are never modified we know that $S^j(b).\bar{d}.\texttt{nonce}$ equals $S^t(b).\bar{d}'.\texttt{nonce}$, as each key in `dcApiKeys` is used by the browser model at most once, see Line 23 of Algorithm 7. Now we need to prove that the script and the location of the documents $S^j(b).\bar{d}$ and $S^t(b).\bar{d}'$ are the same. From above we know that $S^j(b).\bar{d}.\texttt{script} \equiv \texttt{script\_verifier\_authentication}$, from the precondition we know that v is honest in T, and that $S^j(b).\bar{d}.\texttt{location.protocol} \equiv \texttt{S}$. This lets us apply Lemma 9 that gives us that the DC API response is written to the script state of the script *script_verifier_authentication* in Line 78 of Algorithm 8, i.e., $S^{t'}(b).\bar{d}'.\texttt{scriptstate} := \langle \texttt{DC\_API\_RESPONSE}, [\texttt{vp\_token}, pres]\rangle$. After that the processing step finishes in Line 80 of Algorithm 8 without sending *pres* to the network.

The only line where b reads the script state of a document is Line 9 of Algorithm 15. This happens after b receives an event of the type $e_{\text{in}} \sim \langle *, *, \texttt{TRIGGER}\rangle$, selects `script` in Line 8 of Algorithm 17, and chooses the $\bar{w}$ (Line 10 of Algorithm 17) and $\bar{d}$ (Line 12 of Algorithm 17) such that $S^u(b).\bar{d}.\texttt{nonce} \equiv docnonce$. In the following we will call this processing step $U = (S^u, E^u, N^u) \xrightarrow[b \to E_{\text{out}}]{e_{\text{in}} \to b} (S^{u'}, E^{u'}, N^{u'})$. The processing step U must exists because otherwise v

could not receive *pres* in P as stated by $\mathsf{acceptsPresentation}_{\rho}^{P}(v, id, sTID, pres, k_{\text{holder}})$. This is due to the fact, that Line 9 of Algorithm 15 is the only line that does read *pres* from b's state again.

In the following execution of the processing step U the script *script_verifier_authentication* is run in Line 10 of Algorithm 15. Looking into the definition of this script gives us that due to the script state of the form $\langle \texttt{DC\_API\_RESPONSE}, [\texttt{vp\_token}, pres]\rangle$ we know that the script terminates with a `FORM` command that sends $[\texttt{vp\_token}, pres]$ to a URL url'. The domain of url' is from GETURL and the protocol and path is hard-coded to `S` and `/dc-api-response`. Since GETURL gets the document nonce $S^u(b).\bar{d}.nonce$ it is clear from the definition of the function that $url'.\texttt{host} \in \mathsf{dom}(v)$. The `FORM` command is processed in Lines 36ff. of Algorithm 15 which ends in a `POST` request to v in Line 51 of Algorithm 15.

Due to the fact, that v receives *pres* there must be a processing step $Z = (S^z, E^z, N^z) \xrightarrow[b \to E_{\text{out}}]{e_{\text{in}} \to b} (S^{z'}, E^{z'}, N^{z'})$ in which b emits an output event of the form $e_{\text{out}} \sim \langle *, *, \mathsf{enc_a}(\langle\langle \texttt{HTTPReq}, *, *, url'.\texttt{host}, *, *, *, [\texttt{vp\_token}\colon pres]\rangle, k\rangle, \mathsf{pub}(priv_v))\rangle$ with $k \in \mathcal{N}$ which is stored in $S^{z'}(b).\texttt{pendingRequests}$ in Line 70 of Algorithm 17 $\mathsf{pub}(priv_v)$ is retrieved from $S^z(b).\texttt{keyMapping}[url'.\texttt{host}]$ in Line 71 of Algorithm 17. From the initialization of the browser's initial state in Definition 8 we know that $priv_v \equiv \mathsf{tlskey}(url'.\texttt{host})$ (a browser never changes its *keyMapping* state). Due to the fact that tlskey assigns a different key to every domain, there is no other function that distributes K_{TLS}, and v does not leak $priv_v$ (because v is honest in S^*, $S^*(b).\texttt{tlskeys}$ are not used in Algorithm 1 to Algorithm 4 and the generic HTTPS server does not leak them either) we can apply Lemma 11 and get that only v can decrypt this message.

From $\mathsf{acceptsPresentation}_{\rho}^{P}(v, id, sTID, pres, k_{\text{holder}})$ we know that $m := \langle *, *, \mathsf{enc_a}(\langle\langle \texttt{HTTPReq}, *, *, *, *, *, *, [\texttt{vp\_token}\colon pres]\rangle, *\rangle, \mathsf{pub}(*))\rangle$ is processed by v in the processing step P. Due to the fact that $url'.\texttt{path} \equiv \texttt{/dc-api-response}$ and the request method is `POST` the message is processed by v in Line 23 of Algorithm 1. In the following v creates *sTID* in Line 28 of Algorithm 1 and includes it in the header of the response in Line 34 of Algorithm 1 as $\langle \texttt{Set-Cookie}, [\texttt{serviceTokenID}\colon \langle sTID, \top, \top, \top\rangle]\rangle$. The HTTP response $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 303, \langle *, \langle \texttt{Set-Cookie}, [\texttt{serviceTokenID}\colon \langle sTID, \top, \top, \top\rangle]\rangle\rangle, \langle\rangle\rangle, k)$ is encrypted with k in Line 33 of Algorithm 1. Additional *sTID* is stored in $S^{p'}(v).\texttt{sessions}$ in Line 29 of Algorithm 1. We now have to prove that *sTID* does not leak to the attacker:

(C.i) **v does not leak *sTID*:** The verifier v stores *sTID* in its *sessions* state in Line 29 of Algorithm 1 and includes it in the HTTPS response in Line 34 of Algorithm 1. *sTID* does not leak from the HTTPS response as shown in (C.ii). Due to the fact that *sessions* is never read it can not be leaked from the *sessions* state.

The *sTID* cookie can also be received in a header of an HTTP request from b to v, but v does never process cookies that are sent to it.

(C.ii) **Only b can and does decrypt m':** v does never process a message of the form m'. Looking into the definition of Algorithm 1 gives us that k is never leaked because it is only used as an encryption key in Line 21 of Algorithm 1 and Line 33 of Algorithm 1, v is honest in S of P, and the generic HTTPS server does also not leak k. If at some point b processes m' we can apply Lemma 11 again which gives us that *sTID* is only known to b and v. Looking into the definition of a browser shows that a message like m' is processed by Algorithm 16 after it is received by Algorithm 17. The browser b stores *sTID* in its *cookies* state under a host of v in Line 4 of Algorithm 16.

(C.iii) **b does not send *sTID* to a different process than v:** Due to the fact that b is honest in S^* it will only every read *cookies* in Line 4 of Algorithm 12 and Line 3 of Algorithm 15. The later one will not retrieve *sTID* because it has set the HTTP only flag. In Line 4

of Algorithm 12 *sTID* is only retrieved if it is send over HTTPS to v because the cookie has the secure flag set and cookies are always stored with the host that set the cookie. In Line 15 of Algorithm 12 the cookie is stored in *pendingRequests* in b's state as part of an HTTPS request. This is always an HTTP request because Algorithm 12 is only called with an HTTP message. Since b is honest in S^* the only lines where *pendingRequests* is read is Lines 65ff. of Algorithm 17. This results in an encrypted HTTPS request in Line 71 of Algorithm 17. Due to the fact that it is encrypted with v's public key in Line 71 of Algorithm 17, we can again apply Lemma 11 and get that only v can decrypt the HTTPS request. ■

D. Notes on the OID4VP Specification

In our report for Deliverable A.1(A), we proposed two changes to the OID4VP specification, which the working group subsequently incorporated. For completeness, we include the proposed changes below.

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

The working group introduced the corresponding requirement on validating the origin in OpenID4VP PR#544.

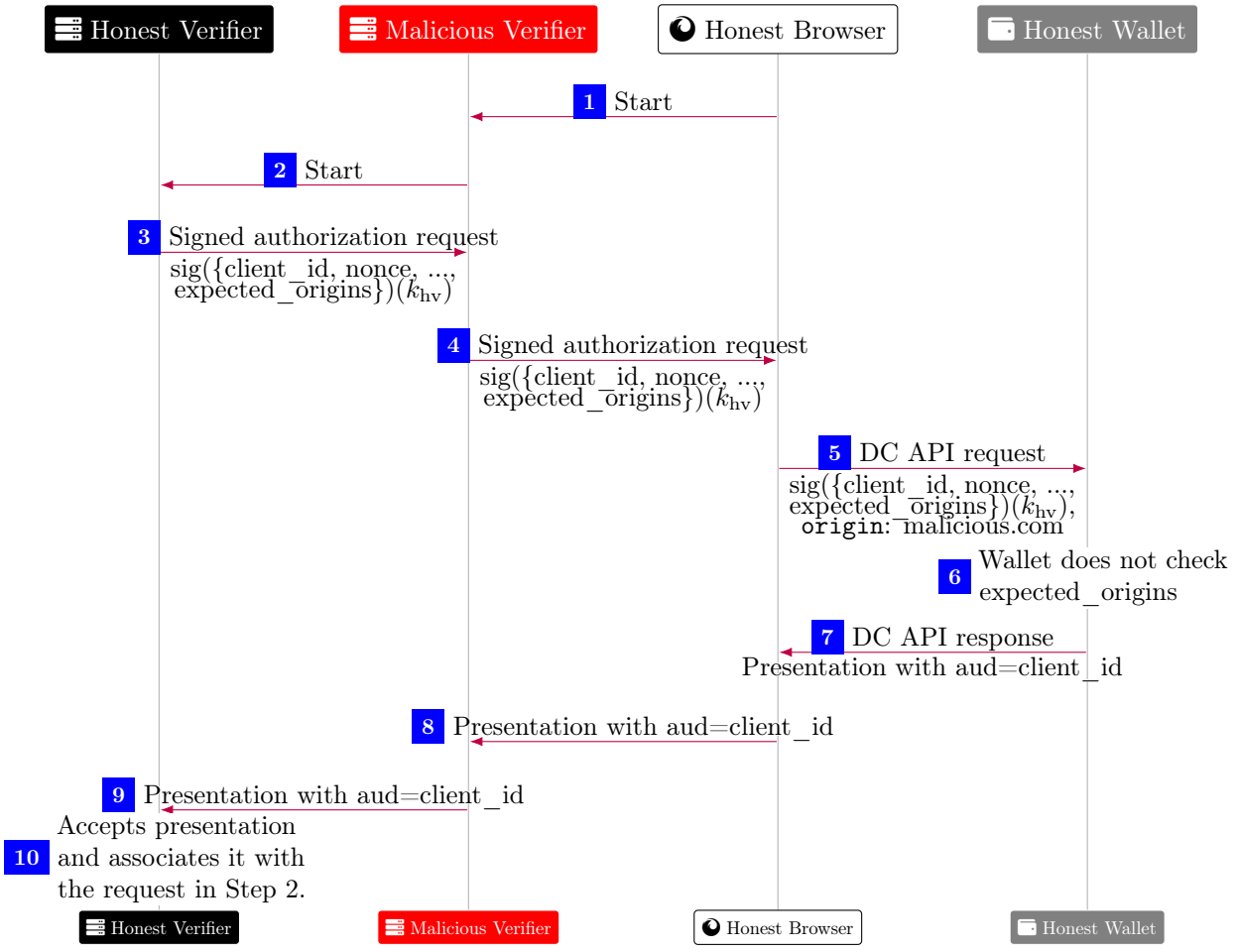

Figure 1: Attack on draft version 24 if expected_origins is not validated: The honest verifier associates the honest wallet's presentation with the session between the honest and the malicious verifier.

Section 14.1 and DC API For the draft that we modeled for Deliverable A.1(A) [OID4VP], we suggested updating Section 14.1 to incorporate OID4VP over the DC API. Paragraph 3 of the draft, 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.

The working group introduced corresponding changes to the specification in OpenID4VP PR#465.

E. Technical Definitions

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

E.1. Terms and Notations

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

Definition 16 (Signature). In the case of the WIM, the signature Σ consists of the following pairwise disjoint sets:

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

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

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

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

Definition 17 (Nonces and Terms). Given this signature, we define $X = \{x_1, x_2, \ldots\}$ to be an infinite set of variables, and $\mathcal{N}$ to be an infinite set of constants (*nonces*) such that $\Sigma, X, \mathcal{N}$ are pairwise disjoint. With these, we can now define the set of terms $\mathcal{T}_N(X)$ over $\Sigma \cup X \cup N$ for any set $N \subseteq \mathcal{N}$ inductively as follows:

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

Definition 18 (Equational Theory and Term Equivalence). Furthermore, we associate an equational theory with Σ, modeling the semantics of the function symbols. Our equational theory is defined by the following equations:

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

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

Definition 19 (Ground Terms, Messages, Placeholders, Protomessages). $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$ denotes the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

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

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

Definition 20 (Events and Protoevents). An *event (over* IPs *and* $\mathcal{M}$*)* is a term of the form $\langle a, f, m\rangle$, for a, $f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events. Events over IPs and $\mathcal{M}^\nu$ are called *protoevents* and are denoted $\mathcal{E}^\nu$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^\nu\langle\rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle\rangle$, $\langle\langle a, f, m\rangle, \langle a', f', m'\rangle, \dots\rangle$, etc.).

Definition 21 (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 18. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 22 (Pattern Matching). Let *pattern* $\in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches pattern* iff t can be acquired from *pattern* by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim$ *pattern*. For a sequence of patterns *patterns* we write $t \dot{\sim}$ *patterns* to denote that t matches at least one pattern in *patterns*.

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

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

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

Definition 23 (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 24 (Sequence Notations). Let $t = \langle t_1, \dots, t_n\rangle$ and $r = \langle r_1, \dots, r_m\rangle$ be sequences, s a set, and x, y terms. We define the following operations:

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

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

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

Definition 25 (Dictionaries). A *dictionary over X and Y* is a term of the form

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

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

Definition 26 (Operations on Dictionaries). Let $z = [k_1 \colon v_1, k_2 \colon v_2, \ldots, k_n \colon v_n]$ be a dictionary with unique keys, i.e., $\forall i, j \colon k_i \neq k_j$. In addition, let t and v be terms. We define the following operations:

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

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

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

Definition 27 (Pointers). A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

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

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

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

Definition 28 (Concatenation of Sequences). For a sequence $a = \langle a_1, \ldots, a_i \rangle$ and a sequence $b = \langle b_1, b_2, \ldots \rangle$, we define the *concatenation* as $a \cdot b := \langle a_1, \ldots, a_i, b_1, b_2, \ldots \rangle$.

Definition 29 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a non-deterministically chosen occurence of that element in X is removed.

E.2. Message and Data Formats

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

E.2.1. URLs

Definition 30. A *URL* is a term of the form

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

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

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

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 27):

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

E.2.2. Origins

Definition 31. An *origin* is a term of the form $\langle host, protocol \rangle$ with $host \in \mathsf{Doms}$ and $protocol \in \{\texttt{P}, \texttt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins.

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

E.2.3. Cookies

Definition 32. A *cookie* is a term of the form $\langle name, content \rangle$ where $name \in \mathcal{T}_{\mathcal{N}}$, and *content* is a term of the form $\langle value, secure, session, httpOnly \rangle$ where $value \in \mathcal{T}_{\mathcal{N}}$, $secure$, $session$, $httpOnly \in \{\top, \bot\}$. As *name* is a term, it may also be a sequence consisting of two parts. If the name consists of two parts, we call the first part of the sequence (i.e., $name.\texttt{1}$) the *prefix* of the name. We write $\mathsf{Cookies}$ for the set of all cookies and $\mathsf{Cookies}^{\nu}$ for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

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

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

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

E.2.4. HTTP Messages

Definition 33. An *HTTP request* is a term of the form shown in (15). An *HTTP response* is a term of the form shown in (16).

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

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

The components are defined as follows:

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

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

Example 6 (HTTP Request and Response).

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

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

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

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

Definition 34. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k' \rangle, k)$, where $k \in \mathit{terms}$, $k' \in \mathcal{N}$, and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc_s}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

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

Example 7.

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

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

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

E.2.5. DNS Messages

Definition 35. A *DNS request* is a term of the form $\langle \texttt{DNSResolve}, \mathit{domain}, \mathit{nonce} \rangle$ where $\mathit{domain} \in \mathsf{Doms}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$.

Definition 36. A *DNS response* is a term of the form $\langle \texttt{DNSResolved}, \mathit{domain}, \mathit{result}, \mathit{nonce} \rangle$ with $\mathit{domain} \in \mathsf{Doms}$, $\mathit{result} \in \mathsf{IPs}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS responses $\mathsf{DNSResponses}$.

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

E.3. Atomic Processes, Systems and Runs

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

Definition 37 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

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

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

Definition 38 (Configurations). A *configuration of a system* $\mathcal{P}$ is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence[3] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$.

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

Definition 39 (Processing Steps). A *processing step of the system* $\mathcal{P}$ is of the form

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

where

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

such that there exists

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

with

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

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

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

Definition 40 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0* is a finite sequence of configurations $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

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

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

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

E.4. Atomic Dolev-Yao Processes

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

Definition 41 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with variables V* if there exist $m_1, \ldots, m_n \in M$ with $n \geq 0$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \ldots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.

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

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

Definition 42 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ such that p is an atomic process and for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_\mathcal{N}$, $s' \in \mathcal{T}_\mathcal{N}(V_{\text{process}})$, with $((e, s), (E, s')) \in R^p$ it holds true that $E, s' \in d_{V_{\text{process}}}(\{e, s\})$.

E.5. Attackers

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

Definition 43 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$* is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e, and $s \in \mathcal{T}_\mathcal{N}$ we have that $((e, s), (E, s')) \in R$ iff $s' = \langle e, E, s\rangle$ and $E = \langle\langle a_1, f_1, m_1\rangle, \ldots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_1, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e, s\})$. 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.

E.6. Notations for Functions and Algorithms

When describing algorithms, we use the following notations:

E.6.1. Non-deterministic choosing and iteration

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

element in M, where the variable s is the current element. The order in which the elements are processed is chosen non-deterministically. We write, for example,

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

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

E.6.2. Function calls

When calling functions that do not return anything, we write

call FUNCTION_NAME(x, y)

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

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

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

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

E.6.3. Stop without output

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

E.6.4. Placeholders

In several places throughout the algorithms we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 17). Table 2 shows a list of some of the placeholders, generally denoted by ν with some subscript to distinguish between multiple fresh values.

E.6.5. Abbreviations for URLs and Origins

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

E.7. Browsers

Here, we present the formal model of browsers.

E.7.1. Scripts

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

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
$\nu_{10}, \dots$	Algorithm 15, replacement for placeholders in script output

Table 2: List of placeholders used in browser algorithms.

Definition 44 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 45 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{\text{script}}}(s)$.

A script is called by the browser which provides it with state information (such as the script's last scriptstate and limited information about the browser's state) s. The script then outputs a term s', which represents the new scriptstate and some command which is interpreted by the browser. The term s' may contain variables $\lambda_1, \dots$ which the browser will replace by (otherwise unused) placeholders $\nu_1, \dots$ 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 46 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$.

E.7.2. Web Browser State

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

Definition 47. A *window* is a term of the form $w = \langle nonce, documents, opener \rangle$ with $nonce \in \mathcal{N}$, $documents \subset^{\langle\rangle}$ Documents (defined below), $opener \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle} documents$ if $documents$ is not empty (we then call d the *active document of* w). We write Windows for the set of all windows. We write $w.\texttt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else.

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

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

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

Definition 48. A *document* d is a term of the form

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

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

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

Definition 49. For two window terms w and w' we write

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

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

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

Definition 50. A reference for a normal HTTP(S) request is a sequence of the form $\langle \texttt{REQ}, nonce\rangle$, where $nonce$ is a window reference. A reference for a XMLHttpRequest is a sequence of the form $\langle \texttt{XHR}, nonce, xhrreference\rangle$, where $nonce$ is a document reference and $xhrreference$ is some nonce that was chosen by the script that initiated the request.

We can now define the set of states of Web browsers. Note that we use the dictionary notation that we introduced in Definition 25.

Definition 51. The *set of states* $Z_{\text{webbrowser}}$ *of a Web browser atomic Dolev-Yao process* consists of the terms of the form

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

with the subterms as follows:

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

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

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

E.7.3. Web Browser Relation

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

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

Subwindows and Docs. Given a browser state s, Subwindows(s) denotes the set of all pointers[4] to windows in the window list $s.\texttt{windows}$ and (recursively) the subwindows of their active documents. We exclude subwindows of inactive documents and their subwindows. With Docs(s) we denote the set of pointers to all active documents in the set of windows referenced by Subwindows(s).

Definition 52. For a browser state s we denote by Subwindows(s) the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\texttt{windows}$ there is a $\overline{p} \in$ Subwindows(s) such that $s.\overline{p} = w$. (2) For all $\overline{p} \in$ Subwindows(s), the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in$ Subwindows(s) such that $s.\overline{p'} = w$.

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

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

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

[4] Recall the definition of a pointer in Definition 27.

Definition 53. Let s be a browser state and d a document. By $\mathsf{Clean}(s, d)$ we denote the term that equals $s.\texttt{windows}$ but with (1) all inactive documents removed (including their subwindows etc.), (2) all subterms that represent non-same-origin documents w.r.t. d replaced by a limited document d' with the same nonce and the same subwindow list, and (3) the values of the subterms `headers` for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.)

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

Definition 54. For a sequence of cookies (with pairwise different names) *oldcookies*, a sequence of cookies *newcookies*, and a string $protocol \in \{\texttt{P}, \texttt{S}\}$, the set $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\texttt{content.httpOnly} \equiv \top$ or where $(c.\texttt{name.1} \equiv \texttt{\_\_Host}) \wedge ((protocol \equiv \texttt{P}) \vee (c.\texttt{secure} \equiv \bot))$. For any c, $c' \in^{\langle\rangle}$ *newcookies*, $c.\texttt{name} \equiv c'.\texttt{name}$, remove the cookie that appears left of the other in *newcookies*. Let m be the set of cookies that have a name that either appears in *oldcookies* or in *newcookies*, but not in both. For all pairs of cookies $(c_{\text{old}}, c_{\text{new}})$ with $c_{\text{old}} \in^{\langle\rangle}$ *oldcookies*, $c_{\text{new}} \in^{\langle\rangle}$ *newcookies*, $c_{\text{old}}.\texttt{name} \equiv c_{\text{new}}.\texttt{name}$, add c_{new} to m if $c_{\text{old}}.\texttt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ is m.

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

Definition 55. For a sequence of cookies (with pairwise different names) *oldcookies*, a cookie c, and a string $protocol \in \{\texttt{P}, \texttt{S}\}$ (denoting whether the HTTP response was received from an insecure or a secure origin), the sequence $\mathsf{AddCookie}(oldcookies, c, protocol)$ is defined by the following algorithm: Let $m :=$ *oldcookies*. If $(c.\texttt{name.1} \equiv \texttt{\_\_Host}) \wedge \neg((protocol \equiv \texttt{S}) \wedge (c.\texttt{secure} \equiv \top))$, then return m, else: Remove any c' from m that has $c.\texttt{name} \equiv c'.\texttt{name}$. Append c to m and return m.

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

Definition 56. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

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

Algorithm 9 Web Browser Model: Determine window for navigation.

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

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

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

Functions

- The function GETNAVIGABLEWINDOW (Algorithm 9) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When it is given a window reference (nonce) *window*, this function returns a pointer to a selected window term in s':
  - If *window* is the string `_BLANK`, a new window is created and a pointer to that window is returned.
  - If *window* is a nonce (reference) and there is a window term with a reference of that value in the windows in s', a pointer $\overline{w'}$ to that window term is returned, as long as the window is navigable by the current window's document (as defined by NavigableWindows above).

  In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).
- The function GETWINDOW (Algorithm 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.

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

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

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

Algorithm 13 Web Browser Model: Navigate a window backward.

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

Algorithm 14 Web Browser Model: Navigate a window forward.

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

Algorithm 15 Web Browser Model: Execute a script.

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

This algorithm is continued on the next page.

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

Algorithm 16 Web Browser Model: Process an HTTP response.

```
1: function PROCESSRESPONSE(response, reference, request, requestUrl, a, f, s′)
2:     if Set-Cookie ∈ response.headers then
3:         for each c ∈^⟨⟩ response.headers[Set-Cookie], c ∈ Cookies do
4:             let s′.cookies[request.host]
                   ↪ := AddCookie(s′.cookies[request.host], c, requestUrl.protocol)
5:     if Strict-Transport-Security ∈ response.headers ∧ requestUrl.protocol ≡ S then
6:         let s′.sts := s′.sts +^⟨⟩ request.host
7:     if Referer ∈ request.headers then
8:         let referrer := request.headers[Referer]
9:     else
10:        let referrer := ⊥
11:    if Location ∈ response.headers ∧ response.status ∈ {303, 307} then
12:        let url := response.headers[Location]
13:        if url.fragment ≡ ⊥ then
14:            let url.fragment := requestUrl.fragment
15:        let method′ := request.method
16:        let body′ := request.body
17:        if Origin ∈ request.headers
               ↪ ∧ request.headers[Origin] ≠ ◇
               ↪ ∧ (⟨url.host, url.protocol⟩ ≡ ⟨request.host, requestUrl.protocol⟩
                     ↪ ∨ ⟨request.host, requestUrl.protocol⟩ ≡ request.headers[Origin]) then
18:            let origin := request.headers[Origin]
19:        else
20:            let origin := ◇
21:        if response.status ≡ 303 ∧ request.method ∉ {GET, HEAD} then
22:            let method′ := GET
23:            let body′ := ⟨⟩
24:        if ∃ w̄ ∈ Subwindows(s′) such that s′.w̄.nonce ≡ π2(reference) then  → Do not redirect XHRs.
25:            let req := ⟨HTTPReq, ν6, method′, url.host, url.path, url.parameters, ⟨⟩, body′⟩
26:            let referrerPolicy := response.headers[ReferrerPolicy]
27:            call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
28:        else
29:            stop ⟨⟩, s′
```

This algorithm is continued on the next page.

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

- The function HTTP_SEND (Algorithm 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'.pendingDNS until the DNS resolution finishes. *reference* is a reference as defined in Definition 50. *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 $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted.

- The function PROCESSRESPONSE (Algorithm 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 50. *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 57. 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).

E.8. Definition of Web Browsers

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

Definition 58 (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 59 (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 51) with the following properties:

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

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

E.9. Helper Functions

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

Algorithm 17 Web Browser Model: Main Algorithm.

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

let $s' := s$
if $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
  let $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests}\rangle$ → Collect incoming messages
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle$, s'
if $m \equiv \texttt{TRIGGER}$ **then** → A special trigger message.
  let *switch* $\leftarrow \{\texttt{script}, \texttt{urlbar}, \texttt{reload}, \texttt{forward}, \texttt{back}\}$
  if *switch* $\equiv \texttt{script}$ **then** → Run some script.
    let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
    let n **such that** $s'.\overline{w}.n.\texttt{active} \equiv \top$ **if possible; otherwise stop**
    let $\overline{d} := \overline{w} +^{\langle\rangle} n$
    call RUNSCRIPT($\overline{w}$, $\overline{d}$, a, s')
  else if *switch* $\equiv \texttt{urlbar}$ **then** → Create some new request.
    let *newwindow* $\leftarrow \{\top, \bot\}$
    if *newwindow* $\equiv \top$ **then** → Create a new window.
      let *windownonce* $:= \nu_1$
      let $w' := \langle$*windownonce*$, \langle\rangle, \bot\rangle$
      let $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
    else → Use existing top-level window.
      let $\overline{tlw} \leftarrow \mathbb{N}$ **such that** $s'.\overline{tlw}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some top-level window.
      let *windownonce* $:= s'.\overline{tlw}.$*nonce*
    let *protocol* $\leftarrow \{\texttt{P}, \texttt{S}\}$
    let *host* $\leftarrow \mathsf{Doms}$
    let *path* $\leftarrow \mathbb{S}$
    let *fragment* $\leftarrow \mathbb{S}$
    let *parameters* $\leftarrow [\mathbb{S} \times \mathbb{S}]$
    let *url* $:= \langle\texttt{URL}, $*protocol*$, $*host*$, $*path*$, $*parameters*$, $*fragment*$\rangle$
    let *req* $:= \langle\texttt{HTTPReq}, \nu_2, \texttt{GET},$ *host*, *path*, *parameters*, $\langle\rangle, \langle\rangle\rangle$
    call HTTP_SEND($\langle\texttt{REQ},$ *windownonce*$\rangle$, *req*, *url*, $\bot$, $\bot$, $\bot$, a, s')
  else if *switch* $\equiv \texttt{reload}$ **then** → Reload some document.
    let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
    let *url* $:= s'.\overline{w}.\texttt{activedocument.location}$
    let *req* $:= \langle\texttt{HTTPReq}, \nu_2, \texttt{GET},$ *url*.host, *url*.path, *url*.parameters, $\langle\rangle, \langle\rangle\rangle$
    let *referrer* $:= s'.\overline{w}.\texttt{activedocument.referrer}$
    let $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\texttt{nonce}, s')$
    call HTTP_SEND($\langle\texttt{REQ}, s'.\overline{w}.\texttt{nonce}\rangle$, *req*, *url*, $\bot$, *referrer*, $\bot$, a, s')
  else if *switch* $\equiv \texttt{forward}$ **then**
    let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
    call NAVFORWARD($\overline{w}$, s')
  else if *switch* $\equiv \texttt{back}$ **then**
    let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
    call NAVBACK($\overline{w}$, s')
else if $m \equiv \texttt{FULLCORRUPT}$ **then** → Request to corrupt browser
  let $s'.\texttt{isCorrupted} := \texttt{FULLCORRUPT}$
  stop $\langle\rangle$, s'
else if $m \equiv \texttt{CLOSECORRUPT}$ **then** → Close the browser
  let $s'.\texttt{secrets} := \langle\rangle$
  let $s'.\texttt{windows} := \langle\rangle$
  let $s'.\texttt{pendingDNS} := \langle\rangle$

This algorithm is continued on the next page.

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

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

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

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

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

CHOOSEINPUT (Algorithm 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 CHOOSEINPUT(s', *scriptinputs*) is used, where s' denotes the scripts current state. It saves the indexes of already handled messages in the scriptstate s' and chooses a yet unhandled input message from *scriptinputs*. The index of this message is then saved in the scriptstate (which is returned to the script).

CHOOSEFIRSTINPUTPAT (Algorithm 19) Similar to the function CHOOSEINPUT above, we define the function CHOOSEFIRSTINPUTPAT. This function takes the term *scriptinputs*, which as above records the input this document has obtained so far (via XHRs and postMessages, append-only), and a pattern. If called, this function chooses the first message in *scriptinputs* that matches *pattern* and returns it. This function is typically used in places, where a script only processes the first message that matches the pattern. Hence, we omit recording the usage of an input.

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

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

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

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

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

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

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

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

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

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

E.10. DNS Servers

Definition 60. A *DNS server* d (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form

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

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

The relation $R^d \subseteq (\mathcal{E} \times \{s_0^d\}) \times (2^{\mathcal{E}} \times \{s_0^d\})$ of d above is defined by Algorithm 20.

E.11. Web Systems

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

Algorithm 20 Relation of a DNS server R^d.

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

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

Definition 61. A *Web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a tuple with its components defined as follows:

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

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

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

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

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

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

E.12. Generic HTTPS Server Model

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

Definition 62 (Base state for an HTTPS server). The state of each HTTPS server that is an instantiation of this relation must contain at least the following subterms: $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \mathsf{IPs}$ (containing the IP address of a DNS server), $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (containing a mapping from domains to public keys), $tlskeys \in [\mathsf{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $corrupt \in \mathcal{T}_{\mathcal{N}}$ (either $\perp$ if the server is not corrupted, or an arbitrary term otherwise).

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

Let ν_{n0} and ν_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. We now define the default functions of the generic Web server in Algorithms 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).

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

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

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

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

function PROCESS_TRIGGER(a, s')
    stop

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

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

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

function PROCESS_OTHER(m, a, f, s')
    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).

function MERGE_DICTS(*dict1*, *dict2*)
    for $key \in \mathcal{T}_{\mathcal{N}}$ **do**
        if $key \in dict2$ **then**
            if $key \in dict1$ **then**
                stop → Duplicate key.
            let $dict1[key] := dict2[key]$
    return *dict1*

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

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

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

E.13. General Security Properties of the WIM

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

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

Definition 63 (Emitting Events). Given an atomic process p, an event e, and a finite run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \ldots)$ we say that p *emits* e iff there is a processing step in ρ of the form

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

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

Definition 64. We say that a term t *is derivably contained in (a term)* t' *for (a set of DY processes)* P *(in a processing step* $s_i \to s_{i+1}$ *of a run* $\rho = (s_0, s_1, \ldots)$*)* if t is derivable from t' with the knowledge available to P, i.e.,

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

Definition 65. We say that *a set of processes* P *leaks a term* t *(in a processing step* $s_i \to s_{i+1}$*) to a set of processes* P' if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.

Definition 66. We say that a DY process p *created* a message m in a processing step

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

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

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

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

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

Definition 67. 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 68. 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 69. Let $N \subseteq \mathcal{N}$, $t \in \mathcal{T}_N(X)$, and $k \in \mathcal{T}_N(X)$. We say that k *appears only as a public key* in t, if

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

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

Definition 70. 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 71. 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 $\rho = s_0, s_1, \ldots$ of any $\mathcal{WS}$, we state the following lemmas:

Lemma 11. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b

(I) emits an HTTPS request of the form

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

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

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

(III) u never leaks k',

then all of the following statements are true:

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

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

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

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

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

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

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

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

The browser is honest in s_i. From the definition of the browser b, we see that the key k is always chosen as a fresh nonce (placeholder ν_3 in Lines 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 11, as long as u does not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ and the browser does not become fully corrupted, k is not known to any DY process $p \notin \{u, b\}$ (i.e., $\nexists\, s' = (S', E') \in \rho\colon\ k \in d_{N^p}(S'(p))$).

Lemma 12. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\texttt{windows}$ with the origin $\langle dom, \mathsf{S}\rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then b extracted (in Line 37 in Algorithm 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 $S_i(b).\texttt{keyMapping}[dom] = S_0(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ for p (the key mapping cannot be altered during a run), we can see that *req* was encrypted using a public key that matches a private key which is only (if at all) known to p. With Lemma 11 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 13. If in a processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, \mathtt{S} \rangle$ where $S_i(b).\mathtt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then

- that request was initiated by a script that b extracted (in Line 37 in Algorithm 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 12 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 11, but is applied to the generic HTTPS server (instead of the Web browser).

Lemma 14. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest instance s of the generic HTTPS server model

(I) emits an HTTPS request of the form

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

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

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

(III) u never leaks k',

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

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

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

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

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

then all of the following statements are true:

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

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

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

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

Proof. **(1)** follows immediately from the preconditions. The proof is the same as for Lemma 11:

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

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

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

The process s is honest in s_i. s emits HTTPS requests like m only in Line 18 of Algorithm 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 $S^{i+1}(s).\texttt{pendingRequests}$. 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'.