The Web Infrastructure Model (WIM)*

Daniel Fett[1], Ralf Küsters[2], and Guido Schmitz[3]

[1]yes.com AG. mail@danielfett.de
[2]University of Stuttgart, Stuttgart, Germany. ralf.kuesters@sec.uni-stuttgart.de
[3]Royal Holloway University of London, Egham, Surrey, United Kingdom.
guido.schmitz@rhul.ac.uk

Contributors: Florian Helmschmidt, Pedram Hosseyni, Tim Würtele

Abstract. The Web Infrastructure Model (WIM) is a generic Dolev-Yao style model of the Web infrastructure. The WIM has first been published in [9] and has been extended in follow-up publications [6, 7, 10, 11, 12, 13]. This document consolidates these various extensions to provide a single reference version of the WIM for subsequent work.

Contents

1 Overview of the WIM **3**
1.1 Introduction . . . 3
1.2 Communication Model . . . 3
1.3 Terms, Messages, and Events . . . 3
1.4 Processes, Systems, and Runs . . . 5
1.5 Attackers . . . 6
1.6 Browsers and Scripts . . . 7
1.6.1 Dispatchment of HTTP(S) Requests . . . 8
1.6.2 Handling of Trigger Messages . . . 9
1.6.3 Handling of DNS Responses . . . 10
1.6.4 Handling of HTTP(S) Responses . . . 11
1.6.5 Corruption . . . 11
1.7 Web Servers . . . 12
1.8 DNS Servers . . . 12
1.9 Limitations . . . 12

2 Technical Definitions **13**
2.1 Terms and Notations . . . 13
2.2 Message and Data Formats . . . 15
2.2.1 URLs . . . 15
2.2.2 Origins . . . 15
2.2.3 Cookies . . . 15
2.2.4 HTTP Messages . . . 16
2.2.5 DNS Messages . . . 17
2.3 Atomic Processes, Systems and Runs . . . 17
2.4 Atomic Dolev-Yao Processes . . . 18
2.5 Attackers . . . 18
2.6 Notations for Functions and Algorithms . . . 19
2.6.1 Non-deterministic choosing and iteration . . . 19
2.6.2 Function calls . . . 19
2.6.3 Stop without output . . . 19

*Version 1.0

2.6.4 Placeholders 19
2.6.5 Abbreviations for URLs and Origins 19
2.7 Browsers 20
2.7.1 Scripts 20
2.7.2 Web Browser State 20
2.7.3 Web Browser Relation 22
2.8 Definition of Web Browsers 28
2.9 Helper Functions 30
2.10 DNS Servers 32
2.11 Web Systems 32
2.12 Generic HTTPS Server Model 32
2.13 General Security Properties of the WIM 34

3 Modeling Decisions **39**
3.1 Web attackers can't perform IP spoofing 39
3.2 Documents can't contain multiple scripts 39
3.3 Domain boundaries of web pages and cookies can't be extended 40

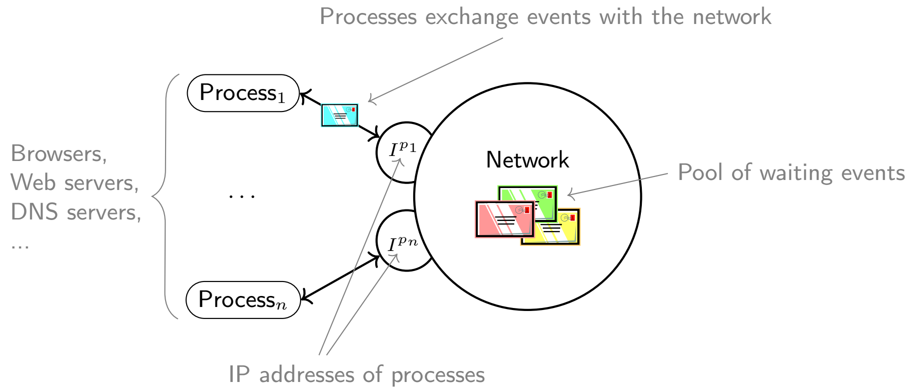


Figure 1: Illustration of the WIM's communication model.

1 Overview of the WIM

1.1 Introduction

The Web Infrastructure Model (WIM) is a generic Dolev-Yao style model of the Web infrastructure. The WIM has first been published in [9] and has been extended in follow-up publications [6, 7, 10, 11, 12, 13]. This document consolidates these various extensions to provide a single reference version of the WIM for subsequent work. In this section we recall this model following the descriptions in said publications with more technical definitions given in Section 2.

The WIM is designed independently of a specific Web application and closely mimics published (de-facto) standards and specifications for the Web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The WIM defines a general communication model, and, based on it, Web systems consisting of Web browsers, DNS servers, and Web servers as well as Web and network attackers.

Extensions not covered by this document. The following mechanisms and extensions have been introduced in previous publications and are not covered by this document:

- Extensions for **WebSockets** and **WebRTC** can be found in [6] in Section 2.10.6 and Section 2.12 respecetively, with further details in Appendices A.4.6 and A.4.7.
- The formal definition of a **challenge browser** for the privacy analysis of single sign-on systems can be found in Definition 74 in Appendix C of [17].
- Extensions for **DOM event processing**, the **Web Payment APIs** and a framework for **service workers** can be found in Section V of [4], with details in Appendices B and C.
- Extensions for the **OpenID Financial-grade API** that model **OAuth 2.0 Token Binding** can be found in Appendix G of [8].

1.2 Communication Model

The communication model of the WIM is illustrated in Figure 1. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes communicate via *events*, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from a "pool" of waiting events and is delivered to one of the processes which listens to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on.

1.3 Terms, Messages, and Events

As usual in Dolev-Yao models (see, e.g., [1, 5]), events and messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures).

Formally, the signature Σ for the terms and messages considered in this work is the union of the following pairwise disjoint sets of function symbols:

- constants $C = \mathsf{IPs} \cup \mathbb{S} \cup \{\top, \bot, \Diamond\}$ where the three sets are pairwise disjoint, $\mathbb{S}$ is interpreted to be the set of ASCII strings (including the empty string ε), and IPs is interpreted to be a set of (IP) addresses,
- function symbols for public keys, (a)symmetric encryption/decryption, signatures, hashing, and computing and verifying message authentication codes: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot)$, $\mathsf{extractmsg}(\cdot)$, $\mathsf{hash}(\cdot)$, $\mathsf{mac}(\cdot,\cdot)$, and $\mathsf{checkmac}(\cdot,\cdot)$,
- n-ary sequences $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc., and
- projection symbols $\pi_i(\cdot)$ for all $i \in \mathbb{N}$.

Based on Σ, we define terms as follows: Let $X = \{x_0, x_1, \dots\}$ be a set of variables and $\mathcal{N}$ be an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual:

1. If $t \in N \cup X$, then t is a term.
2. If $f \in \Sigma$ is an n-ary function symbol in Σ for some $n \geq 0$ and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.

By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*.

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models and is depicted in Figure 4 on Page 13. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example, $\mathsf{dec_a}(\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}})), k_{\text{ex.com}}) \equiv \langle r, k'\rangle$, i.e., these two terms are equivalent w.r.t. the equational theory.

For some term t, we use $t{\downarrow}$ to refer its normal form, i.e., a term t' with $t' \equiv t$ where all function symbols have been reduced from left to right as far as possible using the equational theory. For example, for some term $t = \mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y)$, we use $t{\downarrow}$ to refer to x.

For readability, strings (elements in $\mathbb{S}$) are depicted using a specific font. For example, `HTTPReq` and `HTTPResp` are strings. Further, we use a notation for *mappings* (dictionaries). For example:

$$[\mathtt{dictkey1}: \mathtt{value1}, \mathtt{dictkey2}: \mathtt{value2}] = \langle\langle \mathtt{dictkey1}, \mathtt{value1}\rangle, \langle \mathtt{dictkey2}, \mathtt{value2}\rangle\rangle$$

Full definitions of terms and notations can be found in Section 2.1.

In the context of the Web, we define several specific sets of terms: We denote by $\mathsf{Doms} \subseteq \mathbb{S}$ the set of *domains*, e.g., $\mathtt{example.com} \in \mathsf{Doms}$. By $\mathsf{Origins} \subseteq \mathsf{Doms} \times \{\mathtt{P}, \mathtt{S}\}$, we denote the set of *(Web) origins* with the second element of the origin denoting the protocol, i.e., `P` denoting an (insecure) HTTP origin and `S` denoting a (secure) HTTPS origin of the domain (first element of the origin). For example, the term $\langle \mathtt{example.com}, \mathtt{S}\rangle$ denotes the origin `https://example.com`. For HTTP(S) requests, we denote by $\mathsf{Methods} \subseteq \mathbb{S}$ the set of *methods*, e.g., $\mathtt{GET}, \mathtt{POST} \in \mathsf{Methods}$.

As already mentioned above, entities in our model communicate via events that contain a message. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$. 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 $\mathsf{pub}(k)$.

While messages can be arbitrary terms, we also define special kinds of messages:

- DNS messages
- HTTP messages,
- HTTPS messages,
- trigger messages, and
- corrupt messages.

While DNS and HTTP(S) messages model their real-world counterparts, *trigger messages* can be seen as "dummy messages" that are used to invoke actions in processes that are not taken in direct response to another message as in our communication model an action of some entity is always based on the processing of some message. For example, when a trigger message is delivered to a browser (which incorporates the behavior of a user, see below) and then this browser might (non-deterministically) follow some link on some Web page (currently opened in this browser). *Corrupt messages* are sent by attackers to honest parties in order to corrupt these parties, i.e., an

honest party becomes a (collaborator of an) attacker. We will discuss corruption in more depth in Section 1.5 below.

DNS and HTTP(S) messages can be further divided into *requests* and *responses*. A response is associated with a request. To match a response to a request, both kinds of messages contain a nonce.[1] When a request is created by some process, this process freshly chooses this nonce. This nonce is then also used for the corresponding response message. For example, an HTTP request is represented as a term r containing the nonce mentioned above (say, n_1), an HTTP method, a domain name, a path, URI parameters, request headers, and a message body. For instance, an HTTP GET request for the URI `http://example.com/show?p=1` is represented as

$$req := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{example.com}, \mathtt{/show}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$$

where the body and the list of request headers are empty. A corresponding HTTP response is represented as

$$resp := \langle \mathtt{HTTPResp}, n_1, \mathtt{200}, \langle\rangle, body\rangle$$

where the status code of the response 200 indicates that the request was accepted and processed by the server, the list of headers is empty, and *body* is a term containing the requested Web page.

For HTTPS messages, the underlying TLS channel is modeled in an abstract way as follows: The sender of the request (say, A, typically a browser) chooses a fresh symmetric key k' (a nonce) and includes k' in the request message. The request message is then asymmetrically encrypted with the public key of the receiver (say, B). Such an HTTPS request for the HTTP request *req* above is of the form

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

If $k_{\text{example.com}}$ (the private key for `example.com`) is only known to B, only B can decrypt this request message.[2] The symmetric key k' (now only known by A and B) is used by B to encrypt the response message, which can then later be decrypted by A using k'. Such an HTTPS response sent from B to A is of the form

$$\mathsf{enc_s}(resp, k').$$

An *event* (over IPs and $\mathcal{M}$) is 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 to be the sender address of the event. We denote by $\mathcal{E}$ the set of all events. Events can be compared to IP messages in practice, that carry some payload (a message) between two entities which are referred to by IP addresses.

For all formal definitions of messages and data formats, we refer the reader to Section 2.2.

1.4 Processes, Systems, and Runs

An (atomic) process takes its current state and an event as input, and then (non-deterministically) outputs a new state and a sequence of events. We typically require that the events and the state that an atomic process outputs can be computed (more formally, derived) from the current input event and state. Such atomic processes are called *atomic Dolev-Yao processes (or simply, a DY process)*.

An atomic Dolev-Yao process $p = (I^p, Z^p, R^p, s_0^p)$ is defined to have

- a set of associated (IP) addresses I^p,
- a set of (possible) states Z^p ($\subset \mathcal{T}_N$),
- a process relation R^p that defines transitions from an (input) event and a (current) state to a set of (output) events and a (new) state such that both its output is derivable from its input,[3] and
- an initial state s_0^p ($\in Z^p$).

We combine processes to a *system* $\mathcal{P}$ which is a (possibly infinite) set of atomic processes. A system itself does include state (except for the initial states of the processes). Also, as explained in the communication model above, processes communicate via events. Further, processes may use fresh nonces that have not been used before. We capture these aspects in a *configuration of a system*. A configuration is a tuple (S, E, N), which contains

[1] The nonce of an HTTP(S) message models the TCP/TLS connection of the real world. DNS messages are typically sent as UDP messages in the real world and contain such a nonce by specification.

[2] In analyses, we typically show that a private key for a domain that is not controlled by an attacker is only known to its legitimate Web server (B in this example), i.e., the key is initially only known to this server and does not leak to any other party.

[3] We typically describe a process relation using pseudo-code. See Algorithm 12 on Page 32 for a simple example.

- a mapping S from each atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$,
- an infinite sequence of waiting events E, i.e., $E = (e_1, e_2, \dots)$ where e_i are events that are about to be delivered to a process (for $i \in \mathbb{N}$), and
- an infinite sequence of (fresh) nonces $N = (n_1, n_2, \dots)$.

The sequence of waiting events E contains all events that have been sent by some process (but have not been delivered yet) and a (possibly infinite) number of trigger messages addressed to each (IP) address (interleaved by addresses). The sequence of nonces N is used to provide fresh, unique nonces to what we call a *processing step*. A processing step describes a transition from one configuration to a new configuration. In such a processing step, an event is non-deterministically taken from the sequence of waiting events, a process that is associated with the receiver's (IP) address of the event is selected,[4] and a new set of events and a new state for this process is (non-deterministically) derived using this process' relation. The relation might use nonces but does not assign them immediately. The nonces are described by placeholders, which are replaced with fresh nonces from the sequence of nonces by the processing step. We write, for example,

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

to denote the processing step from the configuration (S, E, N) to the configuration (S', E', N') in which some event e_{in} was delivered to some process p and p has outputted the set of events E_{out} (with S and S' are the states of the processes in the system, E and E' are pools of waiting events, and N and N' are sequences of unused nonces, i.e., N' contains all nonces from N except for the ones that are "used" in this processing step). We may omit the superscript and/or subscript of the arrow.

The output configuration of the processing step then contains

- the states of all processes S' (as a mapping as above), which are the same as in the previous configuration for each process except for the selected process,
- the sequence of waiting events E' where the delivered event has been removed and events output by the process are added, and
- the sequence of nonces N' without nonces that are used in this processing step.

A *run* ρ of a system is a (possibly infinite) sequence of configurations, such that there exists a processing step between each consecutive configurations.

For readability, given a finite run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \dots)$, we denote by Q_i a processing step $(S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ in ρ (with $i \geq 0$ and, for finite runs, $i < n$).

Based on this generic Dolev-Yao style model, we define *Web systems*. A Web system formalizes the Web infrastructure and Web applications. It contains a system $\mathcal{W}$ consisting of honest and attacker processes. Honest processes can be Web browsers, Web servers, or DNS servers. Attackers and (generic) honest processes are described in the next sections below. A Web system further contains a set of so-called scripts, which we will cover in Section 1.6. For each Web system, we also define the mapping addr that describes the ownership of (IP) addresses to DY process and the mapping dom that describes the ownership of domains to DY processes.

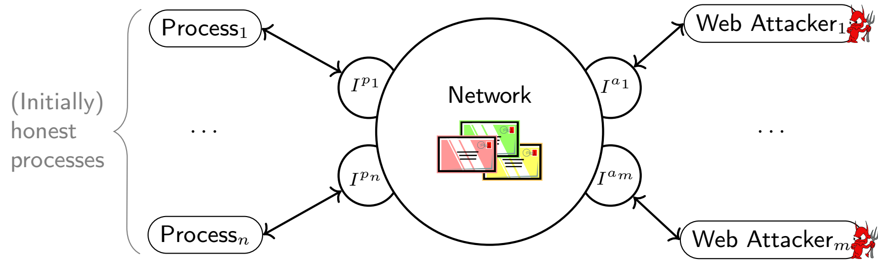
For full definitions of processes, systems, and runs, we refer the reader to Section 2.3, for Web systems to Section 2.11.

1.5 Attackers

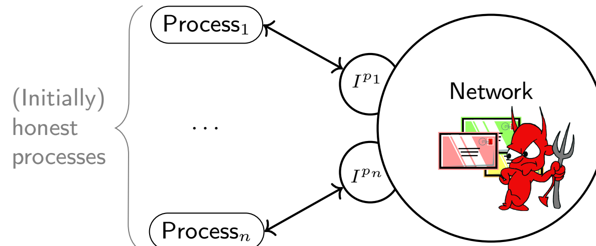
Attackers are modeled as processes in the WIM. An attacker is a Dolev-Yao process, which records all messages it receives and outputs any finite sequence of events that it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. We distinguish two types of attackers: *Web attackers* and *network attackers*. Both types are illustrated in Figure 2. Web attackers participate in the network as any other process, i.e., they can receive events that are addressed to the respective Web attacker process, and they can send events to arbitrary receivers. A network attacker[5] essentially controls the network. The network attacker can not only do the same actions as a Web attacker but can also receive messages

[4] If multiple processes are associated with the same (IP) address, one of these processes is selected non-deterministically.

[5] Note that one network attacker can subsume any number of network attackers and Web attackers.



(a) Web attackers participate in the same way as any other process.



(b) A network attacker controls the network and can intercept and spoof events.

Figure 2: Illustration of different attackers in the WIM.

that are not addressed to it and perform IP spoofing. As the delivery of events is non-deterministic, all cases of a network attacker intercepting or blocking a message are captured.[6]

As already mentioned above, attackers can also corrupt other processes, e.g., browsers. If an honest process receives a corrupt message, it effectively turns into a Web attacker process that is given the process' state including secret values, such as credentials (passwords), cookies, etc.[7] Note that in security analyses, we typically require that certain processes do not become corrupted, while all other parties might be dishonest.

For the formal definition of attacker processes, we refer the reader to Section 2.5. A mechanism for malicious Web pages opened in a browser is covered below.

1.6 Browsers and Scripts

In the WIM, a browser is a specific kind of DY process that mimics the behavior of a real-world Web browser. In Section 2.7, we provide the full definition of a browser's state and relation.

A browser's state includes (among others) a list of open windows, cookies, and Web storage (localStorage and sessionStorage). A *window* inside a browser's state contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window (see Figure 3 for an illustration). Each document represents one loaded Web page and, again, includes a list of windows, creating a tree of windows and documents.

Each document contains a *script*[8] and a state of this script (*scriptstate*). A script is a relation that models the behavior of one Web page and subsumes all components of this Web page including all external resources, such as external JavaScript files. Each document and its script behave similar to processes: Whenever a browser (during a processing step) decides to run a script in a document, the browser provides the script with its scriptstate (stored in the document) and a limited view on the browser's state (based on the document)[9] and expects to receive a command in return along with an updated scriptstate. This way, scripts can navigate or create windows, send XMLHttpRequests and postMessages, submit forms, set/change cookies and Web storage data, and create iframes.

[6]Note that an analysis in the WIM typically reasons about all possible runs of a system.

[7]In our security analyses of SSO protocols, an attacker initially does not have any credentials for any account at (honest) IdPs. Using corruption, the attacker can gain access to such credentials, modeling that the attacker can use accounts of (now dishonest) users at an IdP.

[8]More precisely, a document contains a script identifier which refers to a script relation.

[9]The view of a script includes a (limited) view on other documents and windows, certain cookies, Web storage data, and certain user credentials.

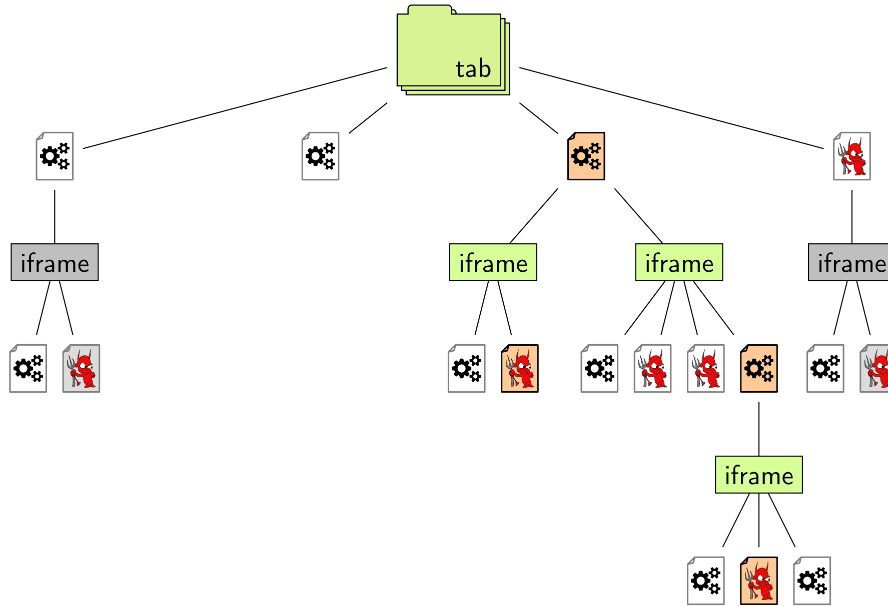


Figure 3: Illustration of the window and document structure inside a browser. The depicted tree shows an example of the structure of one browser tab. Active documents are highlighted in orange.

Scripts can be either honest, in which case they model the behavior of the modeled application, or scripts can be dishonest, in which case they can derive any possible output, i.e., any output that is derivable from the script's input. We subsume all dishonest scripts in the so-called *attacker script* (R^{att}). Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the relevant Web standards. See Section 1.6.2 below for more details on scripts.

One browser is thought to be used by one user, who is modeled as part of the browser. User actions, such as following a link, are modeled as non-deterministic actions of the Web browser and scripts. User credentials are stored in the initial state of the browser annotated by an origin and are given to scripts according to the origin under which the script is running.

The browser relation takes care of processing many different types of messages, i.e., trigger messages, DNS responses, HTTP(S) responses, and corruption messages. As a result, the relation can then output messages such as DNS and HTTP(S) requests (including XHRs). When handling HTTP(S) messages, the relation takes care of important HTTP(S) headers, for example, cookie, location, strict transport security (STS), and origin headers.

Before we describe the handling of each of these messages in detail, we will first give a description of the dispatching mechanism for HTTP(S) requests, which is one of the browser's core functionality and spread across multiple processing steps.

1.6.1 Dispatchment of HTTP(S) Requests

If a browser wants to send out some HTTP(S) request (either as a result of a script, a URL bar action, or a reload), the browser prepares the (term of the) HTTP(S) request. Before sending this HTTP(S) request out, the browser first performs a DNS lookup for the domain to which the request is addressed to (recall that in the network model, all messages need to be addressed to some (IP) address, which is not a domain). For this DNS lookup, the browser creates an event that is addressed to some DNS server and that contains a DNS request message for the domain. The DNS server is determined by the browser's state, i.e., the browser is configured to send DNS requests to some (IP) address. The browser assigns the DNS request a fresh nonce as its message id. In the browser's state (namely in a subterm called *pendingDNS*), the browser records that it expects a DNS response that bears this message id. Further, along with this message id, the browser stores the HTTP(S) request that is to be sent out as well as a term *reference* that will be used to process the HTTP(S) response to this HTTP(S) request. The reference can be of two different kinds: (1)

If the HTTP(S) request is an XHR (caused by a script), then the reference contains an identifier for the script's document as well as some term determined by the script (the script will later use this term to match the response to the request), or (2) in any other case, the reference contains an identifier for the window that will consume the HTTP(S) response, i.e., that will load the response's body as a new document. Finally, the browser outputs the event containing the DNS request and its modified state.

When the browser concludes the DNS lookup, i.e., when the browser receives a matching DNS response, it sends out the actual HTTP(S) request. At the same time, the browser moves the corresponding entry in the browser's state from the subterm *pendingDNS* to a subterm called *pendingRequests*. Entries in this subterm are used to match the corresponding HTTP(S) response when it is received by the browser in a later processing step (see also Section 1.6.3 below).

In the case of HTTPS messages, the browser further takes care of encrypting the request. To this end, the browser looks up the public key of the receiving domain and chooses a fresh nonce as the (symmetric) key for the response (see also our explanation of HTTPS in Section 1.3 above). The nonce is stored along with the *reference* explained above.

Finally, if the browser receives a matching HTTP(S) response, it removes the corresponding entry from *pendingRequests* and processes the content of the message (see Section 1.6.4 below for more details).

1.6.2 Handling of Trigger Messages

A browser, at any time, can receive a trigger message upon which the browser non-deterministically chooses one of the following actions:

- Running a script in some document,
- mimic a user entering a URL in the location bar,
- reload a window, or
- navigate a window forward or backward.

We will describe each of these action in more detail below.

Running a script in some document The browser non-deterministically selects a window and runs the script of the document contained within this window. As already mentioned above, the script subsumes the behavior of (a) a real-world HTML document that contains JavaScript and (b) user interaction with this document. Similar to process relations, a script is also non-deterministic and can use nonces by providing placeholders that will be filled with nonces by the process transition. The script relation maps from

- a limited view on the browser's window structure based on the document,
- an identifier of the current document in that window structure,
- the script's current *scriptstate*, a term that is stored in the document and that is used to store data of the script between different runs of that script (i.e., to keep state),
- a sequence of *script inputs*, a term that is stored in the document and that is used to store XMLHttpRequest responses and postMessages for this document,
- a sequence of cookies for the document's domain (which are not marked as HTTPonly),
- a dictionary containing the localStorage of the document's origin within the browser's state,
- a dictionary containing the sessionStorage of the document's origin within the current window tree,
- a sequence of ids of the browser's user, and
- a sequence of secrets (passwords) of the document's origin within the browser's state.[10]

to

- a new scriptstate that will be stored in the document,
- a new set of cookies for the current domain,
- a new dictionary for the localStorage as above,
- a new dictionary for the sessionStorage as above, and
- a command.

Commands issued by a script can be:

- `HREF`: The script outputs a URL and a window reference. This command models the user clicking on a link or a JavaScript navigating some window.

[10]Note that we model a user who is at least cautious about at which origin she enters passwords.

- `IFRAME`: The script outputs a URL and a window reference. This command models the script creating an iframe in the indicated window.
- `FORM`: The script outputs a URL, an HTTP method (GET or POST), form data, and a window reference. This models a user (or the JavaScript itself) filling out a form and submitting this form.
- `SETSCRIPT`: The script outputs a window reference and a script identifier. This command models the script replacing the content of the document of the indicated window.
- `SETSCRIPTSTATE`: The script outputs a window reference and a (second) scriptstate. This command models the script changing variables of JavaScript running in the document of the indicated window.
- `XMLHTTPREQUEST`: The script outputs a URL, an HTTP method, a term for the body of the HTTP request, and some term that will be used as a reference to match the response to this request at a later point in time. The response to an XMLHttpRequest will be later added to the script inputs of the current document.
- `BACK`, `FORWARD`, or `CLOSE`: The script outputs a window reference. This command models the script navigating the indicated window backward or forward, or closing the window. The navigation over the history of a window changes a flag that marks the currently active document in the window's history.
- `POSTMESSAGE`: The script outputs a window reference, a message term, and an origin. This command models the script sending a postMessage to the indicated window. (The origin is used to restrict the delivery of this message to documents of this origin.) The postMessage will be added to the active document's script inputs in the indicated window.

A browser immediately processes the command output by the script (within the same processing step). For most commands, certain restrictions apply. For example, a script may only instruct the browser to replace a script by the SETSCRIPT command if the target document is of the same origin as the current document. A browser ignores invalid or forbidden commands.

Some commands, however, require interaction with the network. The browser relation then prepares the HTTP request, outputs the corresponding DNS request and stores the HTTP request along with some other metadata (e.g., which document/window, XMLHttpRequest reference, ...) to later continue processing the command when the DNS response is delivered.

Mimic a user entering a URL in the location bar The browser non-deterministically decides whether it creates a new (top-level) window. If it does not create a new window, it non-deterministically selects some existing top-level window. The browser also non-deterministically chooses a URL (that does not contain any nonces and hence no secrets). For the selected (new or existing) window, the browser behaves as if an (imaginary) script of this window (more precisely of the document within that window, assuming such a document exists) issued the `HREF` command for the chosen URL and a reference for this window.

Reload a window The browser non-deterministically selects some window and navigates this window to its current document's URL similar to the `HREF` command as described above. This choice models a user's click on the reload button in her browser's navigation bar or of the context menu of some window (which might be an iframe).

Navigate a window forward or backward The browser non-deterministically selects some window and then behaves similar to the `BACK` and `FORWARD` commands as described above. This mimics a user's click on the backward or forward button in her browser's navigation bar or the context menu of some window (which again might be an iframe).

1.6.3 Handling of DNS Responses

When the browser receives an event containing a DNS response, it looks up in its state whether it expects such a response. If the browser does not find an entry in *pendingDNS*, it ignores the DNS response. If the browser finds an entry, the browser considers the DNS lookup to be finished and removes this entry from *pendingDNS*. Now the actual HTTP request will be sent out. The browser checks whether the HTTP request is to be sent to an HTTPS URL, i.e., the browser determines whether the original message needs to be encrypted. As mentioned above, the browser chooses a

fresh nonce that will be used as a symmetric key to encrypt the HTTP response later and encrypts the HTTP request along with this nonce asymmetrically with the public key of the domain (taken from the browser's state). The browser now takes this encrypted term[11] or the plain HTTP request (depending on whether the URL is HTTPS or not) as the actual message and creates a new event with this message that is addressed to the (IP) address contained in the DNS response. Further, the browser creates a new entry in its state in the subterm *pendingRequests*. This entry essentially contains the same information as the former entry in *pendingDNS* (HTTP request, reference) plus the (IP) address and (in the case of HTTPS) the nonce that will be used as the symmetric key to encrypt the response. Finally, the browser outputs the event containing the HTTP request as well as its modified state.

1.6.4 Handling of HTTP(S) Responses

When the browser receives an HTTP(S) response, it uses the subterm *pendingRequests* in its state to look up whether it expects such a response. If it does not expect such an event, the browser ignores it. Otherwise, the browser finds an entry in this subterm that contains all necessary information to process this response (including a symmetric key to decrypt the response in the case of HTTPS). Similar to DNS responses, the browser removes this entry from *pendingRequests*. The browser now checks whether the HTTP(S) response contains a redirect (i.e., whether the response contains a redirect status code) and continues as described below.

The HTTP response does not contain a redirect As above, when the browser created the corresponding request, the browser again has to distinguish two cases:

- The response is for an XMLHttpRequest. In this case, the browser uses the reference term contained in the entry in *pendingRequests* to retrieve the document's identifier and the script's reference term that was used when a script instructed the browser to send out the request. The browser looks up the document in its window structure and appends the script's reference along with the body and selected headers of the HTTP response to the list of script inputs. The script inputs can be used by the script running within this document when it is triggered in a later processing step.
- The response is for any other kind of request. In this case, the browser expects the HTTP response's body to contain a script identifier and an initial state for this script. The browser creates a new document with this information. This document is then appended to the history of the window identified by the window identifier contained in the entry in *pendingRequests*. This document is also marked as being the active document of this window.

The HTTP response contains a redirect In this case, the browser follows the redirect except if the corresponding HTTP request was caused by an XMLHttpRequest (the browser knows this from the reference term of the entry that it took from *pendingRequests*). Following a redirect means that the browser behaves similar to sending the HTTP request in the first place: The browser creates an event with a DNS request for the domain of the URL it is redirected to and creates an entry in *pendingDNS* that uses the same reference term as for the first HTTP request (this term is contained in the entry the browser took from *pendingRequests*. Depending on the type of the redirect (the WIM implements the redirect codes 303 and 307, which subsume the behavior of all kinds of redirect codes), the request's body is dropped.[12]

1.6.5 Corruption

A browser can also become corrupted as explained above. We model two types of corruption of browsers, namely *full corruption* and *close-corruption*, both of which are triggered by special network messages in the WIM. In the real world, an attacker can exploit buffer overflows in Web browsers, compromise operating systems (e.g., using Trojan horses), and physically take control over shared internet terminals.

[11] Note that we only consider symbolic encryption in the WIM.

[12] If the HTTP request was a POST request, the request might contain request parameters in its body. In the case of a 307 redirect, the new request remains the same as the old request except for the URL it is directed to and the *Origin* HTTP header. In the case of a 303 redirect, the request is changed to a GET request, and the body is removed.

Full corruption models an attacker that gained full control over a Web browser and its user. Besides modeling a compromised system, full corruption can also serve as a vehicle for the attacker to participate in a protocol using secrets of honest browsers (think of the attacker "recruiting" collaborators).

Close-corruption models a browser that is taken over by the attacker after a user finished her browsing session, i.e., after closing all windows of the browser. This form of corruption is relevant in situations where one browser can be used by many people, e.g., in an Internet café or in a hotel lobby. Information left in the browser state after closing the browser could be misused by malicious users.

1.7 Web Servers

Web servers are application-specific processes. The primary function of a Web server is to respond to HTTP(S) requests (with HTTP(S) responses). A Web server might also send out HTTP requests on its own (e.g., invoked by a trigger message) and process responses to its requests.

As Web servers model application-specific behavior, they need to be defined depending on the respective application. To ease the definition of the behavior of Web servers, the WIM provides a generic server template in Section 2.12 on Page 32. This template provides basic functionality to handle incoming HTTPS requests as well as sending out HTTPS requests. To model application logic in this framework, several "dummy" functions can be replaced with application specific behavior (i.e., pseudo code).

Note that a Web server can typically also become corrupted as explained above.

1.8 DNS Servers

DNS servers respond to DNS requests, which ask for the IP address of some domain. The DNS server looks up this IP address and responds with a DNS response. Here, we consider a flat DNS model in which DNS queries are answered directly by one DNS server and always with the same address for a domain. Hence, DNS servers contain a list of assignments of domain names to IP addresses in their state and use only their state to look up the result of a DNS query, i.e., they do not perform further DNS resolving by forwarding the DNS request to another DNS server. A full (hierarchical) DNS system with recursive DNS resolution, DNS caches, etc. could also be modeled to cover specific attacks on the DNS system itself.

Here, we only model plain DNS and do not consider recent improvements and extensions such as DNSSEC [16] and DNS over HTTPS [14], which provide integrity (in the case of DNSSEC) and confidentiality (in the case of DNS over HTTPS). This is motivated by the fact that these technologies are not mandatory and — in the worst case — not used at all. Hence, omitting these techniques is a safe overapproximation. In a system that contains a network attacker, we typically do not consider honest DNS servers but configure all parties to use the attacker as their DNS server. In such a system, the attacker is always able to intercept and respond to all DNS messages and hence, we do not get any guarantees from an honest DNS server. DNS servers might also become corrupted, giving Web attackers also the ability to interfere with DNS resolution.

1.9 Limitations

Of course, a model cannot reflect all aspects of the real world as a model is always an abstraction. As such, the WIM also abstracts away some aspects: There are no details of programming languages, such as JavaScript, or byte-level attacks, such as buffer overflows. Still, the WIM can capture the outcome of such attacks by using dynamic corruption or the attacker script within the browser. The WIM also omits user interface details, which might miss user-level attacks such as Clickjacking. Further, the WIM, as a Dolev-Yao style model, uses an abstract view on cryptography and TLS. Still, the WIM provides us with a comprehensive view on the logic of interactions between different entities, or even scripts within a browser, and allows analyses to yield meaningful results for this abstraction level.

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \quad (1)$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x \quad (2)$$
$$\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) = \top \quad (3)$$
$$\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x \quad (4)$$
$$\mathsf{checkmac}(\mathsf{mac}(x, y), y) = \top \quad (5)$$
$$\mathsf{extractmsg}(\mathsf{mac}(x, y)) = x \quad (6)$$
$$\pi_i(\langle x_1, \dots, x_n\rangle) = x_i \;\text{ if } 1 \le i \le n \quad (7)$$
$$\pi_j(\langle x_1, \dots, x_n\rangle) = \Diamond \;\text{ if } j \notin \{1, \dots, n\} \quad (8)$$
$$\pi_j(t) = \Diamond \;\text{ if } t \text{ is not a sequence} \quad (9)$$

Figure 4: Equational theory for Σ.

2 Technical Definitions

Here, we provide technical definitions that complete our description of the WIM in Section 1. As mentioned in the introduction, we follow the descriptions in [6, 7, 9, 10, 11, 12, 13].

2.1 Terms and Notations

Definition 1 (Nonces and Terms). By $X = \{x_0, x_1, \dots\}$ we denote a set of variables and by $\mathcal{N}$ we denote an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol for some $n \ge 0$ and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term. $\diamond$

By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by the theory associated with Σ (see Figure 4). 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 2 (Ground Terms, Messages, Placeholders, Protomessages). By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote 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. $\diamond$

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 3 (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.). $\diamond$

Definition 4 (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 Figure 4. For a term t, we denote its normal form as $t{\downarrow}$. $\diamond$

Definition 5 (Pattern Matching). Let $\mathit{pattern} \in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches* $\mathit{pattern}$ iff t can be acquired from $\mathit{pattern}$ by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim \mathit{pattern}$. For a sequence of patterns $\mathit{patterns}$ we write $t \dot{\sim} \mathit{patterns}$ to denote that t matches at least one pattern in $\mathit{patterns}$.

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

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

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

Definition 6 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$, and $t_1, \dots, t_n \in \mathcal{T}_N$.

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 $i \in \{1, \dots, n\}$. $\diamond$

Definition 7 (Sequence Notations). For a sequence $t = \langle t_1, \dots, t_n \rangle$ and a set s we use $t \subset^{\langle\rangle} s$ to say that $t_1, \dots, t_n \in s$. We define $x \in^{\langle\rangle} t \iff \exists i : t_i = x$. For a term y we write $t +^{\langle\rangle} y$ to denote the sequence $\langle t_1, \dots, t_n, y \rangle$. For a sequence $r = \langle r_1, \dots, r_m \rangle$ we write $t \cup r$ to denote the sequence $\langle t_1, \dots, t_n, r_1, \dots, r_m \rangle$. We define $|t| = n$. If t is not a sequence, we set $|t| = \Diamond$. For a finite set M with $M = \{m_1, \dots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \dots, m_n \rangle$. (The order of the elements does not matter; one is chosen arbitrarily.) $\diamond$

Definition 8 (Dictionaries). A *dictionary over X and Y* is a term of the form

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

where $k_1, \dots, k_n \in X$, $v_1, \dots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \dots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$. $\diamond$

We note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$. Figure 5 shows the short notation for dictionary operations. For a dictionary $z = [k_1 : v_1, k_2 : v_2, \dots, k_n : v_n]$ we write $k \in z$ to say that there exists i such that $k = k_i$. We write $z[k_j]$ to refer to the value v_j. (Note that if a dictionary contains two elements $\langle k, v \rangle$ and $\langle k, v' \rangle$, then the notations and operations for dictionaries apply non-deterministically to one of both elements.) If $k \notin z$, we set $z[k] := \langle\rangle$.

$$[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n]\,[k_i] = v_i \tag{10}$$

$$\begin{aligned}[k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1} \dots, k_n : v_n] - k_i = \\ [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1} \dots, k_n : v_n]\end{aligned} \tag{11}$$

Figure 5: Dictionary operators with $1 \leq i \leq n$.

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

Definition 9 (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. $\diamond$

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 $o.$`protocol` instead of $\pi_2(o)$ or $o.2$. See also Example 4.

Definition 10 (Concatenation of Sequences). For a sequence $a = \langle a_1, \dots, a_i \rangle$ and a sequence $b = \langle b_1, b_2, \dots \rangle$, we define the *concatenation* as $a \cdot b := \langle a_1, \dots, a_i, b_1, b_2, \dots \rangle$.
$\diamond$

Definition 11 (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. $\diamond$

2.2 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.

2.2.1 URLs

Definition 12. A *URL* is a term of the form

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

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

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 $\mathsf{URL}^{\mathit{host}}_{\mathit{path}}$ to denote the URL $\langle \mathtt{URL}, \mathtt{S}, \mathit{host}, \mathit{path}, \langle\rangle, \bot \rangle$.

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 9):

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

2.2.2 Origins

Definition 13. An *origin* is a term of the form $\langle \mathit{host}, \mathit{protocol} \rangle$ with $\mathit{host} \in \mathsf{Doms}$ and $\mathit{protocol} \in \{\mathtt{P}, \mathtt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins. $\diamond$

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

2.2.3 Cookies

Definition 14. A *cookie* is a term of the form $\langle \mathit{name}, \mathit{content} \rangle$ where $\mathit{name} \in \mathcal{T}_{\mathcal{N}}$, and *content* is a term of the form $\langle \mathit{value}, \mathit{secure}, \mathit{session}, \mathit{httpOnly} \rangle$ where $\mathit{value} \in \mathcal{T}_{\mathcal{N}}$, *secure*, *session*, $\mathit{httpOnly} \in \{\top, \bot\}$. 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., $\mathit{name}.1$) 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 $\mathcal{T}_{\mathcal{N}}(V)$. $\diamond$

If the *secure* attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections.[13] 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 $\mathit{name}.1 \equiv \mathtt{__Host}$), the browser accepts the cookie only if the *secure* attribute is set. As such cookies are only transferred over secure channels (i.e., with TLS), the cookie cannot be set by a network attacker. Note that the WIM does not model the domain attribute of the Set-Cookie header, so cookies in the WIM are always sent to the originating domain and not some subdomain. Therefore, the WIM models only the __Host prefix, but not the __Secure prefix.

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

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

2.2.4 HTTP Messages

Definition 15. An *HTTP request* is a term of the form shown in (12). An *HTTP response* is a term of the form shown in (13).

$$\langle \texttt{HTTPReq}, \mathit{nonce}, \mathit{method}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \tag{12}$$
$$\langle \texttt{HTTPResp}, \mathit{nonce}, \mathit{status}, \mathit{headers}, \mathit{body}\rangle \tag{13}$$

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 $c \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ (note that in this header, only names and values of cookies are transferred),
  - $\langle \texttt{Location}, l\rangle$ 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 [15]),
  - $\langle \texttt{ReferrerPolicy}, p\rangle$ 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).

$$\begin{aligned} r :=& \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, \texttt{/show}, \langle\langle \texttt{index}, \texttt{1}\rangle\rangle, \\ & [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S}\rangle], \langle \texttt{foo}, \texttt{bar}\rangle\rangle \end{aligned} \tag{14}$$
$$s := \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top\rangle\rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x\rangle\rangle \tag{15}$$

An HTTP `POST` request for the URL `http://example.com/show?index=1` is shown in (14), with an Origin header and a body that contains $\langle \texttt{foo}, \texttt{bar}\rangle$. A possible response is shown in (15), 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 2.11) 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 16. 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.

$$\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{16}$$
$$\mathsf{enc_s}(s, k') \tag{17}$$

The term (16) shows an encrypted request (with r as in (14)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (17) is a response (with s as in (15)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (16).

2.2.5 DNS Messages

Definition 17. A *DNS request* is a term of the form $\langle \mathtt{DNSResolve}, \mathit{domain}, \mathit{nonce}\rangle$ where $\mathit{domain} \in \mathsf{Doms}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$. ◇

Definition 18. A *DNS response* is a term of the form $\langle \mathtt{DNSResolved}, \mathit{domain}, \mathit{result}, \mathit{nonce}\rangle$ 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.

2.3 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 19 (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 20 (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[14] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$. ◇

Definition 21 (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 $(v_1, v_2, \dots, v_i)$ of all placeholders appearing in E^ν_{out} (ordered lexicographically),
4. a sequence $N^\nu = (\eta_1, \eta_2, \dots, \eta_i)$ of the first i elements in N

with

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

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

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

Intuitively, for a processing step, we select one of the processes in $\mathcal{P}$, and call it with one of the events in the list of waiting events E. In its output (new state and output events), we replace any 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 22 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0* is a finite sequence of configurations $((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \dots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

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

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

2.4 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 23 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with placeholders V* if there exist $n \geq 0$, $m_1, \dots, m_n \in M$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \dots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \dots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V. ⋄

For example, the term a can be derived from the set of terms $\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\}$, 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 24 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s^p_0)$ 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, $s' \in d_{V_{\text{process}}}(\{e, s\})$. ⋄

2.5 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 25 (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 $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_1, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e, s\})$. $\diamond$

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.

2.6 Notations for Functions and Algorithms

When describing algorithms, we use the following notations:

2.6.1 Non-deterministic choosing and iteration

The notation **let** $n \leftarrow N$ is used to describe that n is chosen non-deterministically from the set N. We write **for each** $s \in M$ **do** to denote that the following commands (until **end for**) 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.

2.6.2 Function calls

When calling functions that do not return anything, we write

call $\mathsf{FUNCTION_NAME}(x, y)$

to describe that a function $\mathsf{FUNCTION_NAME}$ 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 $z := \mathsf{FUNCTION_NAME}(x, y)$

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.

2.6.3 Stop without output

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

2.6.4 Placeholders

In several places throughout the algorithms presented next we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 1). Table 1 shows a list of all placeholders used.

2.6.5 Abbreviations for URLs and Origins

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

Placeholder	Usage
ν_1	Algorithm 9, new window nonces
ν_2	Algorithm 9, new HTTP request nonce
ν_3	Algorithm 9, lookup key for pending HTTP requests entry
ν_4	Algorithm 7, new HTTP request nonce (multiple lines)
ν_5	Algorithm 7, new subwindow nonce
ν_6	Algorithm 8, new HTTP request nonce
ν_7	Algorithm 8, new document nonce
ν_8	Algorithm 4, lookup key for pending DNS entry
ν_9	Algorithm 1, new window nonce
$\nu_{10}, \dots$	Algorithm 7, replacement for placeholders in script output

Table 1: List of placeholders used in browser algorithms.

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 $\mathsf{origin}_\mathsf{P}$.

2.7 Browsers

Following the informal description of the browser model in Section 1, we now present the formal model of browsers.

2.7.1 Scripts

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

Definition 26 (Placeholders for Scripts). By $V_\text{script} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts. $\diamond$

Definition 27 (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 $s' \in d_{V_\text{script}}(s)$. $\diamond$

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

2.7.2 Web Browser State

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

Definition 29. A *window* is a term of the form $w = \langle \mathit{nonce}, \mathit{documents}, \mathit{opener} \rangle$ with $\mathit{nonce} \in \mathcal{N}$, $\mathit{documents} \subset^{\langle\rangle} \mathsf{Documents}$ (defined below), $\mathit{opener} \in \mathcal{N} \cup \{\bot\}$ where $d.\mathtt{active} = \top$ for exactly one $d \in^{\langle\rangle} \mathit{documents}$ if *documents* is not empty (we then call d the *active document of* w). We write $\mathsf{Windows}$ for the set of all windows. We write $w.\mathtt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else. $\diamond$

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., $b.\mathtt{opener} = a.\mathtt{nonce}$.

Definition 30. A *document* d is a term of the form

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

where $nonce \in \mathcal{N}$, $location \in \mathsf{URLs}$, $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $referrer \in \mathsf{URLs} \cup \{\bot\}$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $subwindows \subset^{\langle\rangle} \mathsf{Windows}$, $active \in \{\top, \bot\}$. 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 $d.\mathtt{origin}$ to denote the origin of the document, i.e., the term $\langle d.\mathtt{location.host}, d.\mathtt{location.protocol}\rangle \in \mathsf{Origins}$. ◇

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

Definition 31. For two window terms w and w' we write

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

if $w \in^{\langle\rangle} w'.\mathtt{activedocument.subwindows}$. 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 32. A reference for a normal HTTP(S) request is a sequence of the form $\langle \mathtt{REQ}, nonce\rangle$, where $nonce$ is a window reference. A reference for a XMLHttpRequest is a sequence of the form $\langle \mathtt{XHR}, nonce, xhrreference\rangle$, 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 8.

Definition 33. 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).
- $secrets \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ 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.
- $localStorage \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ stores the data saved by scripts using the localStorage API (separated by origins).
- $sessionStorage \in \left[OR \times \mathcal{T}_{\mathcal{N}}\right]$ for $OR := \{\langle o, r\rangle | \, o \in \mathsf{Origins},\ r \in \mathcal{N}\}$ similar to localStorage, but the data in sessionStorage is additionally separated by top-level windows.

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

2.7.3 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 $\mathsf{Subwindows}$, Docs, Clean, $\mathsf{CookieMerge}$, $\mathsf{AddCookie}$, and $\mathsf{NavigableWindows}$.

Subwindows and Docs. Given a browser state s, $\mathsf{Subwindows}(s)$ denotes the set of all pointers[15] to windows in the window list $s.\mathtt{windows}$ and (recursively) the subwindows of their active documents. We exclude subwindows of inactive documents and their subwindows. With $\mathsf{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\mathsf{Subwindows}(s)$.

Definition 34. For a browser state s we denote by $\mathsf{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\mathtt{windows}$ there is a $\overline{p} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p} = w$. (2) For all $\overline{p} \in \mathsf{Subwindows}(s)$, the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p'} = w$.

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

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

Definition 35. Let s be a browser state and d a document. By $\mathsf{Clean}(s, d)$ we denote the term that equals $s.\mathtt{windows}$ 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 $\mathtt{headers}$ for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.) $\diamond$

CookieMerge. The function $\mathsf{CookieMerge}$ 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.

[15] Recall the definition of a pointer in Definition 9.

Algorithm 1 Web Browser Model: Determine window for navigation.

```
1: function GETNAVIGABLEWINDOW(w̄, window, noreferrer, s')
2:     if window ≡ _BLANK then    → Open a new window when _BLANK is used
3:         if noreferrer ≡ ⊤ then
4:             let w' := ⟨ν_9, ⟨⟩, ⊥⟩
5:         else
6:             let w' := ⟨ν_9, ⟨⟩, s'.w̄.nonce⟩
7:         let s'.windows := s'.windows +⟨⟩ w'
            ↪ and let w̄' be a pointer to this new element in s'
8:         return w̄'
9:     let w̄' ← NavigableWindows(w̄, s') such that s'.w̄'.nonce ≡ window
        ↪ if possible; otherwise return w̄
10:    return w̄'
```

Definition 36. For a sequence of cookies (with pairwise different names) *oldcookies*, a sequence of cookies *newcookies*, and a string *protocol* $\in \{\mathtt{P}, \mathtt{S}\}$, the set $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies}, \mathit{protocol})$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\mathtt{content}.\mathtt{httpOnly} \equiv \top$ or where $(c.\mathtt{name}.1 \equiv \mathtt{__Host}) \wedge ((\mathit{protocol} \equiv \mathtt{P}) \vee (c.\mathtt{secure} \equiv \bot))$. For any c, $c' \in^{\langle\rangle}$ *newcookies*, $c.\mathtt{name} \equiv c'.\mathtt{name}$, 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*, $c_{\text{old}}.\mathtt{name} \equiv c_{\text{new}}.\mathtt{name}$, add c_{new} to m if $c_{\text{old}}.\mathtt{content}.\mathtt{httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies}, \mathit{protocol})$ 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 37. For a sequence of cookies (with pairwise different names) *oldcookies*, a cookie c, and a string *protocol* $\in \{\mathtt{P}, \mathtt{S}\}$ (denoting whether the HTTP response was received from an insecure or a secure origin), the sequence $\mathsf{AddCookie}(\mathit{oldcookies}, c, \mathit{protocol})$ is defined by the following algorithm: Let $m := \mathit{oldcookies}$. If $(c.\mathtt{name}.1 \equiv \mathtt{__Host}) \wedge \neg((\mathit{protocol} \equiv \mathtt{S}) \wedge (c.\mathtt{secure} \equiv \top))$, then return m, else: Remove any c' from m that has $c.\mathtt{name} \equiv c'.\mathtt{name}$. 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 [2], Section 5.1.4 for this definition.

Definition 38. 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 $s'.\overline{w'}.\mathtt{activedocument}.\mathtt{origin} \equiv s'.\overline{w}.\mathtt{activedocument}.\mathtt{origin}$ (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 $s'.\overline{w'} \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\ s'.\overline{p}.\mathtt{activedocument}.\mathtt{origin} = s'.\overline{w}.\mathtt{activedocument}.\mathtt{origin}$ ($\overline{w'}$ 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 $s'.\overline{w'}.\mathtt{opener} = s'.\overline{p}.\mathtt{nonce} \wedge \overline{p} \in \overline{W}$ ($\overline{w'}$ 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 $\overline{w'} \in \overline{W}$.

◇

Functions

- The function GETNAVIGABLEWINDOW (Algorithm 1) 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

Algorithm 2 Web Browser Model: Determine same-origin window.

1: **function** GETWINDOW($\overline{w}$, $window$, s')
2: **let** $\overline{w'} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w'}.\mathtt{nonce} \equiv window$
$\hookrightarrow$ **if possible; otherwise return** $\overline{w}$
3: **if** $s'.\overline{w'}.\mathtt{activedocument.origin} \equiv s'.\overline{w}.\mathtt{activedocument.origin}$ **then**
4: **return** $\overline{w'}$
5: **return** $\overline{w}$

Algorithm 3 Web Browser Model: Cancel pending requests for given window.

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

Algorithm 4 Web Browser Model: Prepare headers, do DNS resolution, save message.

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

Algorithm 5 Web Browser Model: Navigate a window backward.

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

Algorithm 6 Web Browser Model: Navigate a window forward.

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

Algorithm 7 Web Browser Model: Execute a script.

```
1: function RUNSCRIPT(w̄, d̄, a, s′)
2:     let tree := Clean(s′, s′.d̄)
3:     let cookies := ⟨{⟨c.name, c.content.value⟩ | c ∈⟨⟩ s′.cookies [s′.d̄.origin.host]
       ↪  ∧ c.content.httpOnly ≡ ⊥
       ↪  ∧ (c.content.secure ≡ ⊤ ⟹ (s′.d̄.origin.protocol ≡ S))}⟩
4:     let tlw ← s′.windows such that tlw is the top-level window containing d̄
5:     let sessionStorage := s′.sessionStorage [⟨s′.d̄.origin, tlw.nonce⟩]
6:     let localStorage := s′.localStorage [s′.d̄.origin]
7:     let secrets := s′.secrets [s′.d̄.origin]
8:     let R := script⁻¹(s′.d̄.script) if possible; otherwise stop
9:     let in := ⟨tree, s′.d̄.nonce, s′.d̄.scriptstate, s′.d̄.scriptinputs, cookies,
       ↪  localStorage, sessionStorage, s′.ids, secrets⟩
10:    let state′ ← T_N(V_process), cookies′ ← Cookies^ν, localStorage′ ← T_N(V_process),
       ↪  sessionStorage′ ← T_N(V_process), command ← T_N(V_process),
       ↪  out := ⟨state′, cookies′, localStorage′, sessionStorage′, command⟩
       ↪  such that out := out^λ[ν_10/λ_1, ν_11/λ_2, ...] with (in, out^λ) ∈ R
11:    let s′.cookies [s′.d̄.origin.host] :=
       ↪  ⟨CookieMerge(s′.cookies [s′.d̄.origin.host], cookies′, s′.d̄.origin.protocol)⟩
12:    let s′.localStorage [s′.d̄.origin] := localStorage′
13:    let s′.sessionStorage [⟨s′.d̄.origin, tlw.nonce⟩] := sessionStorage′
14:    let s′.d̄.scriptstate := state′
15:    let referrer := s′.d̄.location
16:    let referrerPolicy := s′.d̄.headers[ReferrerPolicy]
17:    let docorigin := s′.d̄.origin
18:    switch command do
19:        case ⟨HREF, url, hrefwindow, noreferrer⟩
20:            let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, noreferrer, s′)
21:            let reference := ⟨REQ, s′.w̄′.nonce⟩
22:            let req := ⟨HTTPReq, ν_4, GET, url.host, url.path, url.parameters, ⟨⟩, ⟨⟩⟩
23:            if noreferrer ≡ ⊤ then
24:                let referrerPolicy := noreferrer
25:            let s′ := CANCELNAV(reference, s′)
26:            call HTTP_SEND(reference, req, url, ⊥, referrer, referrerPolicy, a, s′)
27:        case ⟨IFRAME, url, window⟩
28:            if window ≡ _SELF then
29:                let w̄′ := w̄
30:            else
31:                let w̄′ := GETWINDOW(w̄, window, s′)
32:            let req := ⟨HTTPReq, ν_4, GET, url.host, url.path, url.parameters, ⟨⟩, ⟨⟩⟩
33:            let w′ := ⟨ν_5, ⟨⟩, ⊥⟩
34:            let s′.w̄′.activedocument.subwindows := s′.w̄′.activedocument.subwindows +⟨⟩ w′
35:            call HTTP_SEND(⟨REQ, ν_5⟩, req, url, ⊥, referrer, referrerPolicy, a, s′)
```

36: **case** $\langle \texttt{FORM}, url, method, data, hrefwindow \rangle$

37: **if** $method \notin \{\texttt{GET}, \texttt{POST}\}$ **then**

38: **stop**

39: **let** $\overline{w'} :=$ GETNAVIGABLEWINDOW$(\overline{w},\ hrefwindow,\ \bot,\ s')$

40: **let** $reference := \langle \texttt{REQ}, s'.\overline{w'}.\texttt{nonce} \rangle$

41: **if** $method = \texttt{GET}$ **then**

42: **let** $body := \langle\rangle$

43: **let** $parameters := data$

44: **let** $origin := \bot$

45: **else**

46: **let** $body := data$

47: **let** $parameters := url.\texttt{parameters}$

48: **let** $origin := docorigin$

49: **let** $req := \langle \texttt{HTTPReq}, \nu_4, method, url.\texttt{host}, url.\texttt{path}, parameters, \langle\rangle, body \rangle$

50: **let** $s' :=$ CANCELNAV$(reference, s')$

51: **call** HTTP_SEND$(reference,\ req,\ url,\ origin,\ referrer,\ referrerPolicy,\ a,\ s')$

52: **case** $\langle \texttt{SETSCRIPT}, window, script \rangle$

53: **let** $\overline{w'} :=$ GETWINDOW$(\overline{w}, window, s')$

54: **let** $s'.\overline{w'}.\texttt{activedocument}.\texttt{script} := script$

55: **stop** $\langle\rangle$, s'

56: **case** $\langle \texttt{SETSCRIPTSTATE}, window, scriptstate \rangle$

57: **let** $\overline{w'} :=$ GETWINDOW$(\overline{w}, window, s')$

58: **let** $s'.\overline{w'}.\texttt{activedocument}.\texttt{scriptstate} := scriptstate$

59: **stop** $\langle\rangle$, s'

60: **case** $\langle \texttt{XMLHTTPREQUEST}, url, method, data, xhrreference \rangle$

61: **if** $method \in \{\texttt{CONNECT}, \texttt{TRACE}, \texttt{TRACK}\} \vee xhrreference \notin V_{\text{process}} \cup \{\bot\}$ **then**

62: **stop**

63: **if** $url.\texttt{host} \not\equiv docorigin.\texttt{host} \vee url.\texttt{protocol} \not\equiv docorigin.\texttt{protocol}$ **then**

64: **stop**

65: **if** $method \in \{\texttt{GET}, \texttt{HEAD}\}$ **then**

66: **let** $data := \langle\rangle$

67: **let** $origin := \bot$

68: **else**

69: **let** $origin := docorigin$

70: **let** $req := \langle \texttt{HTTPReq}, \nu_4, method, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, data \rangle$

71: **let** $reference := \langle \texttt{XHR}, s'.\overline{d}.\texttt{nonce}, xhrreference \rangle$

72: **call** HTTP_SEND$(reference,\ req,\ url,\ origin,\ referrer,\ referrerPolicy,\ a,\ s')$

73: **case** $\langle \texttt{BACK}, window \rangle$

74: **let** $\overline{w'} :=$ GETNAVIGABLEWINDOW$(\overline{w},\ window,\ \bot,\ s')$

75: **call** NAVBACK$(\overline{w'},\ s')$

76: **case** $\langle \texttt{FORWARD}, window \rangle$

77: **let** $\overline{w'} :=$ GETNAVIGABLEWINDOW$(\overline{w},\ window,\ \bot,\ s')$

78: **call** NAVFORWARD$(\overline{w'},\ s')$

79: **case** $\langle \texttt{CLOSE}, window \rangle$

80: **let** $\overline{w'} :=$ GETNAVIGABLEWINDOW$(\overline{w},\ window,\ \bot,\ s')$

81: **remove** $s'.\overline{w'}$ from the sequence containing it

82: **stop** $\langle\rangle$, s'

83: **case** $\langle \texttt{POSTMESSAGE}, window, message, origin \rangle$

84: **let** $\overline{w'} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w'}.\texttt{nonce} \equiv window$

85: **if** $\exists \overline{j} \in \mathbb{N}$ **such that** $s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{active} \equiv \top$
$\hookrightarrow \wedge (origin \not\equiv \bot \implies s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{origin} \equiv origin)$ **then**

86: **let** $s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{scriptinputs} := s'.\overline{w'}.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
$\hookrightarrow +^{\langle\rangle} \langle \texttt{POSTMESSAGE}, s'.\overline{w}.\texttt{nonce}, docorigin, message \rangle$

87: **stop** $\langle\rangle$, s'

88: **case** else

89: **stop**

Algorithm 8 Web Browser Model: Process an HTTP response.

1: **function** PROCESSRESPONSE($response$, $reference$, $request$, $requestUrl$, key, a, f, s')
2: **if** Set-Cookie $\in$ $response$.headers **then**
3: **for each** $c \in^{\langle\rangle}$ $response$.headers [Set-Cookie], $c \in$ Cookies **do**
4: **let** s'.cookies [$request$.host] $\hookrightarrow$:= AddCookie(s'.cookies [$request$.host] , c, $requestUrl$.protocol)
5: **if** Strict-Transport-Security $\in$ $response$.headers $\wedge$ $requestUrl$.protocol $\equiv$ S **then**
6: **let** s'.sts := s'.sts $+^{\langle\rangle}$ $request$.host
7: **if** Referer $\in$ $request$.headers **then**
8: **let** $referrer$:= $request$.headers[Referer]
9: **else**
10: **let** $referrer$:= $\bot$
11: **if** Location $\in$ $response$.headers $\wedge$ $response$.status $\in \{303, 307\}$ **then**
12: **let** url := $response$.headers [Location]
13: **if** url.fragment $\equiv \bot$ **then**
14: **let** url.fragment := $requestUrl$.fragment
15: **let** $method'$:= $request$.method
16: **let** $body'$:= $request$.body
17: **if** Origin $\in$ $request$.headers $\hookrightarrow$ $\wedge$ $request$.headers[Origin] $\not\equiv \Diamond$ $\hookrightarrow$ $\wedge$ ($\langle url$.host, url.protocol$\rangle \equiv \langle request$.host, $requestUrl$.protocol$\rangle$ $\hookrightarrow$ $\vee$ $\langle request$.host, $requestUrl$.protocol$\rangle \equiv request$.headers[Origin]) **then**
18: **let** $origin$:= $request$.headers[Origin]
19: **else**
20: **let** $origin$:= $\Diamond$
21: **if** $response$.status $\equiv 303 \wedge$ $request$.method $\not\in$ {GET, HEAD} **then**
22: **let** $method'$:= GET
23: **let** $body'$:= $\langle\rangle$
24: **if** $\exists\, \overline{w} \in$ Subwindows(s') **such that** $s'.\overline{w}$.nonce $\equiv \pi_2(reference)$ **then** → Do not redirect XHRs.
25: **let** req := $\langle$HTTPReq, ν_6, $method'$, url.host, url.path, url.parameters, $\langle\rangle$, $body'\rangle$
26: **let** $referrerPolicy$:= $response$.headers[ReferrerPolicy]
27: **call** HTTP_SEND($reference$, req, url, $origin$, $referrer$, $referrerPolicy$, a, s')
28: **else**
29: **stop** $\langle\rangle$, s'
30: **switch** $\pi_1(reference)$ **do**
31: **case** REQ
32: **let** $\overline{w} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}$.nonce $\equiv \pi_2(reference)$ **if possible;** $\hookrightarrow$ **otherwise stop** → normal response
33: **if** $response$.body $\not\sim \langle *, * \rangle$ **then**
34: **stop** $\langle\rangle$, s'
35: **let** $script$:= $\pi_1(response$.body)
36: **let** $scriptstate$:= $\pi_2(response$.body)
37: **let** d := $\langle \nu_7$, $requestUrl$, $response$.headers, $referrer$, $script$, $scriptstate$, $\langle\rangle$, $\langle\rangle$, $\top\rangle$
38: **if** $s'.\overline{w}$.documents $\equiv \langle\rangle$ **then**
39: **let** $s'.\overline{w}$.documents := $\langle d \rangle$
40: **else**
41: **let** $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}$.documents.$\overline{i}$.active $\equiv \top$
42: **let** $s'.\overline{w}$.documents.$\overline{i}$.active := $\bot$
43: **remove** $s'.\overline{w}$.documents.($\overline{i} + 1$) and all following documents $\hookrightarrow$ from $s'.\overline{w}$.documents
44: **let** $s'.\overline{w}$.documents := $s'.\overline{w}$.documents $+^{\langle\rangle}$ d
45: **stop** $\langle\rangle$, s'
46: **case** XHR
47: **let** $\overline{w} \leftarrow$ Subwindows(s'), $\overline{d}$ **such that** $s'.\overline{d}$.nonce $\equiv \pi_2(reference)$ $\hookrightarrow$ $\wedge$ $s'.\overline{d} = s'.\overline{w}$.activedocument **if possible; otherwise stop** → process XHR response
48: **let** $headers$:= $response$.headers $-$ Set-Cookie
49: **let** $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs $+^{\langle\rangle}$ $\langle$XMLHTTPREQUEST, $headers$, $response$.body, $\pi_3(reference)\rangle$
50: **stop** $\langle\rangle$, s'

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 2) 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 3) 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.
- The function HTTP_SEND (Algorithm 4) 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 32. *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 5) and NAVFORWARD (Algorithm 6), 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 7) 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 8) 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 32. *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 39. The pair $((\langle a, f, m\rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 9 (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).

2.8 Definition of Web Browsers

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

Definition 40 (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 $s_0^p \in Z_{\text{webbrowser}}$. ◇

Algorithm 9 Web Browser Model: Main Algorithm.

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

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

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

Algorithm 10 Function to retrieve an unhandled input message for a script.

1: **function** $\mathsf{CHOOSEINPUT}(s', scriptinputs)$
2: **let** iid **such that** $iid \in \{1, \cdots, |scriptinputs|\} \wedge iid \not\in^{\langle\rangle} s'.\texttt{handledInputs}$ **if possible;**
  $\hookrightarrow$ **otherwise return** $(\bot, s')$
3: **let** $input := \pi_{iid}(scriptinputs)$
4: **let** $s'.\texttt{handledInputs} := s'.\texttt{handledInputs} +^{\langle\rangle} iid$
5: **return** $(input, s')$

2.9 Helper Functions

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

CHOOSEINPUT (Algorithm 10) 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 $\mathsf{CHOOSEINPUT}(s', scriptinputs)$ 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 11) 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 $\mathsf{PARENTWINDOW}(tree, docnonce)$ is used. It takes the term *tree*, which is the (partly

Algorithm 11 Function to extract the first script input message matching a specific pattern.

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

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$(\mathit{tree}, \mathit{docnonce})$ 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$(\mathit{tree}, \mathit{docnonce})$ returns $\langle\rangle$. Otherwise, let $\langle \mathit{docnonce}, \mathit{location}, \langle\rangle, \mathit{referrer}, \mathit{script}, \mathit{scriptstate}, \mathit{scriptinputs}, \mathit{subwindows}, \mathit{active}\rangle$ denote the subterm of *tree* corresponding to the document referred to by *docnonce*. Then, SUBWINDOWS$(\mathit{tree}, \mathit{docnonce})$ 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$(\mathit{tree}, \mathit{docnonce})$ 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.

Algorithm 12 Relation of a DNS server R^d.

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

2.10 DNS Servers

Definition 41. 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

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

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 12.

2.11 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).

Definition 42. 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. ◇

2.12 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 43 (Base state for an HTTPS server). The state of each HTTPS server that is an instantiation of this relation must contain at least the following subterms: $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \mathsf{IPs}$ (containing the IP address of a DNS server), $keyMapping \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$ (containing a mapping from domains to

public keys), *tlskeys* $\in$ [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 13–17, and the main relation in Algorithm 18.

Algorithm 13 Generic HTTPS Server Model: Sending a DNS message (in preparation for sending an HTTPS message).

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

Algorithm 14 Generic HTTPS Server Model: Default HTTPS response handler.

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

Algorithm 15 Generic HTTPS Server Model: Default trigger event handler.

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

Algorithm 16 Generic HTTPS Server Model: Default HTTPS request handler.

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

Algorithm 17 Generic HTTPS Server Model: Default handler for other messages.

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

Algorithm 18 Generic HTTPS Server Model: Main relation of a generic HTTPS server

Input: $\langle a, f, m\rangle, s$
1: **let** $s' := s$
2: **if** $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
3: **let** $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
4: **let** $m' \leftarrow d_V(s')$
5: **let** $a' \leftarrow \mathsf{IPs}$
6: **stop** $\langle\langle a', a, m'\rangle\rangle$, s'
7: **if** $\exists\, m_{\text{dec}}$, k, k', $inDomain$ **such that** $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s.\texttt{tlskeys}$ **then**
8: **let** n, $method$, $path$, $parameters$, $headers$, $body$ **such that**
↪ $\langle \texttt{HTTPReq}, n, method, inDomain, path, parameters, headers, body\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop**
9: **call** PROCESS_HTTPS_REQUEST(m_{dec}, k, a, f, s')
10: **else if** $m \in \mathsf{DNSResponses}$ **then** → Successful DNS response
11: **if** $m.\texttt{nonce} \not\in s.\texttt{pendingDNS} \vee m.\texttt{result} \not\in \mathsf{IPs}$
↪ $\vee\, m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].2.\texttt{host}$ **then**
12: **stop**
13: **let** $reference := s.\texttt{pendingDNS}[m.\texttt{nonce}].1$
14: **let** $request := s.\texttt{pendingDNS}[m.\texttt{nonce}].2$
15: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle reference, request, \nu_{n1}, m.\texttt{result}\rangle$
16: **let** $message := \mathsf{enc_a}(\langle request, \nu_{n1}\rangle, s'.\texttt{keyMapping}[request.\texttt{host}])$
17: **let** $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
18: **stop** $\langle\langle m.\texttt{result}, a, message\rangle\rangle$, s'
19: **else if** $\exists\,\langle reference, request, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
↪ **such that** $\pi_1(\mathsf{dec_s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
20: **let** $m' := \mathsf{dec_s}(m, key)$
21: **if** $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
22: **stop**
23: **remove** $\langle reference, request, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$
24: **call** PROCESS_HTTPS_RESPONSE(m', $reference$, $request$, a, f, s')
25: **else if** $m \equiv \texttt{TRIGGER}$ **then** → Process was triggered
26: **call** PROCESS_TRIGGER(a, s')
27: **else**
28: **call** PROCESS_OTHER(m, a, f, s')
29: **stop**

2.13 General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [10].

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

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

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 45. 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 46. 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 47. We say that an DY process p *created* a message m (at some point) in a run if m is derivably contained in a message emitted by p in some processing step and if there is no earlier processing step where m is derivably contained in a message emitted by some DY process p'. ◇

Definition 48. 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 8). ◇

Definition 49. 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 50. 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 51. We say that a *script initiated a request r* if a browser triggered the script (in Line 10 of Algorithm 7) 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 52. 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 18). ◇

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

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

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

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

Proof. **(1)** follows immediately from the pre-conditions.

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 $s_j \rightarrow s_{j+1}$ without u prior leaking the key k to anyone except for u and b and that the browser is not fully corrupted in s_j, and lead this to a contradiction.

The browser is honest in s_i. From the definition of the browser b, we see that the key k is always chosen as a fresh nonce (placeholder ν_3 in Lines 64ff. of Algorithm 9) that is not used anywhere else. Further, the key is stored in the browser's state in *pendingRequests*. The information from *pendingRequests* is not extracted or used anywhere else (in particular it is not accessible by scripts). If the browser becomes closecorrupted prior to s_j (and after s_i), the key cannot be used anymore (compare Lines 46ff. of Algorithm 9). 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 9), a host header is always contained in HTTP requests by browsers. From Line 70 of Algorithm 9 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. In the situation of Lemma 1, as long as u does not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ and the browser does not become fully corrupted, k is not known to any DY process $p \notin \{u, b\}$ (i.e., $\nexists\, s' = (S', E') \in \rho$: $k \in d_{N^p}(S'(p)))$.

Lemma 2. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b)$.`windows` with the origin $\langle dom, \mathtt{S} \rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b)$.`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 8) 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 8. The values (domain and protocol) used there stem from the information about the request (say, *req*) that led to the loading of d. These values have been stored in *pendingRequests* between the request and the response actions. The contents of *pendingRequests* are indexed by freshly chosen nonces and

can never be altered or overwritten (only deleted when the response to a request arrives). The information about the request *req* was added to *pendingRequests* in Line 69 (or Line 72 which we can exclude as we will see later) of Algorithm 9. 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 $\mathtt{S}$. The domain part of the origin (in our case *dom*) is taken directly from the *pendingRequests* and is thus guaranteed to be unaltered.

From Line 70 of Algorithm 9 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 $S_i(b).\mathtt{keyMapping}[dom] = S_0(b).\mathtt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ 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 1 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 3. If in a processing step $s_i \rightarrow s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, \mathtt{S}\rangle$ 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 8) 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 4), 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 7
- Line 72 of Algorithm 7
- Line 27 of Algorithm 8

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 2 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 8 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 1, but is applied to the generic HTTPS server (instead of the web browser).

Lemma 4. If in the processing step $s_i \rightarrow s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest instance s of the generic HTTPS server model

(I) emits an HTTPS request of the form

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

(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 defined on top of the HTTPS server does not read or write the *pendingRequests* subterm of its state,

(V) the instance model defined on top of the HTTPS server does not emit messages in $\mathsf{HTTPSRequests}$,

(VI) the instance model defined on top of the HTTPS server 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 $s_j \rightarrow s_{j+1}$ where some process leaks k to $\mathcal{W} \setminus \{u, s\}$, there is a processing step $s_h \rightarrow s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ or the process s is corrupted in s_j.

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

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

Proof. **(1)** follows immediately from the pre-conditions. The proof is the same as for Lemma 1:

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 $s_j \rightarrow s_{j+1}$ without u prior leaking the key k to anyone except for u and s and that the process s is not corrupted in s_j, and lead this to a contradiction.

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

- The message emitted in Line 3 of Algorithm 13 has a different message structure
- As s is honest, it does not send the message of Line 6 of Algorithm 18
- There is no other place in the generic HTTPS server model where messages are emitted and due to pre-condition (V), the application-specific model does not emit HTTPS requests.

The value k, which is the placeholder ν_{n1} in Algorithm 18, is only stored in the *pendingRequests* subterm of the state of s, i.e., in $S^{i+1}(s).\mathtt{pendingRequests}$. Other than that, s only accesses this value in Line 19 of Algorithm 18, 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 pre-condition (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 18 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 pre-condition (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 18, 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 13, where a DNS request is sent, and in Line 18 of Algorithm 18, 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 18 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'.

□

3 Modeling Decisions

In this section, design choices regarding the modeling of the WIM are discussed.

3.1 Web attackers can't perform IP spoofing

Since the exchange of messages in the WIM always assumes a TCP connection (as HTTP is used), a TCP handshake must be performed before a message can be sent successfully. Since web attackers are not able to complete such a handshake for an honest process, we only allow network attackers to perform IP spoofing, i.e., only network attackers can send events with arbitrary sender addresses.

If the WIM is used in a scenario where other transport protocols are used (e.g., UDP), web attackers can also be allowed to perform IP spoofing.

3.2 Documents can't contain multiple scripts

As described in section 1.6.2, a document of a window can contain only one script. This script subsumes both the behavior of a real-world HTML document containing JavaScript and user interactions with the document. Thus, the WIM also provides only an abstract view on JavaScript.

The WIM can be extended in this respect so that documents can contain multiple scripts, thus eliminating the need for subsumption by a single script. This can also involve a more concrete modeling of JavaScript.

3.3 Domain boundaries of web pages and cookies can't be extended

The WIM does not allow the extension of the domain boundaries of documents and cookies via the *Document.domain* property (which is now deprecated) or the *domain* attribute of cookies. This is because these options undermine the security properties of the same origin policy and would unnecessarily complicate our browser model. However, if needed, the WIM can be extended to include these features.

References

[1] M. Abadi and C. Fournet. "Mobile Values, New Names, and Secure Communication". In: *POPL*. ACM Press, 2001, pp. 104–115.

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

[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] Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, N. Wenzler, and T. Würtele. *A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification*. Cryptology ePrint Archive, Report 2021/1012. `https://ia.cr/2021/1012`. 2021.

[5] 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.

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

[7] 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)*. IEEE, May 2019. DOI: `10.1109/sp.2019.00067`.

[8] 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`.

[9] 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*. IEEE Computer Society, 2014, pp. 673–688.

[10] 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. Springer, 2015, pp. 43–65.

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

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

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

[14] P. E. Hoffman and P. McManus. *DNS Queries over HTTPS (DoH)*. RFC 8484. Oct. 2018. DOI: `10.17487/RFC8484`. URL: `https://www.rfc-editor.org/info/rfc8484`.

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

[16] S. Rose, M. Larson, D. Massey, R. Austein, and R. Arends. *DNS Security Introduction and Requirements*. RFC 4033. Mar. 2005. DOI: `10.17487/RFC4033`. URL: `https://www.rfc-editor.org/info/rfc4033`.

[17] 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`.