Automatic and Robust Client-Side Protection for Cookie-Based Sessions

Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan

Università Ca' Foscari Venezia
{michele,calzavara,focardi,khan}@dais.unive.it

Abstract. Session cookies constitute one of the main attack targets against client authentication on the Web. To counter that, modern web browsers implement native cookie protection mechanisms based on the `Secure` and `HttpOnly` 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, with a mechanized proof of noninterference assessing the robustness of the `Secure` and `HttpOnly` cookie flags against both web and network attacks. 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 such 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.

1 Introduction

Providing access to online content-rich resources such as those available in modern web applications requires tracking a user's identity through 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.

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, for the browser to attach it automatically to all HTTP(S) requests to the server which registered it. If the cookie contains an adequately long string of random data, the server can effectively identify the client and restore its state, thus implementing a web session across different requests.

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 [17, 16, 9, 15]. Interestingly, this problem is so serious that modern web browsers implement native protection mechanisms based on the `Secure` and `HttpOnly` flags to shield session

cookies from unintended access by scripts injected within 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.

Contributions. With the present paper we provide the first such result assessing the robustness of the `Secure` and `HttpOnly` cookie flag mechanisms with respect to a precise and rigorous attacker model, which captures both web threats (based on, e.g., code injection) and network attacks. We state our result in terms of *reactive noninterference* [7], a popular and widely accepted definition of information security, which provides strong, full-rounded protection against any (direct or indirect) information flow occurring in the browser. To carry out our mechanized proof, we extend Featherweight Firefox [6, 5], a core model of a web browser developed in the Coq proof assistant [1], and we rely on Coq's facilities for interactive theorem proving to establish our result.

The security guarantees provided by our mechanized proof apply only to sessions that draw on appropriately flagged cookies. Clearly, however, poorly engineered websites that do not comply with the required 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 `Secure` and `HttpOnly` 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.

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. 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 effects on the user's browsing experience.

Structure of the paper. Section 2 provides background material. Section 3 describes our formal model and the main theoretical results. Section 4 focuses on the practical aspects of session cookie security and presents CookiExt. Section 5 compares the present paper to related work. Section 6 concludes[1].

2 Background

2.1 Session cookies: attacks and defenses

Web attacks. Web browsers store cookies in their local storage and implement a simple protection mechanism based on the so-called "same-origin policy", whereby cookies registered by a given domain are made accessible only

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

to scripts retrieved from that same domain. Unfortunately, as it is well-known, the same-origin policy may be circumvented by a widespread form of code injection attacks known as *cross-site scripting* (XSS). 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 [10]. As a result, the injected script is granted access to the DOM of the page, in particular to the Javascript object `document.cookie` containing the session cookies, which it can leak to the attacker's website.

Network attacks. A network attacker may be able to fully inspect all the unencrypted traffic exchanged between the browser and the server. Though adopting HTTPS connections to encrypt network traffic would provide an effective countermeasure against eavesdropping, protecting session cookies against improper disclosure is tricky. On the one hand, many websites are still only *partially* deployed over HTTPS, and require special attention. In fact, cookies registered by a given domain are by default attached to *all* the requests to that domain: consequently, unless appropriate protection is put in place, loading a page over HTTPS may still leak session cookies, whenever the page retrieves additional contents (e.g., images or scripts) over an HTTP connection to the same domain. On the other hand, even websites which are *completely* deployed over HTTPS are vulnerable to session cookie theft, whenever an attacker is able to inject HTTP links to them in unrelated web pages [13].

Protection mechanisms. Web development frameworks provide two main mechanisms to secure session cookies, based on the `Secure` and `HttpOnly` flags. The `HttpOnly` flag blocks any access to a cookie attempted by JavaScript or any other non-HTTP API, thus making cookies available only upon transmissions of HTTP(S) requests and thwarting 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.

2.2 Formal browser models

Web browsers can be formalized in terms of constrained labelled transition systems known as *reactive* systems [6]. 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 [7]). *We define a* reactive system *as a tuple* $(\mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, \longrightarrow)$*, 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. 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. $C \in \mathcal{C}$ *and* $C \xrightarrow{\alpha} Q$ *imply* $\alpha \in \mathcal{I}$ *and* $Q \in \mathcal{P}$;
2. $P \in \mathcal{P}$, $Q \in \mathcal{S}$ *and* $P \xrightarrow{\alpha} Q$ *imply* $\alpha \in \mathcal{O}$;
3. $C \in \mathcal{C}$ *and* $i \in \mathcal{I}$ *imply* $\exists P \in \mathcal{P} : C \xrightarrow{i} P$;
4. $P \in \mathcal{P}$ *implies* $\exists o \in \mathcal{O}, Q \in \mathcal{S} : P \xrightarrow{o} Q$.

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

Definition 2 (Trace). *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:*

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

A reactive system generates the trace (I, O) *if we have $Q_0(I) \Rightarrow O$, where Q_0 is the initial state of the reactive system.*

2.3 Reactive noninterference

Given the previous definition, we can introduce an effective notion of information security based on the theory of *noninterference*. We presuppose a pre-order of security labels $(\mathcal{L}, \sqsubseteq)$ and characterize the power of an observer in terms of the labels $l \in \mathcal{L}$, where higher labels correspond to higher power.

Definition 3 (Reactive Noninterference [7]). *A reactive system is noninterferent if 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 [7], different definitions of stream similarity correspond to different sensible notions of information security. We focus on a very natural definition, which gives rise to a practically useful (termination-insensitive) notion of noninterference called *indistinguishable security*.

Definition 4 (Stream Similarity). *We let $\approx_l$ be the largest relation closed under the following inference rules:*

$$\frac{}{[] \approx_l []}\ (\text{S-Nil}) \qquad \frac{\mathit{visible}_l(s) \qquad \mathit{visible}_l(s') \qquad s \approx_l s' \qquad S \approx_l S'}{s :: S \approx_l s' :: S'}\ (\text{S-Vis})$$

$$\frac{\neg \mathit{visible}_l(s) \qquad S \approx_l S'}{s :: S \approx_l S'}\ (\text{S-InvisL}) \qquad \frac{\neg \mathit{visible}_l(s') \qquad S \approx_l S'}{S \approx_l s' :: S'}\ (\text{S-InvisR})$$

The definition is parametric with respect to a visibility and a similarity relations for individual stream elements. Different instantiations of these relations entail different security guarantees, as we discuss in Section 3.3.

3 Formalizing Session Security

We continue with 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 (though rigorous) whenever possible: full details can be found in the online Coq scripts at `https://github.com/wilstef/secookie`.

3.1 Extending Featherweight Firefox

Featherweight Firefox (FF) is a core model of a web browser, developed in the Coq proof assistant [6, 5]. Despite its name, the model is not tailored specifically around Firefox. Instead, it provides a fairly rich subset of the main functionalities of any 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.

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.

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 store to capture the `Secure` and `HttpOnly` flags with their intended semantics, and (iii) HTTP(S) redirects, a feature included in related models [2, 3] which has been shown to have a significant impact on browser security.

The implementation of the extended model arises as expected, though it requires several changes to the existing framework to get a working Coq program. 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).

3.2 Threat model

As anticipated, 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:

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

We can easily prove that $(\mathcal{L}, \sqsubseteq)$ is a pre-order, hence our definition is well-suited for the theory of reactive noninterference.

It is very natural to characterize the attacker power in terms of a security label in the previous pre-order. Specifically, level $\mathsf{http}(d)$ corresponds to a *web attacker* at d, which has no network capability and can only observe data sent to d itself. A *network attacker*, instead, resides at level net and is stronger than any web attacker, being able to inspect the contents of all the unencrypted network traffic. 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. Finally, level $\mathsf{https}(d)$ corresponds to an even more powerful attacker, which has fully compromised the web server at d: this attacker has all the capabilities of a network attacker and can also decrypt all the encrypted traffic sent to d.

We anticipate that, by quantifying over all the possible inputs, our model will implicitly provide the attacker (at any level) with the ability to inject malicious contents on all websites, thus naturally capturing XSS attacks.

3.3 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 $\mathsf{https}(d)$ or higher.

In both cases we target strong confidentiality guarantees, to ensure that the secrecy of a session cookie is protected against both explicit and implicit flows of information. Interestingly, we can uniformly characterize properties (1) and (2) in terms of reactive noninterference and carry out a single security proof which entails both. 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 streams' components, will produce indistinguishable output streams for any observer/attacker that is not the intended owner of these cookies.

Overview. Before delving into the technical details, we first provide an intuition about how existing attacks are captured by reactive noninterference. Consider a web attacker running the website `attacker.com` and take the following script snippet:

```
val = get_ck_val(document.cookie,"PHPSESSID");
for (x in val) {
   <contact http://attacker.com/leak?pos=x&char=val[x]>
}
```

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 character of the cookie value to the attacker's website. If the script is injected into a response from `honest.com`, for instance through XSS, 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 (an *implicit* flow of information) are not disclosed to the attacker, and it is thus safe to deem two HTTP(S) responses from `honest.com` as *similar* 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, since the attacker would be able to draw a difference between them based on the outputs observable at `attacker.com`, thus violating reactive noninterference.

The reasoning can be generalized to the `Secure` flag and network attackers, with the proviso that any `Secure` cookie must be marked also as `HttpOnly` to be actually protected: the script above already highlights this point. Indeed, if the cookie `PHPSESSID` is only marked as `Secure`, the previous script could still leak over HTTP all the characters composing the cookie value, thus allowing a network attacker to draw a difference between two responses identical up to the cookies they set.

Formalization. We start by introducing some notation. We 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. We let *uwi* range over window identifiers, i.e., natural numbers serving as an internal representation of browser windows; similarly, we 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, mostly self-explanatory productions:

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

We define *visibility* for output events by means of the following inference rules:

$$\frac{url\_label(ncid) \sqcap \mathsf{net} \sqsubseteq l \qquad * \in \{\mathsf{doc}, \mathsf{script}, \mathsf{xhr}\}}{visible_l(\mathsf{net}\_ * \_\mathsf{req}(ncid, req))} \text{(VO-Net)} \qquad \frac{}{visible_\top(o)} \text{(VO-Top)}$$

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 both the encrypted and the unencrypted network traffic is visible to any attacker l such that $l \sqsupseteq \mathsf{net}$.

We then define *similarity* for output events through the following rules:

$$\frac{\exists d : url\_label(ncid) = \mathsf{https}(d) \not\sqsubseteq l \qquad * \in \{\mathsf{doc}, \mathsf{script}, \mathsf{xhr}\}}{\mathsf{net}\_ * \_\mathsf{req}(ncid, req) \approx_l \mathsf{net}\_ * \_\mathsf{req}(ncid, req')} \text{(SO-Crypt)} \qquad \frac{}{o \approx_l o} \text{(SO-Refl)}$$

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 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 for input events is similar, but subtler. Again, we start by introducing some notation. We 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 productions:

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

We presuppose a further condition to rule out input events built around `Secure` cookies registered over HTTP. This corresponds to assuming the following condition for the last three clauses of the input-event productions above: whenever

[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.

there exists a cookie $c \in \mathit{resp.set\_cookies}$ such that $c.\mathit{secure} = \mathsf{true}$, then $\mathit{ncid.url}$ must be of the form $\mathsf{url}(\mathsf{https}, d, p)$ for some d and p. Any input that does not satisfy this condition is clearly ill-formed, as `Secure` cookies received in the clear cannot be protected at the client side. 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, while current browsers do not seem to implement this check, that is enforced by our browser extension (cf. Section 4).

We define *similarity* for input events as follows:

$$\text{(SI-Net)}\quad \frac{\mathit{in\_erase}_l(\mathit{resp}) = \mathit{in\_erase}_l(\mathit{resp}') \qquad * \in \{\mathsf{doc}, \mathsf{script}, \mathsf{xhr}\}}{\mathsf{net\_} * \mathsf{\_resp}(\mathit{ncid}, \mathit{uwi}, \mathit{resp}) \approx_l \mathsf{net\_} * \mathsf{\_resp}(\mathit{ncid}, \mathit{uwi}, \mathit{resp}')} \qquad \text{(SI-Refl)}\quad \frac{}{i \approx_l i}$$

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

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

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 previous informal overview).

We conclude by defining *visibility* for input events: that is an easy task, since we just stipulate that $\mathit{visible}_l(i)$ holds true for all security labels l and all input events i. In other words, 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.

Formal results. Let EFF denote the extended FF of Section 3.1: 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 A for an intuition about the coinductive technique adopted in the proof. The result is interesting and important in itself, however, as it provides a certified guarantee of the effectiveness of the `Secure` and `HttpOnly` flags as robust protection mechanisms for session cookies. Needless to say, the theorem does not say anything about the security of sessions in existing web applications, as that depends critically on the correct use of the 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 modern browsers.

4 Strengthening Session Security

4.1 Session cookie protection in existing systems

We start with an analysis of the actual adoption of the security flags in existing systems. To accomplish that, we conduct an analysis of the the top 1000 websites of Alexa: we first collect the cookies registered through the HTTP headers by these websites, and then apply a heuristic to isolate session cookies. The heuristic marks a cookie as a session cookie if it satisfies either of the following conditions:

1. the cookie name contains the strings 'sess' or 'sid';
2. the cookie value contains at least 10 characters and its index of coincidence[3] is below 0.04.

Our solution is consistent with previous proposals [17] and has been validated by a manual investigation on known websites. In particular, condition 1 is motivated by the observation that several web frameworks offer native support for cookie-based sessions and by default register session cookies with known names satisfying this condition. In addition, it appears that custom session identifiers tend to include the string 'sess' or 'sid' 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. Clearly, there is no *a prori* guarantee of accuracy for our heuristic. As we will discuss, however, we have strong evidence that our survey is reliable enough (cf. Section 4.5).

4.2 The need for a client-side defense

Table 1 provides some statistics which highlight that the large majority of the session cookies we identified (71.35%) has no flag set: though this percentage may be partially biased by the adoption of a heuristic, it provides clear indications of a limited practical deployment of the available protection mechanisms. Further evidence will be provided by our field experiments.

`HttpOnly`	`Secure`	#cookies	percentage
yes	yes	32	2.81%
yes	no	284	24.96%
no	yes	10	0.88%
no	no	812	71.35%

Table 1. Statistics about cookie flags

Of the two flags, `HttpOnly` appears to be adopted much more widely than `Secure`. We conjecture two reasons for that: 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 standard API; second, `Secure` cookies presuppose an HTTPS implementation, which is not available for all websites. We further investigate this point below.

[3] This is a statistical measure which can be effectively employed to understand how likely a given text was randomly generated [11].

Evaluating client-side protection. 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 [17, 21]. We propose to push this idea further, by automatically flagging session cookies also as `Secure` and enforcing a redirection to HTTPS for supporting websites.

To get a better understanding about the practical implications of this approach, 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 (43.34%) 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 (8.33%) set the `Secure` flag for at least one session cookie. Remarkably, it turns out that 141 out of these 192 websites (73.44%) 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 risk of being disclosed to a network attacker when navigating these websites.

4.3 Client-side protection with CookiExt

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

Overview. At a high level, the behaviour of CookiExt can be summarized as follows: when the browser receives an HTTP(S) response, CookiExt inspects its headers, trying to identify the session cookies based on the heuristic discussed earlier. If a session cookie is found, CookiExt behaves as follows:

- if the response was sent over HTTPS, all the identified session cookies are marked `Secure` and `HttpOnly`;
- if the response was sent over HTTP, all the identified session cookies are erased from the HTTP headers.

In both cases, 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.

Supporting "mixed" websites. Mixed websites are websites which support HTTPS but make some of their contents available only on HTTP. 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, assuming to be able to detect the absence of HTTPS support for some links, even the adoption of a fallback to HTTP would eventually break the user's session: in fact, since session cookies are by default marked `Secure` by our extension, they will not be sent to the HTTP portion of the website.

We therefore adopt the following, more elaborate solution. When CookiExt forces a redirection over HTTPS, we implement a check to detect possible failures (see below). If the redirection cannot be performed, we enforce a fallback to HTTP, distinguishing two cases: if the failure arises from the request of a page, we extend the set of the cookies attached to the request with all the cookies which have been made `Secure` by CookiExt, but were not originally marked `Secure` by the server; if the browser instead is trying to retrieve a sub-resource, like an image or a script, we leave the set of cookies attached to the request unchanged. This way we confine any deviation from the browser behaviour expected by the server only to sub-resources, which typically do not require authenticated access. Clearly, transmitting in clear the session cookies identified by the heuristic exposes the client to a risk. However, this approach offers an interesting compromise between usability and security: in fact, the user's navigation will not break the session, since page requests will always include session cookies, but at the same time the attack surface for network attackers will be significantly reduced, since retrieving an image over HTTP inside an HTTPS page will not leak any session cookie.

Checking HTTPS support. As we said, CookiExt could try to enforce a redirection from HTTP to HTTPS also for websites supporting only HTTP access. The Chrome API already allows one to detect a number of network connection errors which may arise when HTTPS is not supported; however, some of these alerts are only triggered after a significant delay, which may negatively affect usability.

Our choice is to set a relatively small timeout every time CookiExt forces an HTTPS redirection: if no response is received before the timeout expires, the extension fallbacks to HTTP. To prevent a network attacker from tapping with outgoing HTTPS connections and disabling our client-side defense, CookiExt keeps track of all the pages for which a successful redirection from HTTP to HTTPS has been performed in the past, and notifies the user in case of an unexpected lack of HTTPS support possibly due to malicious network activities.

4.4 Noninterference in theory and in practice

Careful readers will argue that our non-interference 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 the desired flagging to all inputs, *de facto* enforcing the similarity condition on the input streams that constitutes the hypothesis of the non-interference definition. The only discrepancy determined by CookiExt arises from the fallback mechanism, which may end up sending over HTTP cookies that the extension promoted to `Secure`. While this effect has no counterpart in standard browsers, the gap is again harmless, as all such cookies can be assimilated to `HttpOnly` cookies, for which confidentiality is guaranteed against web attackers.

4.5 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 (though they do not cause any security flaw). We now report on our experiments to evaluate both these aspects.

Security evaluation. To understand the practical impact of false negatives, we analyze again our survey of websites and isolate the cookies flagged `Secure` or `HttpOnly` which are not identified as session cookies by the heuristic: the intuition here is that cookies which are explicitly protected by web developers are likely to contain session information and we deem them as potential false negatives. As it turns out, only 37 of the 1153 cookies ignored by our heuristic (3.21%) have at least one security flag set: in addition, 8 of these 37 cookies are already flagged `Secure` and `HttpOnly`, hence missing them is completely harmless. We then carried out a manual review of the remaining potential false negatives, which showed that none of the 29 cookies left is a real session identifier. We performed this check by authenticating onto the private areas of the websites registering one of these cookies, and then erasing all the cookies identified by our heuristic: a logout from the website implies that all the real session cookies have been successfully recognized.

Usability tests. The only way to understand the practical impact of the false positives is by testing and hands-on experience with the extension. We performed an empirical evaluation 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. The feedback by the users was very important to refine the original implementation and make it work in practice: with our latest prototype, no major complaint was reported at the time of writing, even though some users have been noticing a slight performance degradation when activating CookiExt.

Manual investigation. We carried out a manual investigation on the top 20 websites from the Alexa ranking where we own a personal account. For all the websites we performed three different experiments:

1. Detecting session cookies. We authenticate to the private area of the website and we delete from the browser all the cookies which have been marked as session cookies by our extension: a logout implies that all the real session cookies have been identified by our heuristic. In all cases, our heuristic over-approximated correctly the real set of session cookies;

2. Preserving usability. We navigate the website as deep as possible, trying to identify visible usability issues. Our most serious concern was about the web session being broken by the security policy applied by CookiExt, but this never happened in practice. From our experience, usability crucially hinges on our choice of discriminating the behaviour of CookiExt based on the request type. Occasionally, we noticed that some images are not loaded when our extension is activated: it seems this typically happens with third-party advertisement, since the choice of the contents to deliver to the browser likely depend on some tracking cookies which are stripped off by CookiExt;

3. Evaluating protection. We navigate the website and we log any redirection attempt from HTTP to HTTPS when navigating to an internal web page: we identified 36 requests overall, among which 23 were successfully redirected to HTTPS; we manually verified that 11 of the 13 remaining links do not support HTTPS.

We remark that, though promising, our extension is still a prototype under active development: we are currently performing a larger scale evaluation and implementing a number of practical improvements.

5 Related Work

Browser-side protection mechanisms. The idea of enforcing security browser-side is certainly not new. Below, we focus on a detailed comparison with the works which share direct similarities with our present proposal. Other approaches exist as well [15, 14, 18–20], but the relationships with ours are loose.

SessionShield [17] 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 [21], 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.

Another particularly relevant client-side defense is HTTPS Everywhere [22]. This is a browser extension which enforces communication with many major websites to happen 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 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 white-list. Similar design choices and limitations apply to ForceHTTPS [13], a proposal aimed at protecting high-security websites from network attacks.

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.* [2]. 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.

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.

Reactive noninterference. The theory of reactive noninterference has been first developed by Bohannon *et al.* [7]. Aaron Bohannon's doctoral dissertation [5] provides a mechanized proof of noninterference for a Coq implementation of the original Featherweight Firefox model [6] extended with a number of dynamic checks aimed at preventing information leakage. In the present work we leverage the existing proof architecture to carry out our formal development, with the notable differences and extensions discussed in Section 3 and Appendix A.

Independently from Bohannon's work, Bielova *et al.* [4] 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.* [12] built on this approach to develop FlowFox, a full-fledged web browser implementing fine-grained information flow control.

6 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. We find preliminary experiences with our tool to be fairly satisfactory.

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. Moreover, we plan to combine the current heuristic for session cookie detection with a learning algorithm, to improve its accuracy by analysing the navigation behaviour.

Finally, we remark that both our theory and implementation just focus on the *confidentiality* of session cookies, which is a necessary precondition for thwarting the risk 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 integrity. In a recently submitted paper we consider a much stronger definition of web session security and we discuss its browser-side enforcement [8].

References

1. The Coq proof assistant. `http://coq.inria.fr/`.
2. D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 290–304, 2010.
3. C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 247–262, 2012.
4. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *IEEE International Conference on Network and System Security (NSS)*, pages 97–104, 2011.
5. A. Bohannon. *Foundations of webscript security.* PhD thesis, University of Pennsylvania, 2012.
6. A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *USENIX Conference on Web Application Development (WebApps)*, pages 1–12, Berkeley, CA, USA, 2010. USENIX Association.
7. A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security (CCS)*, pages 79–90, 2009.
8. M. Bugliesi, S. Calzavara, R. Focardi, M. Tempesta, and W. Khan. Formalizing and enforcing web session integrity. Submitted.
9. I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Transactions on Internet Technology*, 12(1):1, 2012.
10. S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense.* Syngress Publishing, 2007.
11. W. F. Friedman. *The index of coincidence and its applications to cryptanalysis.* Cryptographic Series, 1922.
12. W. D. Groef, D. Devriese, N. Nikiforakis, and F. 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.

13. C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *International Conference on World Wide Web (WWW)*, pages 525–534, 2008.
14. M. Johns and J. Winter. RequestRodeo: client side protection against session riding. *Proceedings of the OWASP Europe Conference*, pages 5–17, 2006.
15. E. Kirda, C. Krügel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *ACM Symposium on Applied Computing (SAC)*, pages 330–337, 2006.
16. A. X. Liu, J. M. Kovacs, and M. G. Gouda. A secure cookie scheme. *Computer Networks*, 56(6):1723–1730, 2012.
17. N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight protection against session hijacking. In *Engineering Secure Software and Systems (ESSoS)*, pages 87–100, 2011.
18. N. Nikiforakis, Y. Younan, and W. Joosen. HProxy: Client-side detection of SSL stripping attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 200–218, 2010.
19. P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *European Symposium on Research in Computer Security (ESORICS)*, pages 100–116, 2011.
20. P. D. Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Serene: Self-reliant client-side protection against session fixation. In *Distributed Applications and Interoperable Systems (DAIS)*, pages 59–72, 2012.
21. S. Tang, N. Dautenhahn, and S. T. King. Fortifying web-based applications automatically. In *ACM Conference on Computer and Communications Security (CCS)*, pages 615–626, 2011.
22. Tor Project and the Electronic Frontier Foundation. HTTPS Everywhere. Available for download at `https://www.eff.org/https-everywhere`.

A Noninterference Proof

Following previous work [7, 4], 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 6 (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 previous proposals, the main difference is in clause 5, 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 1 (Unwinding). *If $Q \simeq_l Q$ for all l, then Q is noninterferent.*

Proof. 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 [5]. 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.

We can finally give a solid intuition about our main result. The browser state b in Featherweight Firefox is represented by a tuple, which contains several data structures representing 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 kind of input 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.

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

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

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 1, 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 [5]. On the other hand, it is a lockstep unwinding relation, which is enough for our present needs.