Università Ca' Foscari di Venezia

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: number

Web Session Security: Formal Verification, Client-Side Enforcement and Experimental Analysis

Wilayat Khan

Supervisor

Prof Michele Bugliesi

PhD Coordinator

Prof Riccardo Focardi

November 27, 2014

Author's Web Page: http://www.unive.it/nqcontent.cfm?a_id=86656&pid=7300426

Author's e-mail: khan@dsi.unive.it

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it

To my mother.

Abstract

Web applications are the dominant means to provide access to millions of on-line services and applications such as banking and e-commerce. To personalize users' web experience, servers need to authenticate the users and then maintain their authentication state throughout a set of related HTTP requests and responses called a *web session*. As HTTP is a stateless protocol, the common approach, used by most of the web applications to maintain web session, is to use HTTP cookies. Each request belonging to a web session is authenticated by having the web browser to provide to the server a unique long random string, known as *session identifier* stored as cookie called *session cookie*. Taking over the session identifier gives full control over to the attacker and hence is an attractive target of the attacker to attack on the confidentiality and integrity of web sessions. The browser should take care of the *web session security*: a session cookie belonging to one source should not be corrupted or stolen or forced, to be sent with the requests, by any other source.

This dissertation demonstrates that security policies can in fact be written down for both, confidentiality and integrity, of web sessions and enforced at the client side without getting any support from the servers and without breaking too many web applications. Moreover, the enforcement mechanisms designed can be proved correct within mathematical models of the web browsers. These claims are supported in this dissertation by 1) defining both, end-to-end and access control,

security policies to protect web sessions; 2) introducing a new and using exiting mathematical models of the web browser extended with confidentiality and integrity security policies for web sessions; 3) offering mathematical proofs that the security mechanisms do enforce the security policies; and 4) designing and developing prototype browser extensions to test that real-life web applications are supported.

Acknowledgments

This dissertation would not have been possible without the help and guidance of a number of other people. First and foremost, I would like to thank Michele Bughliesi for giving me the opportunity to work at Ca Foscari and supporting me throughout my PhD, both, in research and administrative issues. I appreciate the way he normally used to react to my initial stubborn and impractical ideas when I was new to the world of formal methods. He suggested and motivated me to learn the proof assistant Coq and today I proudly can say it was one of the important decisions towards my bright research carrier. During the later stage of my PhD, I was fortunate to work with Frank Piessence at DistriNet during my internship. We worked together on using secure multi-execution technique to the web browser and he put a lot of efforts in the work that produced very good results in short time. Without his assistance and guidance, I would not be able to use secure multi-execution framework to the web session integrity. In the last meeting before my departure from Belgium, his remarks with words "win win situation" will be on my mind for ever.

During my stay at Co Foscari, I was lucky to have the opportunity to work with Stefano Calzavara during my time at Ca Foscari and even now. We worked together in all the research papers and his guidance and experience have a sound impact on my knowledge of formal models and mathematical reasoning. In the initial stage of my research, I really needed help from an expert person, in particular, when I

was formalizing and reasoning about web sessions security. Calzavara provided me that support off and on during my time at Ca Foscari. Together with Calzavara, we started with preliminary work, but Riccardo Focardi was always there to push our ideas further into much more refined ideas. While working on Featherweight Firefox model, I had many long emails exchanged with Aron Bohannon who helped me in understanding a number of concepts in the Coq model.

I got strong support before and during my PhD from my mother, who prayed for my success and always encouraged me. My friend Anwar Ahmad who gave me his support and motivated me towards higher studies. At Ca Foscari, I met with other nice persons including Teresa Scantamburlo. She is an exceptionally positive and humble person and I really enjoyed conversations with her about Italian culture and sometimes about formal methods research.

I would like to acknowledge the support from the Ca Foscari administration for their grant covering the whole duration of my PhD, European Union for supporting my internship at DistriNet, KU Leuven, Microsoft, NSF and Jane Street for sponsoring me to attend OPLSS summer school in the United States of America.

Contents

List of Figures

List of Tables

Preface

This dissertation is based on the research work carried in the Department of Environmental Sciences, Informatics and Statistics, Ca Foscri University of Venice, Italy during Sept 2011-Aug 2014. Part of the research took place at the DistriNet research group, Katholieke Universiteit Leuven, Belgium.

This research has been built upon the Bohannaon's work: reactive systems and Coq browser model Featherweight Firefox. In the Coq formalisms, the original model has been extended with a number of features or updated by removing or renaming features and many new definitions and theorems have been added. In addition to extended/updated and new definitions, a number of Coq definitions from Featherweight Firefox model have been included in this dissertation to keep the document self-contained. Although, the new additions and updates have been highlighted in the text, a parallel comparison with the original model would clear the differences further.

I am the main author of the Coq developments, have written most of the code in CookiExt and performed all the experiments on real-life web applications using CookiExt to test its usability and security. I helped in defining Flyweight Firefox model and its security enhanced version Flyweight Firefox Plus. Using the semantics of Flyweight Firefox, I encoded the attacks captured by the Flyweight Firefox Plus and helped in debugging the extension SessInt. Finally, I put major contributions to the work where we defined the login history-dependent noninterference for reactive systems and designed an enforcement mechanism for it and I completed the security proof.

1

Introduction

The web has evolved from static pages to web applications that dynamically render interactive contents to enrich the user experience with a number of modern features. To further personalize user's web experience, web applications require user authentication using some *secret* information such as cookies and passwords and combine related network requests and responses together in a *web session* identified by a long random value called *session identifier*. A session identifier, which uniquely identifies a web session, is often stored as a cookie and is then referred to as *session cookie*[1].

To deliver modern web applications, web browsers work with data and scripts from different sources with different trust levels. In addition, web browsers allow the users to open more than one web applications in different browser tabs at the same time. Neither these sources trust each other nor the owner of the browser equally trust them. In such a mutually distrust environment, user's password and session cookies need protection from unauthorized access or tampering, together called as *web session security*. In this dissertation, a formal theory of web security[2] threats is developed in order to verify mathematically the correctness of client-side security mechanism. Formal techniques and proof assistant Coq are used to mathematically verify web session security, mechanisms to enforce session security

[1] A session cookie with the *expires* attribute is called *authentication cookie* [82], however, these terms are used as synonyms in this dissertation.

[2] The term 'security' is used to mean both confidentiality and integrity of web session.

are proposed and prototype browser extensions are implemented to secure web applications at the browser side.

1.1 Key Concepts

Web Session Modern web applications consist of multiple network requests and responses to deliver a specific web service, for example, when the user adds items to the shopping cart on an e-commerce website and then at the end pays for those items. The collection of all the related network requests and responses that forms a web service (e.g., online purchase) is called a *web session*.

Session Cookies Web servers use a long random string value, called *session identifier* to identify requests in a web session. Web servers normally store session identifiers at the users' browser using HTTP cookies called *session cookies*.

Access Control Models In the access control, the computer system's behaviour is changed at specific points in time by putting selective restrictions on *subjects* (processes) to access the *objects* (resources). The Multi-Level Security (MLS) model [12], introduced by Bell and LaPadula in 1970s, was used to prevent users from unauthorized reading of confidential information. Denning showed that the same concept would work if, instead of total ordering, a lattice of levels is used [42]. Biba [14] observed that integrity of information could be dealt with as the dual to tracking confidentiality.

Information-Flow Analysis *Secure information flow analysis* involves performing a static analysis of the program with the goal of proving that it will not leak sensitive information (session cookies). To protect the confidentiality and integrity of session cookies, it requires to specify and enforce rules to en-

sure the browser outputs do not, explicitly or implicitly, leak session cookies and high integrity request is not influenced by low integrity inputs. The first static information-flow certification mechanism was presented by Denning and Denning [43]. *Information-flow policies* [57, 83], on the other hand, regulate that the secret input data cannot be inferred by an attacker by observing the system output. In other words, information-flow policies are end-to-end security policies that define which inputs and outputs are considered secret/less trustworthy or public/high trustworthy.

Information-flow models have been widely applied to protect confidential information [101]. While Biba [14] observed that integrity can be enforced using information-flow models as formal dual to confidentiality, others applied it as dual to confidentiality in the *decentralized label model* [87, 86]. Many others, however, observed that treating information-flow based integrity as dual to confidentiality may yield unsatisfactory and weak notion of security [29, 79]. To strengthen information-flow (noninterference) based integrity policies, Li et al [79], suggested *invariants* on quality of data under program execution. In this dissertation, information-flow based integrity is used as the dual of confidentiality (Chapter 8) and enforced using secure-multi execution [44] technique with the enforcement mechanism proved secure.

Noninterference A program (e.g., web browser) is defined to be noninterferent if its outputs cannot be influenced by inputs at a higher or less trustworthy security level than their own. It is the prevailing basic semantic notion of secure information-flow introduced by Goguen and Meseguer [57]. The intuition is that someone observing the final values of public variables cannot conclude anything about the initial values of secret variables [104]. In other words, secret inputs should not, implicitly or explicitly, affect (or flow into) low outputs. To

capture integrity, it can be stated dually: high integrity outputs should not have been, implicitly or explicitly, influenced by low integrity (less trustworthy) inputs. Language-based techniques are considered to be a promising approach to enforce noninterference [101, 79].

1.2 Threats to Web Sessions

To maintain authenticated web session, session cookie[3] is used as authentication credential based on which the server authenticates each request. Any one who gets a valid session cookie and send it to the server, the request will be authenticated as if it was initiated by the real owner of the cookie, thereby, impersonating the owner. A user can also be impersonated without getting his cookie: the attacker needs a way to force the browser to send requests on behalf of the user in an authenticated session. These two categories of threats to web sessions are broadly categorized as the threats to *confidentiality* and *integrity* of web sessions.

1.2.1 Attacks on Confidentiality of Web Sessions

A session cookie may be submitted over unencrypted protocol in clear text which can be intercepted by the network eavesdropper and replayed to enter the ongoing user's session effectively impersonating the victim user. To force the browser to send a cookie only with requests using encrypted protocol HTTPS, the server sets the cookie with `Secure` flag. A cookie transmitted over encrypted protocol, however, ensures protection against network attackers[4] during that particular request, but it can not ensure protection from network attackers before and after

[3]There can be more than one session cookie in the set of cookies (*authentication token*) required to authenticate the user [25].

[4]As in [7], it is assumed that the attacker, without appropriate certificates, cannot read or modify the content sent over encrypted channels.

the encrypted request. The cookie, for example, may be intercepted later when transmitted over unencrypted channel [21].

In mixed-content websites[5], for example, a non-`Secure`[6] cookie is secure from attackers when included with the request to `https://www.webmail.example.com`, but if the victim navigates to `http://www.other.com` where the attacker injects a frame with its *src* attribute pointing to `http://www.webmail.example.com`, the browser will send an HTTP request including the session cookie in clear and the game is pretty much over [115]. The security flag `Secure` can be used to thwart against network attackers [77], however, most of the web applications do not use this flag properly. In particular, the situation becomes more complicated in websites with partial support for HTTPS [21].

Web attackers [10, 20] put another major threat to the confidentiality of session cookies. Assuming a website is vulnerable to Cross-Site Scripting (XSS) [51, 5, 60] attack, then a web attacker can inject malicious script to the page from vulnerable website which executes later in the browser and leaks the cookie to the attacker. In a reflected XSS attack scenario (Figure 1.1), the user signs into a trusted website (e.g., his bank A) (messages 1−4) and then opens a page in new tab from an evil website E (messages 5−6). In response, E injects a script by redirecting the browser to the bank A (messages 7−9). The injected script reads the cookie registered for website A and leak it to E (message 11) which is used later to hijack the session (message 12−13). Such threats from the web attackers (e.g., XSS attacks) can be mitigated by setting the security flag `HttpOnly` [1]: when JavaScript tries to access a cookie with `HttpOnly` flag set, the result will be an empty string.

The protection mechanisms based on security flags, `Secure` and `HttpOnly`, are supported by most of the modern web browsers, however, most of the web

[5]Websites with partial TLS/SSL [45] support.
[6]A cookie without `Secure` flag.

Figure 1.1: Reflected cross-site scripting attack

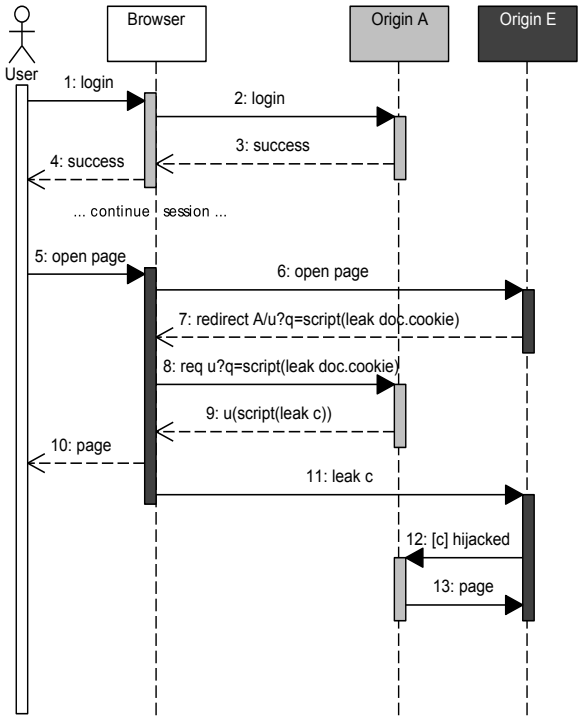
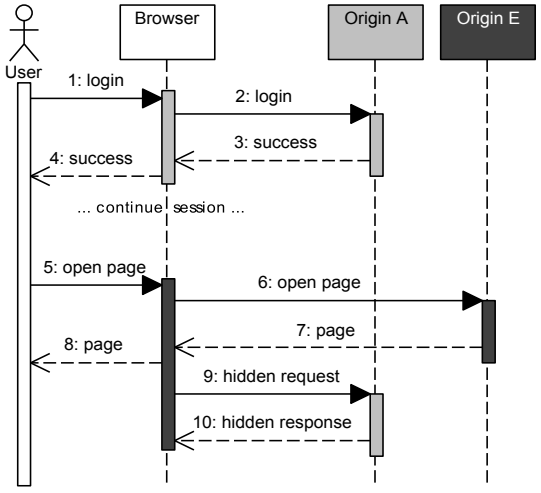


Figure 1.2: Classic CSRF attack



developers [21] do not set these flags properly and hence fail to protect against attacks, on web sessions, from both network and web attackers.

1.2.2 Attacks on Integrity of Web Sessions

Cookies on their own are not sufficient to prevent against the popular class of attacks called Cross-Site Request Forgery (CSRF) attacks [10, 22] The CSRF is a class of attacks where the attacker tricks the user to send authenticated requests using user's credentials (e.g., session cookies).

Consider, for example, the classic CSRF attack scenario in Figure 1.2. The user sings into his bank account at A (messages $1-4$) and then opens another web page in a new tab from a malicious website E (messages $5-8$) which sends a hidden request to A (message 9). This can be done, for example, by setting the *src* attribute of *<img>* tag to a URL of the bank A which loads an implicit request to A and the browser will automatically include the session cookie. If the parameters of the URL to the bank (message 9) are created carefully, it can initiate a bank transfer to the attacker's account as the request is authenticated by the bank A. Even if it is assumed that network communication and session cookies are adequately protected, in addition to CSRF, a number of other attacks are possible such as malicious resource inclusions, session fixation, login and local CSRF and password theft [23, 73].

The problem that is considered in this dissertation is: *how can it be assured that the browser protects the security of the authenticated sessions that it has, for instance, with A (the bank), in the sense that no other web site than A itself can influence authenticated HTTP(S) requests from the browser to A.* Unfortunately, the existing proposals (see Section 2.2 for a detailed survey) in the literature only address very specific classes (e.g., XSS) of known vulnerabilities, often lack rigorous security definitions and proofs and eventually fall short of providing robust

foundations for understanding the real effectiveness of client-side defences against, both confidentiality and integrity, attacks on web authentication. In this dissertation, web security properties are specified and verified mathematically, security policies are enforced at the browser using browser extensions and the results are analysed experimentally.

1.3 Contribution

The main objective of this research is to carry out formal verification of web sessions security and enforce security mechanisms at the browser side. Web sessions face security issues and hence a rigorous analysis using mathematical tools is performed during this research work to ensure protection from attacks on both confidentiality and integrity of web session. This research work consists of mechanized soundness proof of existing protection mechanisms, definition of a novel formal notion for session integrity and proof of security of (a model of the) web browser according to this notion and proof of web session integrity as a noninterference property. In addition to theoretical results, as the proof-of-concept, the proposed mechanisms have been implemented as the browser extensions or in the browser supporting information-flow policies.

1.3.1 Confidentiality of Web Sessions

Soundness of Protection Mechanisms Modern web browsers use protection mechanisms based on the two security flags `Secure` and `HttpOnly` to protect session cookies from being stolen when transmitted over the network or using JavaScript code injected by the attacker. While there is a common belief that these flags provide the security they were designed for, no rigorous study has been carried so far. The first contribution of the research during this dissertation is the for-

malization of web sessions in Coq, security policy based on noninterference that entails properties of both of these protection mechanisms and proof of soundness of the Coq model with respect to this policy. The Coq formalism is built upon the popular browser model Featherweight Firefox [18, 17] by transforming it into an extended version called Extended Featherweight Firefox (EFF).

Enforcing Session Confidentiality in the Browser In the theoretical study of these protection mechanisms, the assumption is that session cookies are properly flagged by the web developers. However, this is not the case in the real world web applications [21] and hence a browser extension CookiExt is developed. CookiExt is intended to guarantee that web applications implement correct security policies, in particular regarding cookie protection. The extension is tested on real-world web applications and the security it can provide is assessed.

Security of CookiExt-Patched Browser Unlike CookiExt, the EFF browser model neither changes the cookie flags nor redirects HTTP requests over HTTPS for the supporting websites. In that sense, the behaviour of the standard browser is different than the browser extended with CookiExt, therefore, the security guarantees of EFF do not apply to EFF patched with CookiExt. Hence, in the next contribution, the EFF model is patched with (a model of) CookiExt and this CookiExt-patched browser model is then proved secure in Coq according the security policy defined in terms of noninterference.

1.3.2 Integrity of Web Sessions

Web Session Integrity as Access Control Property A novel notion of web session integrity is defined and a lightweight version of Featherweight Firefox, called Flyweight Firefox, is introduced which is then extended further to enforce,

using access control/tainting, this notion of session integrity. A proof that a specific countermeasure guarantees session integrity in the formal browser model is also carried. The integrity policy captures common web application attacks such as session hijacking and cross-site request forgery.

Enforcing Session Integrity in the Browser Another browser extension SessInt, the integrity counterpart of CookiExt, is developed to implement the integrity policies in the real browser and its usability and security is analysed. The extension SessInt describes implementation of an approximation of the countermeasures adopted in Flyweight Firefox as a Chrome extension.

Web Session Integrity as Information-Flow Control Both, the CookiExt and SessInt, are extensions to the standard web browsers which do not support information-flow policies. A more permissive client-side enforcements of web session integrity is carried in a browser that supports information-flow security by Secure Multi-Execution (SME) [44, 26]. A simple version of the security policy enforced by SessInt is enforced using SME framework to protect against attacks such as classic CSRF attacks. As proof-of-concept, the theory is implemented as the extension to FlowFox [38] – a browser that supports information-flow policies.

1.4 Methodology

To prove session security against attacks on both confidentiality and integrity of web sessions, security policies are defined, enforcement mechanisms based on information-flow control and access control are developed for them and the mechanisms are then mathematically proved secure. In order to achieve these goals, a model of the standard web browser [19, 18] is used and is then proved that the (model of) web browser is secure according to a security policy. As the proof-

of-concept, an approximation of the theory is enforced by implementing browser extension or is added as a built-in feature in the browser that support information-flow policies. The browser extensions are then tested on real-world web applications and their security and usability are analysed.

1.5 Structure of the Thesis

The next chapter is dedicated to the background materials on web browser and web sessions and research work on their security in the literature. The Chapter 3 include detailed Coq definitions of input and output events of the extended Featherweight Firefox and the confidentiality policy is defined to protect session cookies. The internal state of the browser is defined in the Chapter 4. This chapter include definitions of page, window, cookie, network connection and different stores, such as stores for cookies, network connections, windows and pages. In addition, the waiting and running states of the browser are defined and the main theorem that entails confidentiality of session cookies is defined and proved.

The Chapter 5 explains the design of the CookiExt in detail. Moreover, the methodology of the two sets of experiments is described and then the results achieved are analysed. The formal Coq definitions of the CookiExt-patched EFF are given in the Chapter 6 with the proof of main theorem satisfying the security requirements of the CookiExt-patched browser[7].

Chapters 7 and 8 include the work on enforcing integrity policies. In the Chapter 7, the EFF model is further extended to replicate the features of Flyweight Firefox [23] to enforce meaningful integrity policies using access control/tainting. This chapter also include the design of the SessInt and the experimental results are evaluated. The enforcement of integrity policies using information-flow control

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

and the prototype implementation of such policies in the browser FlowFox [38] is included in the Chapter 8. Finally, the conclusions of this dissertation and some possible future directions are given in the Chapter 8.5.

2

Background and Related Work: Web Browsers and Web Sessions

In this chapter, an overview of the standard web browsers and web sessions is presented along with the major web technologies used in the web platform and mechanisms that govern the security of the browsers and cookies. A survey of research work, in the literature, on web browsers security in general and web sessions security in particular is provided.

2.1 Overview of Web Browser

A web browser is one of the three essential components of the World Wide Web (WWW) used to surf information on the Web – other two components are the servers that supply information to the browsers and computer networks used for browser-server communication. Abstractly, it is a computer software used to retrieve information, available on servers, using computer networks.

2.1.1 Web Technologies

In detailed view, a web browser works as a collection of different standards and technologies [17, 64] including HyperText Transfer Protocol (HTTP) [52, 58], the HyperText Markup Language (HTML) [64, 97, 93, 72], the Cascading Style Sheets

(CSS) [31, 84, 64], the Document Object Model (DOM) [64, 30] and the JavaScript language [64, 48, 53].

In the simple scenario, the user surfs the web by typing the web address `http://www.example.com/figure.jpg` in the address bar of an open browser window. When the user presses the Enter key on the keyboard, the web browser creates a message conforming to the HTTP protocol, gets the IP address associated to `www.example.com`, establishes TCP connection with the machine represented by the IP address, sends it an HTTP request message over the TCP connection established and receives an HTTP response.

HTTP is a detailed specification (protocol) of how client and server should communicate with each other using *request-response* model. In this model, the client initiates an HTTP interaction by sending an (HTTP) request to the server and the server then responds with a response message. Every HTTP request message consists of a start line which include a request method (e.g., GET or POST), request-URI (e.g., */figure.jpg*) and HTTP version, header fields (e.g., Cookie and Referer) and optional message body. The concatenation of the scheme *http://*, domain *www.example.com* and the request-URI */figure.jpg* is known as Uniform Resource Identifier (URI) (e.g., `http://www.example.com/figure.jpg`). A URI with the scheme representing the address of a resource on the web is said to be a URL[1]. The scheme (protocol afterwords) is HTTP when the messages are sent in clear and HTTPS when the messages are encrypted using the protocols Transport Layer Security (TLS)/Secure Socket Layer (SSL) [47, 46]. In a URL, the domain may also follow a port number preceded by a semicolon, otherwise, the default port 80 is used for the TCP connection. Similarly, every HTTP response consists of a status line containing HTTP version, numeric *status code* (e.g., 200) and string *reason phrase* (e.g., OK), header fields (e.g., *Set-Cookie* and *Location*)

[1]In this dissertation, the terms host and domain are used as synonyms and all the URIs are URLs.

and optional message body.

HTML is the family of languages used to write most of the documents communicated between browsers and servers. The web browser (e.g., Google Chrome) retrieves HTML documents and display them as web pages to the user using HTML tags. An HTML document contain tags, that identify the start and end of HTML elements, such as HTML form *<form> elements </form>* which may contain input elements like text fields to get user data (user name and password) and attributes like *action* that identifies the file to process the form-data.

Although, HTML markup can be used for both, the *semantics* and *presentation* of a document, however, normally a different web technology called Cascading Style Sheets (CSS) is used for the later – CSS is used to identify how the contents of the document should be presented to the user. A style sheet may be defined for multiple pages each including a link to the *external* style sheet, added to the head section of an HTML document as an *internal* style sheet or it may be added *inline* to the relevant tag using the HTML *style* attribute. External style sheets can be imported from the source of the page or from the origins different than the source of the page (*cross-origin*)[2].

JavaScript is the client-side programming language supported by most of the modern web browsers. HTML documents (web pages) are using JavaScript programs that can interact with the container document inside the web browser. Among the most popular JavaScript objects used frequently in this dissertation is *XMLHttpRequest* used to exchange data with the server in the background without reloading the whole page. The Document Object Model (DOM), on the other hand, is an API that defines a way for JavaScript to access and manipulate the HTML document. Through DOM API, JavaScript can interact or change the style or contents of the document using the host object *document*. For example,

[2]An *origin* is a triple including the *protocol*, *domain* and *port* number.

JavaScript on the web page can access the password entered into the login form on the page and can write or read the cookies registered by the source of the page[3]. JavaScript code files can be included in the page *inline* or imported from the *external* sources, of the same or different origin, using the *src* attribute of the HTML *<script>* tag.

2.1.2 Web Security Policy

The main policy that controls browser security is the Same Origin Policy (SOP) [99] that constrains the JavaScript access and manipulation of DOM. The simple rule of SOP is: a JavaScript execution context can access the DOM of the other only if their *origins* match. HTTP requests can be initiated by a number of APIs such as HTML forms, *XMLHttpRequest* and HTTP redirects, each with different constraints imposed. The web browser allows requests to any origin, with few constraints on methods and headers [7], if generated by HTTP forms and redirects. Similarly, requests generated by the *<img>* and *<script>* tags are not constrained by the same-origin policy: images and scripts can be loaded from origins different than the page source (*cross-origin*). The *XMLHttpRequest* requests, however, can only be sent to the same origin with two minor tweaks [115]. Even though, the same-origin policy does not allow to share data cross-origin using *XMLHttpRequest* API, sometimes it is desirable to securely share data across the origins. To allow *XMLHttpRequest* API to send and receive cross-origin HTTP requests, an extension Cross-Origin Resource Sharing (CORS) has been proposed [108].

The same-origin policy do not protect cookies (see Section 2.1.3) and web browsers interpret it differently [63]. Furthermore, it does not adequately protect the resources belonging to the user [103, 16] and is ambiguous and imprecise [18, 16]. Even more, it can be circumvented easily if the attacker injects JavaScript code

[3] A cookie can only be read if its security flag `HttpOnly` is not set (Section 2.1.3).

on the page exploiting XSS vulnerability [60]. As an alternative, its replacement with the information-flow control policies has also been studied [17, 15].

2.1.3 Security Policy for Cookies

As HTTP is a stateless protocol [52] with no built-in mechanism to track the user state, modern web browsers instead use cookies to track the user's previous activity (e.g., items added to the shopping cart). An HTTP cookie is a small piece of data, with a number of attributes[4], generated and sent by the server in an HTTP response using *Set-Cookie* header and stored in the user's web browser. The browser then include all the cookies using *Cookie* headers, registered by a site, with each subsequent request to that website [9]. A cookie in the browser store is uniquely identified by a triple cookie *name*, *domain* and *path* value. A server can add, set or delete cookies using HTTP response headers and JavaScript can read and write cookie values using the *document* object.

Most of the users even don't know about cookies, their threats to privacy, whether they are enabled or disabled and how to manage them [85]. Interestingly, cookies are not protected by the main browser's security policy SOP. Cookies do not respect SOP in several ways – web browser allows access to the given domain and its sub-domains regardless of the protocol and port used. A page from a domain can set a cookie for the same and any of its parent domain provided the later is not a *public suffix* [54]. Similarly, the browser include a cookie with a request to a URL if the cookie domain is suffix of the domain in URL and the cookie path is prefix of the path in URL. When the server receives a cookie name and value, it can not determine its attribute values nor the domain from where it was set.

[4]The most popular attributes used in this dissertation are the *name*, *domain*, *path*, `Secure` and `HttpOnly`.

Web server may instruct the browser to put additional constraints on cookies using protection mechanisms by adding independent security attributes `Secure` and `HttpOnly`. Using the `Secure` flag, the server instructs the browser to refrain from including the cookie with HTTP requests over unencrypted protocol (HTTP) to protect it from (network) attackers during transmission. The `Secure` flag, however, does not ensure cookie integrity in the cross-scheme threat model [77, 78]. For example, an active network attacker can set a cookie, with or without `Secure` flag, over HTTP protocol for the same host name as the victim site which is later sent with requests over HTTP(S). The attacker can overwrite the user's session identifier with his or her own, thereby, launching attack on session initialization [10]. The `HttpOnly` flag, on the other hand, prevents JavaScript to access the cookie via *document.cookie* API with the hope to prevent malicious scripts on the page to steal the cookie.

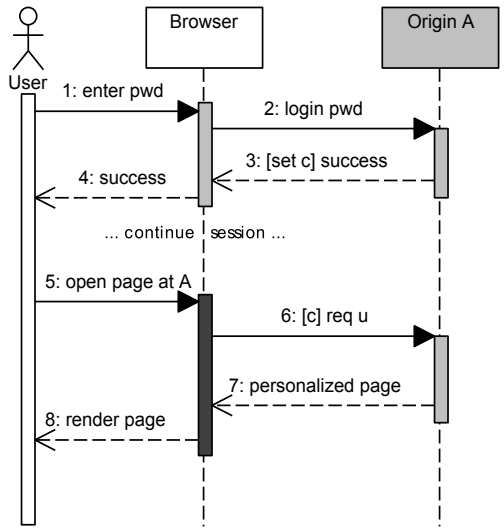
2.1.4 Web Session

To deliver personalized web pages, web applications first require the user to authenticate against web applications and then maintain the user's authenticated state over the series of subsequent HTTP request and response pairs [70]. To authenticate the user, most of the professional web applications use the method password-based authentication or more specifically form-based authentication. In the first step, user provides his user name and password in the login form, which is submitted to the server in an HTTP request using the GET or POST method. If the submitted user name and password match with any of the server's internal records, the user is authenticated.

In the second step, web applications then need to maintain this authenticated state over the series of following HTTP request/response pairs. As HTTP is a stateless protocol, session management is built on top of HTTP using cookies

– the server either promote an existing session identifier (stored as cookie) into an authenticated state or a new cookie is created. In both cases, the (session) cookie serves to combine all the subsequent related network requests and response together into a *web session*. As browser attaches all the cookies registered by a website with request to that website, including the session cookie, the web server authenticates each request by looking at the value of the cookie containing session identifier. All the requests received by the server carrying this cookie value are regarded as authenticated, hence, this cookie serves as authentication credential. Although, there exists other approaches for session management such as URL rewriting or hidden form parameters [98], using cookies to manage web session is the most popular approach adopted by most of the web applications and supported by modern browsers and hence is considered in this dissertation.

Figure 2.1: Web session



Consider, for example, the scenario where a user signs in to his bank account at A using his user name and password (Figure 2.1). The browser submits the user's credentials in a GET or POST request to the website A (messages 1−2).

After verifying the user name and password[5], a session is established and the server associates a long random string as session identifier which is stored as cookie c in the user's browser (messages 3−4). Later on in the same session, when the user submits a bank transaction (e.g., paying bills), the browser will include the session cookie as well and the bank website will verify the user by looking at the session cookie and hence will assume the action of bank transaction was intended by the owner of the account (messages 7−8). As described in Section 1.2.2, this normal browser behaviour can easily be exploited by the attacker.

2.1.5 Reactive System

At the highest level of abstraction, a web browser is modelled as a reactive system [19], which is an event-driven state machine that waits for an input, produces a sequence of outputs in response, and repeats the process indefinitely without ever getting stuck. Formally, a reactive systems is defined as the following:

Definition 1 (Reactive system)**.** *A* reactive system *is 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* $\exists o \in \mathcal{O}, \exists Q \in \mathcal{S} : P \xrightarrow{o} Q$*.*

[5]For simplicity, the user name is omitted.

A reactive system is a constrained labelled transition system that transforms input events into stream of output events. A stream is defined as a coinductive interpretation of the grammar $S ::= [] \mid s :: S'$, where s ranges over stream elements. A coinductive definition of the grammar defines the set of finite and infinite objects that can be built with repeated applications of the term constructors, so a stream is a finite or infinite list of elements [65]. In this dissertation, the attention is limited only to deterministic reactive systems. Bohannon et al. [19] introduced the notion of *reactive noninterference*, which is the classical notion of noninterference tailored towards reactive systems such as web browsers.

2.2 Related Work

There is a large body of research work on web browsers security in general and web authentication security in particular. Johns et al. [70] gives an overview of the attacks on web authentication. The research community has proposed several solutions against these attacks in the last few years, based on server-side countermeasures [10, 60, 69], stronger web authentication schemes [6, 35, 56, 70], or purely client-side solutions [21, 40, 100, 41, 90, 91, 106, 71].

Web session security is often addressed at the browser-side as such defences have a very wide scope and applicability. If a website, for example, does not comply with recommended security practices and/or is affected by a vulnerability, web authentication can often be protected by working solely at the browser's side. Server side defence mechanisms are clearly important and worth of study, since they can precisely fix the root causes of the vulnerabilities and prevent usability issues, but these approaches are considered orthogonal to the endeavours taken in this dissertation. This section is devoted to include work from the literature on formal study of the web browsers and web sessions security.

2.2.1 Formal Verification

Human analysis can be used to detect potential vulnerabilities [33, 80], however, analysis by hand is often too difficult and can miss vulnerabilities in the system that could, otherwise, be captured using formal tools and techniques. Yang et al. [112] showed that formally verified CompCert is more reliable and robust than the non-verified compilers such as GCC and LLVM. After such success stories, it is believed that formal models of web applications and web security mechanisms can reveal practical attacks and are useful to evaluate alternate designs [7]. Moreover, the web platforms continue to grow and hence the importance of using formal and automated tools, to reason about their security properties, increases.

Akhawe et al. [7] built a formal model of web platform and analysed the security of several sample web mechanisms and web applications. Bohannon et al. [19] introduced the notion of *reactive noninterference*, the classic notion of noninterference tailored towards reactive systems such as web browsers and developed a bisimulation-based proof technique to prove that the mechanisms to enforce reactive noninteference are sound. They developed an extensive formalization of the browser, called Featherweight Firefox, as a reactive system in OCaml [18] and proved in Coq that the Featherweight Firefox browser model is noninterferent [17]. Bielova et al. [15] defined end-to-end browser security policies for confidentiality and an enforcement mechanism based on Secure Multi-Execution (SME) [44, 96, 26] was introduced.

Formal and automated proof methods have also been used to other fields, for example, to verify security properties of network protocols [24, 13, 36, 37], mobile applications [27] and web servers [81]. Crafa et al. [34] developed PicNIc: a verification tool, based on models in π-calculus, that automatically checks qualitative noninterference for finite systems.

2.2.2 Web Browser Security

In the literature, a number of techniques have been proposed to improve browser security [11, 59, 28], however, work on using formal techniques to web browser security is included in this section.

To protect against browser-based security attacks launched through JavaScript exploitation, Yu et al. [114] used *program instrumentation* to identify and modify questionable behaviours of untrusted JavaScript and notify the user. They defined a small imperative language, called CoreScript, with big-step style operational semantics. The CoreScript supports some basic Browser Object Model operations (e.g., opening and closing a window) and includes a store of windows and a store of cookies. Yoshihama et al. [113] further extended the CoreScript model with other sophisticated features such as *<img>* and *<script>* tags to load images and scripts from remote resources. The model allows fine grained access control in the web applications to protect mashups and user-generated contents.

In the seminal paper by Akhawe et al. [7], the authors built a formal model of web platform and analysed the security of several sample web mechanisms and applications, using three different threat models. The *backbone* of the formal model include web browsers, servers, scripts, HTTP, DNS and the ways they interact. Using model checking tool Alloy, they introduced formal formulation of web session integrity and found two previously known and three new vulnerabilities. Bansal et al. [8] developed a ProVerif library, called WebSpi, for modelling browsers and web applications. As the authenticity properties in ProVerif are modelled through *correspondence assertions*, it makes the formal verification of certain security properties extremely difficult. If web session integrity is defined in these terms, all the pages of the web server and its authentication goals would need to be defined explicitly making it difficult to provide integrity guarantees for any authenticated session.

As mentioned above, Bohannon et al. proposed to replace the same-origin policy with information-flow control policies defined in terms of (reactive) noninterference. They developed an extensive formalization of the browser, called Featherweight Firefox, as a reactive system and implemented it in OCaml [18]. Featherweight Firefox is a detailed model of a standard web browser which include multiple browser windows, pages, cookies, HTTP requests and responses, simple features of JavaScript, simple model of DOM and basic features of HTML. The inputs of the model include events such as loading a URL in a new window, typing text in a text box, receiving network responses and similarly the outputs include events such as loading and updating a page, displaying error messages and sending network requests. Bohannon proved in Coq that the Featherweight Firefox browser model satisfies end-to-end security policies that ensure *web script security* [17].

Featherweight Firefox model include most of the necessary machinery relevant to cookies and operations on them, however, Bohannon proved noninterference property to ensure *web script security* rather than *web session security.* The Coq version of the Featherweight Firefox allows one to specify information-flow security policies and reason about them making it an ideal option to specify and reason about web session security. The research work in this dissertation is built on reactive noninterference [19], Featherweight Firefox Coq model [17] and reactive noninterference for the browser [15] (see below). Featherweight Firefox model with extended features required to formalize web sessions and security policy to protect against attacks on session confidentiality are discussed in detail in Chapters 3 and 4. The security policies to thwart attacks on session integrity are enforced by further extending Featherweight Firefox and multi-executing the reactive system and are discussed in Chapters 7 and 8.

De Groef et al. used a different approach to web security and developed

a fully functional web browser FlowFox [38] that supports *precise* and *general* information-flow control policies for web scripts based on the technique of SME. Devriese and Piessens [44] proved that SME enforcement technique preserves soundness and precision properties. Later on, Bielova et al. [15] directly applied the secure-multi execution technique to the Featherweight Firefox model achieving a browser model that holds the property of noninterference. They applied SME, in different way than used in FlowFox, to reactive systems such as web browsers: Bielova et al executed multiple copies of the entire reactive system with one sub-execution for each label in the security lattice while De Groef et al. executed each script two times[6]. Similar to Bohannon, they developed a notion of security policy but, unlike Bohannon, they enforced the security policies by executing multiple copies (sub-executions) of Featherweight Firefox model. The information-flow policies, defined and proved sound by both Bohannon [17] and Bielova et al. [15], ensure confidentiality but not integrity.

A common approach adopted in all these proposals is the analysis of a model of the web browser and not the actual implementation. A slightly different approach was used in OP [59] web browser. The authors of OP built a model of their browser kernel and formally verified security properties such as the same-origin policy and address bar correctness using model checker Maude [49]. The design philosophy of OP web browser is to partition the browser into multiple subsystems and interpose all of the communication between these subsystems through the *browser kernel*.

In an effort to fill the *formality gap* between the theory and implementation, Jang et al. [66] introduced a browser QUARK with the structure to ensure all their target security properties. As observed by Jang et al., *fully*[7] formal verification using a proof assistant is a laborious task and hence is infeasible or tremendously

[6]In the former, they used a multi-level lattice while a two-level (High and Low) lattice was used in the later.

[7]Formalizing the full implementation of a system rather than its model.

expensive to verify large and complex software systems such the web browsers. To give *fully formal verification of the browser implementation*, they instead built a small browser kernel where all other browser components were allowed to access system resources only through the kernel. A similar approach, components access to system resources mediated by the kernel, has also been used by Android systems [4]. The QUARK kernel adopts the design strategy *privilege separation* [95] as adopted by other popular browsers Google Chrome [11], Gazelle [110] and OP [59]. Through *formal shim verification*, they formally verified a large system by only verifying a lightweight shim with guarantees the components are restricted only to the allowed behaviours. QUARK kernel is implemented and then proved correct using proof assistant Coq to ensure security properties such as tab non-interference, cross-domain cookie confidentiality and integrity and address bar integrity.

2.2.3 Web Session Confidentiality

Jang et al. [66] proved inter-domain cookie isolation, however, a tab can leak a cookie if the cookie domain matches with or is a sub-domain of the tab domain, making QUARK ineffective against XSS attacks.

A web session may be hijacked by the attacker, exploiting a XSS vulnerability, by injecting a script at the vulnerable web page that steals the session identifier value. The actual vulnerability, in this case, resides on the server side and can be better addressed at the server, for example, by setting the cookie `HttpOnly` flag. However, as experimentally verified [21], most of the web developers fail to follow the recommended practices at the server side: the security flags are not widely adopted in the web applications. These results are similar to the earlier findings by other researchers [90, 106].

Apart from the protection mechanisms based on the security flags, there are *HTML security policy systems* where the security policy is supplied by the website

and enforced by the web browser, such as Content Security Policy (CSP) [105], BEEP [67] and BLUEPRINT [107]. These security policies are concerned more about XSS attacks, however, they have performance and security problems [111]. HTML security policies or server-side solutions in general worth study, however, they are considered only briefly in this dissertation[8].

If the web site operator is not willing or able to fix the server-side vulnerability, a complementary approach would be to protect the user at the browser-side from XSS attacks. The simple rule proposed by Nikiforakis et al. [90], is to prevent JavaScript APIs to access session identifier values. They developed a lightweight mechanism called SessionShield to protect against session hi-jacking. It acts as a proxy between the browser and the network and strips out incoming session cookies from HTTP headers and store them 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.

The SessionShield protection mechanism is very competent, in particular, the idea of relying on a heuristic to identify session cookies which is also used, with slight modifications, in the implementation of CookiExt (Chapter 5). 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 [106] – 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.

Kirda et al. proposed a client-side Web proxy, called Noxes [75], to prevent malicious scripts to send cookies by dynamically encoding them in requests to

[8]Getting policy information from the server result precise policies and is needed for compatibility reasons. A proposal for defining such policies is given in Chapter 8.

the attacker's server. Unlike SessionShield, Noxes allows JavaScript to access session cookie, but prevents them to leak cookies to the attacker. Noxes rely on the assumption that requests to external domains generated by JavaScript are not trustworthy and hence are blocked. It *implicitly* considers the external links embedded, such as values of *href* and *src* HTML attributes, *url* identifier in CSS and local links on a web page, safe with respect to XSS attacks. As observed by Nikiforakis et al. [90], the policy implemented by Noxes is not compatible with several web applications and is not complete: an injected HTML tag which statically references a URL with session identifier value, can leak the session identifier to the attacker. Moreover, Noxes does not prevent the network attacks such as intercepting the unencrypted data including the session cookies.

To protect against network attackers, a particularly relevant client-side defence is HTTPS Everywhere [94]. 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 Force-HTTPS [62], a proposal aimed at protecting high-security websites from network attacks.

Adida [6] proposed a fascinating approach called SessionLock to protect against network eavesdroppers. The SessionLock protocol is best suitable for web applications that offer initial login operation over HTTPS and then delivers the rest of web pages over HTTP. In this protocol, after the user signs into the website using

SSL, a session *secret* is stored in the browser over HTTPS as the cookie which is never sent in clear but communicated from the initial login page over SSL to each subsequent page over HTTP using the URL fragment identifier. Each subsequent HTTP request is then timestamped and HMAC'ed with the secret, received during initial login phase, for authentication. As admitted by Adida, SessionLock protocol fails against network attacks: an active attacker can trivially inject code in a plain HTTP URL that can steal the session secret and hijack the session. Furthermore, similar to HTTPS Everywhere, SessionLock protocol does not prevent XSS attacks.

2.2.4 Web Session Integrity

Akhawe et al. [7] developed an Alloy model of the web platform and defined session integrity as the property that *no attacker is in the causal chain of any HTTP request belonging to the session.* The underlying model does not have a sufficiently detailed representation of scripts to study other application-level session integrity issues. Moreover, the property is very syntactic, so it is hard to generalize it to new settings and carry out a precise comparison with the session integrity definitions introduced in this dissertation (Chapters 7 and 8). It was observed that the definition in [7] is only concerned about web attackers entering the causal chain. It would be difficult to extend the notion to deal with network attackers as they can enter the causal chain of any transaction which includes at least a communication over HTTP and trivially violate session integrity. Moreover, the security properties require browser-server interactions and hence are not amenable to be enforced at browser side without server interactions.

To prevent CSRF attacks in QUARK, cookies are not included with the request if the site domain suffix does not match with the tab domain, however, this lead to major compatibility issues with applications such as Mashups [66]. A similar

approach, with the same compatibility problems, was used in RequestRodeo [71] which selectively removes authentication credentials (e.g., cookies) from outgoing requests. There are a number of others proposals that share the same idea of stripping authentication cookies from (selected class of) cross-site requests including [40, 100, 82], thus making CSRF attacks largely ineffective.

In the design by Philippe et al. [100], all the session and authentication information are stripped out only from the *malicious* cross-origin requests. They have designed and implemented an algorithm to *precisely* identify *expected* cross-origin requests and allows the session information with such requests. The algorithm relies on previous collaboration (e.g., requests in the past using POST method) between the websites. They formally verified, using bounded-scope model checking, that the algorithm protects against CSRF attacks under the specific assumptions about the way the sites legitimately collaborate with each other. However, the algorithm is based on heuristics which eventually is not accurate and the verification excludes from the threat model both XSS flaws and network attackers, which instead are two important aspects that are considered in this dissertation.

Serene [41] is a browser-side solution against session fixation [76, 69] attacks. The core idea is to instruct the browser to attach to outgoing HTTP(S) requests only those authentication cookies which have been set via HTTP(S) headers, thus preventing cookies set by a malicious script from being used for authentication. Serene does not enforce protection against network attacks, since network attackers can arbitrarily overwrite any cookie just by forging HTTP responses [9, 20]. The design of Serene has not been formally validated.

As the HTTP *Referer* header contains the URL which initiated the request, it can be used to distinguish the same-site requests from cross-site requests and hence can be used to defend against CSRF attacks [10]. However, the *Referer* header may contain sensitive information (e.g., the search query) violating the privacy of the

users [61]. Due to the privacy reasons, the header is sometimes suppressed which hampers the technique, especially, if the suppression is widespread. To resolve the issues with privacy, Barth et al. [10] proposed to send an *Origin* header with POST requests where the *Origin* stores the origin that initiated the request. This technique restricts the use of *Referer* header by sending the *Origin* header only with POST requests and includes only the necessary information (protocol, host and port) to identify the source. A request with *Origin* header set to a different (source) origin than the destination origin is rejected by the server.

Bortz et al. [20] introduced a new HTTP header called *Origin Cookie* as a lightweight solution for protecting web sessions, to provide stronger integrity guarantees than standard cookies, based on origin isolation. Origin isolation is a sound security principle, in particular against *related domain* and network attacks, where cookies are isolated based on their origins. Using the header *Origin Cookie*, a cookie can only be sent with request to an origin set by the response from exactly the same origin. A cookie, for example, set over HTTP can not be sent with requests over HTTPS. Origin cookies do not solve the problem of protecting the first authentication step, i.e., when the password is sent from the browser to the server. Moreover, origin cookies do not directly support mixed HTTP/HTTPS websites and does not solve all the potential problems affecting cookie-based authentication: for instance, non-`HttpOnly`[9] origin cookies can still be leaked via XSS. Li et al. [79] classifies integrity as *program correctness* and compares information-flow and access control enforcements for integrity.

[9] A cookie without `HttpOnly` flag set.

2.3 Web Session Security Using Coq

The Coq [32] development accompanied with this dissertation uses a *locksetp* unwinding relation as the definition of web session security, which in turns rely on the definition of reactive systems and policies. The structure of reactive systems can be defined using the Coq module systems. A Coq module is a set of definitions for identifiers and constraints on the terms in the module with the use of keyword `Axiom`. One can write down the type (or signature) of a module, which is simply a set of type declarations for the identifiers that must be present in the corresponding module implementation. Coq can check that a given module respects a given signature.

A module signature can be used to describe the form of a mathematical structure (e.g., reactive system) and a module satisfying that signature (`ReactiveSystem`) can be used to define a particular instance of that structure. In order to build a module satisfying a module signature with an axiom, the module must also provide a proof that the axiom is true for the given definitions of the identifiers. Thus, defining a module type (signature) is similar to defining a theorem and a module satisfying that signature is the proof of the theorem [17]. An example Coq module signature from Featherweight Firefox Coq model (and used in EFF) is given in Figure 2.2.

Figure 2.2: `IOEevents` signature

```
1 Module Type IOEvents.
2   Parameter input_event: Type.
3   Parameter output_event: Type.
4   Definition event := input_event + output_event.
5 End IOEvents.
```

As a theoretical tool, reactive systems work well, but they can not be directly defined conveniently as exactly one output can be released in one step taken from

a producer state. The convenient way is to define a system that allow some finite, possibly empty, sequence of outputs to be produced on each step taken from a producer state. The *multistep reactive systems* [17] does exactly that, where one step of the system corresponds to multiple steps in a standard reactive system. The same approach as Bohannon is used where a translation from multistep reactive system to standard reactive system is constructed. This technique is using the translation from one system to the other that has been defined as a Coq functor. A reactive system is created that will contain a multistep system and will advance it one step and stores the outputs in a buffer and then the reactive system will release the buffered outputs one at a time or will insert a dummy output if the multistep system produced none.

The accompanying Coq code include definitions of module signatures for I/O events, policy, reactive system, multi-step reactive system, secure reactive and secure multi-step reactive systems. Defining a module confirming to the module signature for secure reactive system will be the proof of reactive system security and similar approach based on Coq module feature is used to prove the security of multi-step reactive systems and secure transformation of the later to the former.

3

Web Session Confidentiality: Browser Input Output

The Featherweight Firefox model enforces web script security but not web session security. Its Coq implementation [17] is very huge (more than 50,000 lines of code), however, it still misses a number of interesting features. In this chapter, the original Featherweight Firefox model in Coq is further extended with a number of features (henceforth called Extended Featherweight Firefox (EFF)) to add the support for policies to protect client authentication based on session cookies. In order to support such policies, browser features like support for HTTPS, `Secure` and `HttpOnly` cookie attributes, redirects, *Referer* header and so on are added. In the first part of this chapter, the (updated) definitions of input and output events of the EFF model are included and in the second part of this chapter, the information-flow security policy to protect session confidentiality is defined.

3.1 Input Events

A web browser interacts with its external environment through input and output events – it takes inputs both from the user and network and similarly delivers outputs to the user and network. User inputs come from the user while interacting with the browser such as clicking a link on a page and network inputs come from

the network such as responses to HTTP(S) requests. The type `input_event` is inductively defined in Figure 3.1.

Figure 3.1: `input_event` data type

```
1  Inductive input_event: Type :=
2    | user_load_in_new_window_event:
3       user_win_id → url → input_event
4    | user_load_in_window_event:
5       user_win_id → url → input_event
6    | user_close_window_event:
7       user_win_id → input_event
8    | user_input_text_event:
9       user_win_id → nat → String.t → label → input_event
10   | network_document_response_event:
11      net_conn_id → user_win_id → resp → input_event
12   | network_script_response_event:
13      net_conn_id → resp → input_event
14   | network_xhr_response_event:
15      net_conn_id → resp → input_event.
```

The Coq keyword `Inductive` is used to inductively define data types of some sort (`Type` in this case). The first four type constructors represent user input events (lines 2−9). The constructor `user_load_in_new_window_event` corresponds to the event when a user opens a new window and navigates it with a URL or opens a link in a new window. The overall external structure of `input_event` is kept as original except the first constructor for `user_load_in_new_window_event` event, where the parent (previous) user window identifier `user_win_id` is added.

The `user_win_id`, which wraps a natural number in a record type, refers to a browser window. It tracks the previous window in case the user opens a link in a new window, which in turn is needed to track the referrer of the request. For example, when the user right clicks on a standard HTML link and opens it in a new window. As EFF include HTTP *Referer* header, the field `user_win_id`, which points to the source (previous) window, is used to extract the URL of the source window which is used as the referrer of the request. If the user opens a blank

window and navigate it to a URL, the referrer is blank. The second input event `user_load_in_window_event` represents the user event navigating a window to a specific URL. This occurs, for instance, when the user types a URL in the address bar of the existing window or when the user clicks on a standard HTML link on a page in a window.

The third user input `user_close_window_event` represents the event when a user closes an opened window. The event `user_input_text_event` represents the user event of typing some text in a text box on a page. Its first parameter is the window where the text box lies on a page. The natural number represents a text box number according to its position on that page. The third parameter is the text user enters into the text box. This, for example, models event when the user types the password in a login form on the page in the window. The fourth parameter is the security label[1] which represents the user's intentions for the visibility of this input event. A similar label is derived for other security critical events (such as network events discussed below) from the URLs of these events. The definition of type `url` is given in Figure 3.2.

Figure 3.2: `url` data type

```
1 | Inductive url: Type :=
2 |   | blank_url: url
3 |   | http_s_url: protocol → domain → req_uri → url.
```

The two constructors of the type `url` correspond to the two types blank and HTTP(S) URLs. To accommodate both HTTP and HTTPS URLs, the parameter `protocol` (Figure 3.3) is added to the constructor `http_s_url`. In EFF, the security policy and enforcement mechanisms treat `https:URLs` and `http:URLs` isolated from each other with different security labels and hence with different constraints

[1]The security label (Figure 3.12) represents the attacker's capability.

on them (Section 3.3). In addition to HTTP and HTTPS protocols, type `protocol` also contains `about_protocol`. As `blank_url` gets different label, this is needed in cases (for example in case analysis on `url`) where the `protocol` is to be extracted from the `blank_url`. No security properties related to `about_protocol`, however, are studied in this dissertation.

Figure 3.3: `protocol` data type

```
1 | Inductive protocol: Type :=
2 |   | about_protocol: protocol
3 |   | http_protocol: protocol
4 |   | https_protocol: protocol.
```

The second parameter of the constructor `http_s_url` is the type `domain` which, as in original model, is handling atomic domain names without checking for subdomains. The `req_uri` is a record type modelling the `path` and query string parts of the URLs. Both the `domain` and `path` are used for accessing the cookies registered for the `url`.

The last three constructors of `input_event` represent the three types of network responses corresponding to the three network requests: document, *XMLHttpRequest* and script, respectively. All of these network input events include a `net_conn_id`: a record type including the type `url` and a natural number. It identifies the network connection over which the HTTP(S) request was made to the URL. As there can be more than one connection to the same URL, the natural number is used to uniquely identify the connection. In the model, the parameter `net_conn_id` of network responses is used to match up the HTTP(S) response to the corresponding HTTP(S) request. The definition of `net_conn_id` is shown in Figure 3.4.

In addition to the `domain` name and the natural number that refers to the most recently opened network connection to that `domain`, the `protocol` of the URL

Figure 3.4: `net_conn_id` data type

```
1 Record net_conn_id: Type :=
2   build_net_conn_id {
3     net_conn_id_url: url
4     net_conn_id_value: nat
5   }.
```

through which network request was made is used to assign security label to the event. A higher label (`https_label d`) is assigned to network event over HTTPS from domain `d` than to event with HTTP protocol which is assigned (`http_label d`) label. This is required to assert in the security policy that in case of encrypted communication, the attacker needs to be more powerful to decrypt the messages. The type `label` is discussed in Section 3.3 in more detail. The other part, `req_uri`, of the `url` in `net_conn_id` is redundant, however, it is kept as it is part of the type `url`.

The event `network_document_response_event` includes `user_win_id` which represents the window into which the content of the response (e.g., web page) is intended to be loaded. All of the network input events include the type `resp` modelling the headers and body of the HTTP responses as shown in Figure 3.5.

Figure 3.5: `resp` data type

```
1 Record resp: Type :=
2   build_resp {
3     resp_del_cookies: StringSet.t;
4     resp_set_cookies: StringMap.t cookie_flags_value;
5     resp_redirect_uri: option url;
6     resp_file: file
7   }.
```

The `resp_del_cookies` field (line 3) contains an unordered set of strings – the names (keys) of the cookies that should be removed from the browser's cookie

store. Web sites can store session information as cookie value at the browser using HTTP *Set-Cookie* header in HTTP(S) responses modelled as `resp_set_cookies` (line 4). The `resp_set_cookies` field contains a finite mapping from string cookie names to a record type `cookie_flags_value` – these are the cookie mappings that should be added to the browser's cookie store. Although, the mapping does not include domain and path information, but cookies must be interpreted as relative to the `domain` name and `path`. Both of these values are taken from the URL that is used in the request for the resource.

Usually, cookies can be set for domains other than that of the original host [20]. For example, a sub-domain `subdom.example.com` can set a cookie for the domain `example.com` (e.g., a cookie with domain name `example.com`). According to the *domain-match* [77] rule, this cookie can be then included in requests to all sub-domains of `example.com` (such as `app1.example.com`). In EFF model, sub-domains are not modelled – a cookie with a domain can only be set if the cookie domain exactly match with the domain of the response URL. Including sub-domains would be an interesting addition to reason about *related-domains* attackers as described in [20].

The EFF model include HTTP redirects – a feature included in related models [7, 8] which have been shown to have a significant impact on browser security. The field `resp_redirect_uri` in Figure 3.5 (line 5) has value of type `option url` which represents an optional redirect URL where the browser is redirected. An HTTP(S) response has status code, such as codes for content result and redirect, as modelled in the formal model by Akhawe et al. [7]. In EFF model, however, the optional `resp_redirect_uri` filed in the response `resp` is used to represent both cases. Despite of different styles, both achieve the same objectives – if the value of `resp_redirect_uri` is `Some url`, it is interpreted as a redirect to URL `url` otherwise the response is processed as normal.

Figure 3.6: `cookie_flags_value` data type

```
Record cookie_flags_value: Type :=
  build_cookie_flags_value {
    cookie_flags_value_secure: bool;
    cookie_flags_value_httponly: bool;
    cookie_flags_value_value: String.t
  }.
```

The type `cookie_flags_value` (shown in Figure 3.6) is one of the important additions to the model and hence is discussed in more detail. Since EFF supports HTTPS and JavaScript in the model can now access cookies (Figure 3.9), to define interesting security policies, the model is extended with cookie attributes `Secure` and `HttpOnly`. The first two fields, `cookie_flags_value_secure` and `cookie_flags_value_httponly`, of type `cookie_flags_value` models the `Secure` and `HttpOnly` flags, respectively while the third field `cookie_flags_value_value` contains the cookie value of type `String`.

Servers that support TLS/SSL protocols can protect cookies against eavesdroppers and *man-in-the-middle* attacks by setting cookies with the `Secure` attribute (there are tools, though, such as SSLStrip [50] that can pose HTTPS stripping attacks). The browsers include `Secure` cookies only with requests over secure communication. This restriction on cookies in EFF is captured by adding the facility of cookie flag `Secure` and accordingly security rules in the security policy (Section 3.3).

In case of origin cookies [20], the `Secure` flag is not needed to protect cookies. The authors, instead, enforce the strict same-origin policy for cookies using, in addition to *Cookie*, a new cookie attribute *Origin* using *Origin-Cookie* header isolating cookies based on their *origins*. This mechanism, however, poses compatible problems to many applications that share cookies between schemes (use both secure and insecure communication as in mixed-content websites) and related

domains (sub-domains) as their main features.

Similarly, cookies with `HttpOnly` flags are not included if the cookie-string is being generated for a non-HTTP API [9, 77]. In other words, JavaScript is not allowed to access `HttpOnly` cookies. Hence, the flag `HttpOnly` can be used as a partial mitigation for XSS by preventing script from accessing cookies in the web browsers by setting cookies with `HttpOnly` flag [1]. Both of these attributes, `Secure` and `HttpOnly`, are of type `String` [9], however, to make the reasoning in Coq simple, instead the type `bool` is used without affecting the security goals these attributes achieve.

Figure 3.7: `file` data type

```
1 | Inductive file: Type :=
2 |   | empty_file: file
3 |   | html_file: list doc_tree → file
4 |   | script_file: script → file.
```

The body of the HTTP(S) response is modelled as a data type `file` as shown in Figure 3.7. There are three types of files the browser may receive: an empty file, a document file, or a script file. The `doc_tree` data type (shown in Figure 3.8) is a recursive data type corresponding to the (parsed) HTML document. All of the document tags include an optional `elt_id` which corresponds to the *id* attribute in HTML. The `inl_script_doc` corresponds to the HTML <script> tag that is used to add a script in-line in an HTML document. The `rem_script_doc` corresponds to the HTML <script src> tag that is used to include a script into a web page by instructing the browser to retrieve it from a remote source, typically using HTTP(S) request. The `textbox_doc` constructor represents a text box on a web page that can be updated by the user and the `div_doc` constructor corresponds to the HTML tag.

The parameter `script`, in both `file` and `doc_tree`, is one of the most im-

Figure 3.8: `doc_tree` data type

```
Inductive doc_tree: Type :=
  | inl_script_doc: option elt_id → script → doc_tree
  | rem_script_doc: option elt_id → url → doc_tree
  | textbox_doc: option elt_id → String.t → doc_tree
  | div_doc: option elt_id → list doc_tree → doc_tree.
```

Figure 3.9: `script` data type

```
Inductive script: Type :=
  | null_script: script
  | nat_script: nat → script
  | str_script: String.t → script
  | url_script: url → script
  | code_script: script → script
  | app_script: script → script → script
  | var_script: var → script
  | fun_script: var → list var → script → script
  | eval_script: script → script
  | seq_script: script → script → script
  | get_cookies_script: script
  | set_var_script: var → script → script
  | xhr_script: script → script → script → script
  | self_script: script
  | get_win_root_node_script: script → script
  | new_div_node_script: script
  | remove_node_script: script → script
  | insert_node_script: script → script → script → script.
```

portant types and is shown in Figure 3.9. The data type `script` represents the abstract syntax of scripting expressions. It does not capture all of the details of the JavaScript language, however, it contains most of the JavaScript features needed to put important constraints on the implementation of the model as a whole. It is further extended with a new script expression `get_cookies_script`, which is added in particular to capture XSS attacks and is discussed in more detail below. The rest of constructors of type `script` are as original and their detail can be found in Bohannon's thesis [17].

As mentioned in the beginning of this section, EFF include `Secure` and `HttpOnly` cookie attributes. To depend against XSS [51, 5] attacks, a cookie is set with `HttpOnly` flag and hence can only be accessed through HTML code and no access to such cookies is given to JavaScript [9, 77]. To capture this notion and prove related security properties, the script expression `get_cookies_script` is added to EFF. Modelling the JavaScript command *document.cookie*, this script expression residing on a page from a `domain` can access all non-`HttpOnly` cookies registered for that `domain`, as permitted by the SOP [99]. The retrieved cookies make up a string with cookie key-value pairs are separated by semi-colons and the string is then stored in the `str_script` expression which may latter be leaked to unrelated domains using network requests.

3.2 Output Events

The type `output_event` (Figure 3.10) models the user interface of the browser model. Like the input event, the first category of events comprises the browser outputs to the user. They include events of opening and closing a window, loading and updating a page and displaying a string error message on the screen. The type `rendered_doc_tree` represents the elements of the HTML document that

are supposed to be visible to the user. For example, it does not not include inline and remote script elements. Further details of the user outputs can be found in [17], however, the last three output events are more important in terms of web sessions security and hence are explained below.

Figure 3.10: `output_event` data type

```
1  Inductive output_event: Type :=
2    | ui_window_opened_event: output_event
3    | ui_window_closed_event: user_win_id → output_event
4    | ui_page_loaded_event:
5       user_win_id → url → option rendered_doc_tree → output_event
6    | ui_page_updated_event:
7       user_win_id → option rendered_doc_tree → output_event
8    | ui_error_event: String.t → output_event
9    | network_document_request_event:
10      user_win_id → net_conn_id → req → output_event
11   | network_script_request_event: net_conn_id → req → output_event
12   | network_xhr_request_event: net_conn_id → req → output_event.
```

The last three constructors represent network requests for network document, script and data loaded using asynchronous request. All of the three network output events include `net_conn_id` and `req` and each event gets a `label` depending on the `protocol` used in the corresponding HTTP request and the destination `domain` of the request. Unlike the input events, the network connection identifier in `output_event` is not used to match up the requests with the corresponding responses but to get the `domain` and the `protocol` used for the request to label it. There are different security labels for HTTP and HTTPS protocols, therefore, more sophisticated security policies (Section 3.3) can be defined by putting constraints on output events based on the protocol used.

The definition of type `req` is shown in Figure 3.11. It represents the content of an HTTP request. The first field `req_req_uri` of type `req_uri` is the URI of the requested resource. Similar to Bohannon [17], the theoretical models are

Figure 3.11: `req` data type

```
Record req: Type :=
  build_req {
   req_req_uri: req_uri;
   req_cookies: StringMap.t String.t;
   req_referer: url;
   req_body: String.t
  }.
```

limited only to the most frequently used request methods GET and POST and are used interchangeably. Considering a distinction between these methods would be useful to thwart CSRF attacks [100, 40], however, different approaches (Chapters 7 and 8) are taken in this dissertation to prevent against such attacks. The second field represents a map from `String` cookie name to `String` cookie value, which corresponds to the *Cookie* HTTP header [9] used to send the stored cookies to the server. For an HTTP(S) request to `domain`, the field `req_cookies` contain all the cookies registered for that `domain`. Note that, web browser include cookies according to the cookie flag values: for example, the browser does not send `Secure` cookies with unencrypted request.

The browsers normally allow to attach cookies to HTTP(S) request to URL with a host equal to, or a sub-domain of, the cookie's domain. Similarly, non-`Secure` cookies received in clear from `http://example.com` are included with encrypted requests to `https://example.com`. This lack of isolation based on sub-domains and scheme may result *related-domain* attacks by compromising session integrity [20] and hence worth study in future extension of the model.

The third field `req_referer` is an extension to the model to include HTTP *Referer* header. This feature has been included in related models [7, 8] which have been shown to have a significant impact on browser security. An `http_s_url` URL represents the referrer of the request while a `blank_url` represent empty

header. Including information about the referrer in the request might have security implications and needs to be carefully dealt with. The address of a page having (source of) a link might be private information or might reveal an otherwise private information source (for example, Alice might not want the service provider Bob to know David who had been referred Alice to this service). In such cases, it is strongly recommended to get the user choice to whether or not the referrer be revealed [52]. Moreover, the RFC 2616 [52] suggests that the referrer should not be included in a non-secure HTTP request if the referring page was transferred with a secure protocol. Such security measures would be trivial addition to the model though, however, the first one would create many user dialogue messages which eventually would lead the user to ignore them.

In case of redirects, an interesting question would be whether to keep the original referrer or replace it with the source of redirect response. For example, if a user clicks on a link that points to domain X on a page from domain A and the request is redirected to domain B. In such redirect case, the original referrer (domain A) is kept. This is not specified in the relevant RFC document [52], however, most of the browsers (e.g., IE8, Safari4, FF3.6.10 and Chrome5) keep the original referrer.

The last field `req_body` of type `req` represents the actual string message body of the HTTP(S) request.

3.3 A Confidentiality Policy

After events of the browser are defined, its now time to define a precise confidentiality policy over the input and output events of the browser to secure session cookies. To start with, the attacker is characterized using security *labels* where higher label in the lattice accounts for more power. The principals, including attackers, are

modelled as parties that can see fragments of the stream of input/output events to/from the browser and the security label represents the capabilities of the attacker.

Figure 3.12: `label` data type

```
1 Inductive label: Type :=
2   | top_label: label
3   | https_label: domain → label
4   | net_label : label
5   | http_label: domain → label
6   | bot_label: label.
```

The fundamental part of the security policy defined here is the type (set) of security levels defined as the inductive type `label` (Figure 3.12). The intuitive meaning of the `label` is that the input/output events and data having `label` defined on a domain is intended to be shared only between that domain and the user. The security label used here is much more flexible similar to the *decentralized label model* [87]: a model used in *Java Information Flow* (Jif) (previously known as JFlow) [86, 88] language to enforce information-flow security policies in a mutual distrusting environment.

The labels defined over domains (lines 3 and 5) are categorized based on the protocol of the URL used to communicate with that domain. The domain labels associated with HTTP protocol are represented with constructor `http_label` and those with HTTPS protocol are represented with `https_label`. Some principals can only read HTTP traffic sent to a particular domain, while others can read all unencrypted network traffic. Traffic over HTTPS, on other hand, is only visible to the most privileged principal (per domain).

The sort `label` forms a partial order, where the relation `label_lt_equiv` is defined in Figure 3.13 as the Coq proposition of sort `Prop`. Such definitions in Coq can be read as sets of inference rules, where each constructor corresponds to one

Figure 3.13: `label_lt_equiv` function

```
1  Inductive label_lt_equiv: label → label → Prop :=
2    | top_label_lt_equiv: ∀ l, label_lt_equiv l top_label
3    | label_lt_equiv_refl: ∀ l, label_lt_equiv l l
4    | net_https_lt_equiv:
5        ∀ d, label_lt_equiv net_label (https_label d)
6    | http_net_lt_equiv:
7        ∀ d, label_lt_equiv (http_label d) net_label
8    | http_https_lt_equiv:
9        ∀ d1 d2, label_lt_equiv (http_label d1) (https_label d2)
10   | bot_label_lt_equiv: ∀ l, label_lt_equiv bot_label l.
```

rule of inference. The security label (`http_label d`) corresponds to the standard view of a web attacker: this attacker controls the web server at domain `d`, but has no network capability. A network attacker, instead, resides at level `net_label` and is stronger than a web attacker, since it has the ability to inspect the contents of all the unencrypted network traffic. The label (`https_label d`) corresponds to more powerful attacker which may have fully compromised a running web server: an attacker at this level has all the capabilities of a network attacker and can also decrypt all the encrypted network traffic sent to the domain `d`.

Furthermore, two labels associated with different domains are considered incomparable in the partial order (for example, `http_label d1` is not related to `http_label d2`, where `d1` $\neq$ `d2`). Finally, there are labels that are not defined on domains. The `bot_label` is less than all other labels which is associated with the content that is public to all principals at any security level. Similarly, the label `top_label` is greater than all other labels which is associated to private contents.

To define precise security policies, labels need to be associated with input and output events. The network input and output events are given labels based on the destination domain and the protocol in the request/response URL. The function `url_label` in Figure 3.14 defines a mapping from URLs to labels. It

Figure 3.14: `url_label` function

```
1  Definition url_label (u: url): label :=
2    match u with
3    | blank_url ⇒ top_label
4    | http_s_url p d _ ⇒
5       match p with
6       | about_protocol ⇒ top_label
7       | http_protocol ⇒ http_label d
8       | https_protocol ⇒ https_label d
9       end
10   end.
```

assigns `top_label` to the URL of a blank page (`blank_url`). The non-blank URL can either be with protocol HTTP or HTTPS, representing non-secure and secure communication, respectively, which are assigned different security labels based on the protocol. The URL with HTTP protocol is given label (`http_label d`) which is below in lattice (Figure 3.13) than the label (`https_label d`) for URL with HTTPS protocol. The label `top_label` is assigned to the URLs with `about_protocol`. As noted in Section 3.1, this is needed to ease reasoning when case analysis on `url` or `protocol` is used, where a URL with `about_protocol` (a non-reachable case) is treated as the `blank_url` and is assigned the `top_label` as in the model by Bohannon [17].

The intuition of the label is to represent the minimum point of observation. The components of security policy for confidentiality pertaining to the label represent the aspects of the input events (e.g., session cookies) that the browser must hide relative to the points of observation. On the other hand, the policy components pertaining to the output events describe the aspects of the output streams that are assumed to be hidden from the points of observation. To elaborate this notion further, the confidentiality policy for input and output events is defined in the next subsections.

3.3.1 The Policy for Input Events

The most important part of (network) input events to protect is the cookies. After extending the model with security flags `Secure` and `HttpOnly` for cookies, more interesting information-flow security policies can be defined for input and output events. The tricky part of security policy for network events is to assign security labels to cookies. The definition of the function assigning labels to cookies is given in Figure 3.15.

Figure 3.15: `cookie_label` function

```
1 | Definition cookie_label (d: domain) (zs zh: bool) : label :=
2 |   match zs, zh with
3 |   | false, true ⇒ http_label d
4 |   | true, true ⇒ https_label d
5 |   | _, _ ⇒ bot_label
6 |   end.
```

The function `cookie_label` assigns the label (`http_label d`) to `HttpOnly` cookie registered by the domain `d`. The label for `Secure` and `HttpOnly` cookie registered by the domain `d` instead is (`https_label d`) and it is `bot_label` otherwise. This sitting does not allow to set `Secure` cookies using non-secure communication. A cookie with `Secure` flag set using unencrypted communication is not secure in first place, even if the browser include it only with requests over encrypted communication. Moreover, a cookie with `Secure` flag is assumed to have always `HttpOnly` flag set: this is not specified by the standard and some web applications do not follow this assumption, however, it is adopted to strengthen security. This obviously results some usability issues which are further discussed in the Chapter 5.

Based on the cookie label, the visibility of the cookie is defined by the predicate `is_vis_resp_cookie` in Figure 3.16. The function `erase_invis_cookies` (Figure 3.17) gets a label, URL of the network input event and a map from `String` key

Figure 3.16: `is_vis_resp_cookie` function

```
Definition is_vis_resp_cookie (l: label) (d: domain)
  (ck: cookie_flags_value): bool :=
    cookie_label d ck.(cookie_flags_value_secure)
    ck.(cookie_flags_value_httponly) <?= l.
```

to `cookie_flags_value` and erases confidential cookies (at the label) from the network input event. The map represents the *Set-Cookie* header of the network response and the record type `cookie_flags_value` include `Secure` and `HttpOnly` flags and the `String` cookie value (Figure 3.6).

Figure 3.17: `erase_invis_cookies` function

```
Definition erase_invis_cookies (l: label) (u: url)
  (rssc: StringMap.t cookie_flags_value)
  : StringMap.t cookie_flags_value :=
    let '(invalid, d) := match u with
     | blank_url ⇒ (true, build_domain "")
     | http_s_url _ d _ ⇒ (false, d)
     end
    in if invalid then rssc else
    StringMap_key_filter (is_vis_resp_cookie l d) rssc.
```

Figure 3.18: `StringMap_key_filter` function

```
Definition StringMap_key_filter {A: Type}
  (f: A → bool) (ckm: StringMap.t A) : StringMap.t A :=
   StringMapProperties.filter (fun _ ck ⇒ f ck ) ckm.
```

Using the standard notion of noninterference, the confidential part (the invisible cookies) are stripped away from the input events. The intuition is that whether or not the input contains confidential data, the observation of the attacker is unchanged only by looking at the public output. The eraser function removes cookies, from the *Set-Cookie* header in network input event, that are not visible

at a label according to the definition in Figure 3.16. The eraser could simply be to replace cookies with constants, however, instead the Coq built-in `filter` function (Figure 3.18) is used to filter the visible cookies which is much easier to deal with in proofs. The function `StringMap_key_filter` recursively applies the predicate `is_vis_resp_cookie` to each value of the cookie map and keeps a cookie only if the predicate is true, hence only visible cookies are kept.

After defining all the necessary pieces of the security policy, all that remains is to put these pieces together in the required form. The binary relation `same_form_ie` defined in Figure 3.19 combines these pieces in one equivalence relation. This function states that two input events are in the same form if they are syntactically identical after invisible cookies are erased and the keys in the corresponding maps (the cookie maps from `String` key to `cookie_flags_value`) are equivalent. User events are not relevant from cookies' security point of view, therefore, the relation `same_form_ie` does not strip any information away from the non-network input events (last case of pattern matching).

After defining the functions and relations on cookies and input events, the two required predicates for input events, `vis_ie` and `sim_ie`, are defined. The visibility predicate for input events, shown in Figure 3.20, is an inductively defined proposition in Coq. There is just one constructor `vis_ie_all` representing the rule for `vis_ie` that all input events are visible at any label.

The second predicate `sim_ie` is similarity relation on input events as shown in Figure 3.21. Similar to `vis_ie` predicate, it also consists of just one rule which asserts that two input events are similar at a label if they are in the same form at that label. The relation `sim_ie` is also reflexive. It could have been written as an inference rule of the relation, however, to make it consistent to `vis_ie` in number of rules, reflexivity is proved as a separate lemma.

Figure 3.19: `same_form_ie` relation

```
Definition same_form_ie l (ieL ieR: input_event) : Prop :=
  match ieL, ieR with
  | network_document_response_event nciL uwiL rsL,
     network_document_response_event nciR uwiR rsR ⇒
     nciL == nciR ∧ uwiL == uwiR ∧
     rsL.(resp_del_cookies) == rsR.(resp_del_cookies) ∧
     rsL.(resp_redirect_uri) == rsR.(resp_redirect_uri) ∧
     rsL.(resp_file) == rsR.(resp_file) ∧
     (∀ k, StringMap.In k (resp_set_cookies rsL) ↔
       StringMap.In k (resp_set_cookies rsR)) ∧
     (erase_invis_cookies l nciL.(net_conn_id_url)
       rsL.(resp_set_cookies) == erase_invis_cookies l
       nciR.(net_conn_id_url) rsR.(resp_set_cookies))
  | network_script_response_event nciL rsL,
     network_script_response_event nciR rsR ⇒
     nciL == nciR ∧
     rsL.(resp_del_cookies) == rsR.(resp_del_cookies) ∧
     rsL.(resp_redirect_uri) == rsR.(resp_redirect_uri) ∧
     rsL.(resp_file) == rsR.(resp_file) ∧
     (∀ k, StringMap.In k (resp_set_cookies rsL) ↔
       StringMap.In k (resp_set_cookies rsR)) ∧
     (erase_invis_cookies l nciL.(net_conn_id_url)
       rsL.(resp_set_cookies) == erase_invis_cookies l
       nciR.(net_conn_id_url) rsR.(resp_set_cookies))
  | network_xhr_response_event nciL rsL,
     network_xhr_response_event nciR rsR ⇒
     nciL == nciR ∧
     rsL.(resp_del_cookies) == rsR.(resp_del_cookies) ∧
     rsL.(resp_redirect_uri) == rsR.(resp_redirect_uri) ∧
     rsL.(resp_file) == rsR.(resp_file) ∧
     (∀ k, StringMap.In k (resp_set_cookies rsL) ↔
       StringMap.In k (resp_set_cookies rsR)) ∧
     (erase_invis_cookies l nciL.(net_conn_id_url)
       rsL.(resp_set_cookies) == erase_invis_cookies l
       nciR.(net_conn_id_url) rsR.(resp_set_cookies))
  | _, _ ⇒ ieL == ieR
```

Figure 3.20: `vis_ie` relation

```
Inductive vis_ie (l: label): input_event → Prop :=
  | vis_ie_all: ∀ ie, vis_ie l ie.
```

Figure 3.21: `sim_ie` relation

```
Inductive sim_ie (l: label): input_event → input_event → Prop :=
  | sim_ie_same_form: ∀ ieL ieR,
     same_form_ie l ieL ieR → sim_ie l ieL ieR.
```

3.3.2 The Policy for Output Events

The policy for output events is determined by the way these events impact the outside world. Similar to the input events, there are two predicates, `vis_oe` and `sim_ie`, which form the policy on the output events.

Figure 3.22: `vis_oe` relation

```
Inductive vis_oe (l: label): output_event → Prop :=
  | vis_oe_ui_window_opened_event:
     top_label <= l → vis_oe l ui_window_opened_event
  | vis_oe_ui_window_closed_event: ∀ uwi,
     top_label <= l → vis_oe l (ui_window_closed_event uwi)
  | vis_oe_ui_page_loaded_event: ∀ uwi u rdto,
     top_label <= l → vis_oe l (ui_page_loaded_event uwi u rdto)
  | vis_oe_ui_page_updated_event: ∀ uwi rdto,
     top_label <= l → vis_oe l (ui_page_updated_event uwi rdto)
  | vis_oe_ui_error_event: ∀ z,
     top_label <= l → vis_oe l (ui_error_event z)
  | vis_oe_network_document_request_event: ∀ uwi nci rq,
     url_label nci.(net_conn_id_url) <= l ∨ net_label <= l →
     vis_oe l (network_document_request_event uwi nci rq)
  | vis_oe_network_script_request_event: ∀ nci rq,
     url_label nci.(net_conn_id_url) <= l ∨ net_label <= l →
     vis_oe l (network_script_request_event nci rq)
  | vis_oe_network_xhr_request_event: ∀ nci rq,
     url_label nci.(net_conn_id_url) <= l ∨ net_label <= l →
     vis_oe l (network_xhr_request_event nci rq).
```

The first part of the policy is defined as predicate `vis_oe` given in Figure 3.22. It is an inductive type with inference rules one for each output event. To put security constraints on the way output events should impact the outside world, significant changes are added to the last three constructors of the relation corre-

sponding to network events. Both the encrypted and the unencrypted network traffic is visible (at least the occurrence of the event) to an attacker at or above `net_label`. Additionally, the outputs to the user are visible at `top_label`.

Figure 3.23: `sim_oe` relation

```
1 | Inductive sim_oe (l: label): output_event → output_event → Prop :=
2 |   | sim_oe_same_form: ∀ oeL oeR,
3 |     same_form_oe l oeL oeR → sim_oe l oeL oeR.
```

The second part of the policy is a binary relation `sim_oe` as shown in Figure 3.23. This relation consists of just one rule which asserts that two output events are similar if they are in the same form. The relation `same_form_oe` (Figure 3.24) defined over output events is large but simple to understand.

Two unencrypted network requests or two user output events are in the same form if they are identical. For encrypted requests, they are in the same form if they differ only in the confidential cookies. This later condition is enforced through the function `erase_cookies` as shown in the Figure 3.25. It erases cookies from requests over HTTPS such that for any label `l`, cookies are invisible at `l` while the unencrypted requests are unchanged. As every `Secure` cookie is also marked as `HttpOnly`, a request over HTTPS to domain `d` will only include cookies with both flags set and hence will get the label (`https_label d`). Therefore, the request URL and cookie key-value pairs (without flags) are sufficient to pass as the argument to the function `erase_cookies`, which returns only visible cookies.

The intuition of the function `erase_cookies` is 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 (at or above `https_label`), who is able to decrypt the message. A randomized encryption scheme is assumed, whereby encrypting the same request twice always

Figure 3.24: `same_form_oe` relation

```
Definition same_form_oe (l: label) (oeL oeR: output_event) : Prop :=
  match oeL, oeR with
  | network_document_request_event uwiL nciL reqL,
     network_document_request_event uwiR nciR reqR ⇒
     uwiL == uwiR ∧ nciL == nciR ∧
     reqL.(req_req_uri) == reqR.(req_req_uri) ∧
     reqL.(req_referer) == reqR.(req_referer) ∧
     reqL.(req_body) == reqR.(req_body) ∧
     erase_cookies l nciL.(net_conn_id_url) reqL.(req_cookies)
      == erase_cookies l nciR.(net_conn_id_url) reqR.(req_cookies)
  | network_script_request_event nciL reqL,
     network_script_request_event nciR reqR ⇒
     nciL == nciR ∧
     reqL.(req_req_uri) == reqR.(req_req_uri) ∧
     reqL.(req_referer) == reqR.(req_referer) ∧
     reqL.(req_body) == reqR.(req_body) ∧
     erase_cookies l nciL.(net_conn_id_url) reqL.(req_cookies)
      == erase_cookies l nciR.(net_conn_id_url) reqR.(req_cookies)
  | network_xhr_request_event nciL reqL,
     network_xhr_request_event nciR reqR ⇒
     nciL == nciR ∧
     reqL.(req_req_uri) == reqR.(req_req_uri) ∧
     reqL.(req_referer) == reqR.(req_referer) ∧
     reqL.(req_body) == reqR.(req_body) ∧
     erase_cookies l nciL.(net_conn_id_url) reqL.(req_cookies)
      == erase_cookies l nciR.(net_conn_id_url) reqR.(req_cookies)
  | _, _ ⇒ oeL == oeR
```

Figure 3.25: `erase_cookies` function

```
Definition erase_cookies (l: label) (u: url)
  (cks: StringMap.t String.t) : StringMap.t String.t :=
    let (prot, d) := match u with
     | http_s_url prot d ru ⇒ (prot, d)
     | blank_url ⇒ (about_protocol, build_domain "")
    end in
    if (prot =?= https_protocol && !(https_label d <?= l )
    then StringMap.empty else cks.
```

produces two different cipher texts. Notice that similar output events must be sent to the same URL, i.e., it is assumed that the attacker is able to observe the recipient of any visible network event.

3.3.3 Cross-domain Requests

In the modern web framework, most of the web applications are relying on data and content (e.g., images, CSS and JavaScript code) from third parties. When the application is being loaded, the web browser *implicitly* requests resources (for remote scripts and images) from servers that might be different than the *origin* of the application. JavaScript is used by web developers to enhance interactivity of their sites by *loading* the work to user's browser and to improve their sites' responsiveness and user friendliness. Most of the websites (more than 85%) include remote scripts offering services such as Web Analytics, Market Research, User Tracking, Dynamic Ads and Social Networking, from origins different than the origins of the applications [89, 40]. Although, cross-domain requests are necessary for proper web applications functionality, unsafe third party script inclusion lead to a number of security vulnerabilities [89]. Web application may also include HTML links, pointing to origins different than the application origin, which may later be clicked by the user. Both explicit and implicit cross-origin requests causes the popular class of attacks CSRF [40, 100].

Cross-domain requests are, therefore, some time stopped to prevent attacks such as CSRF [102] or to simply reasoning about other security properties in the web browsers [17]. To prevent CSRF attacks while allowing cross-domain requests, the complex and challenging issue of integrity of web sessions must be solved. A general solution would be to allow cross-domain request but strip away the cookies from such requests. This would, however, make many web applications, such as single sign on and e-payment, incompatible. A solution to this problem would

be to allow cookies with legitimate cross-domain requests and strip away from malicious cross-domain requests. The challenge, however, is to find when a request is malicious. The solution can be based on heuristic to determine if a request is legitimate (expected) or malicious [100, 40].

Unlike Featherweight Firefox [17], cross-domain remote script requests are allowed in EFF. Web applications can load JavaScript codes remotely from cross origins as permitted by the same-origin policy, however, can only load contents from the same origin using AJAX requests. This behaviour does not add any additional confidentiality threats to cookies other than existing threats (e.g., XSS) to session cookies, which are protected by the security policy (Section 3.3) and the EFF model enforcing this policy is proved secure according to the policy in next chapter[2].

Allowing cross-domain requests, however, can lead to CSRF attacks as mentioned before. The model does not include remote image inclusion, however, it allows to include cross-domain remote scripts. This would be sufficient to reason about the security against CSRF attacks caused by cross-domain requests, however, such attacks are considered as session integrity problem. The flexibility regarding cross-domain requests would require consideration of integrity which would in turn have over complicated the reasoning about other security features in EFF. Capturing attacks on session integrity in formal setting is quite complicated and is, therefore, addressed separately in Chapters 7 and 8 using access control and information-flow control models.

[2]Third-party script inclusions and HTTP requests initiated by different origins can, however, increase the surface of these attacks but the proposed defences prevent them.

4

Web Session Confidentiality: The Browser State

In Chapter 3, the input and output events of the model and the information-flow security policy over these events were defined. Unlike the abstract reactive systems model [19], the EFF model include a number of stores and small-step operational semantics for script execution to enforce fine-grained information-flow security policies. To achieve this, the state of EFF include stores for cookies, windows, pages, open network connections and so on. The detail of these stores that comprise the state of the browser, the main security theorem and its mechanised proof are discussed in this chapter.

4.1 Browser State

The web browser is modelled as a (multi-step) reactive system consisting of two states[1]: the `waiting_state` and `runnin_state`, where both states contain the same basic information represented the same way. The basic information that makes the core of the browser state consist of twelve fields as shown in Figure 4.1. Each state include pages, windows, document nodes, activation records, meta

[1]The terms *Consumer* and *Producer* are used for the states in standard reactive systems. The EFF is modelled as a multi-step reactive system with states and the way they release their outputs are different than the reactive system states (see section 2.3).

data, cookies and open network connections. There are five stores for windows, pages, nodes, node forests and activation records that are indexed by dynamically allocated references.

The references pointing to different kinds of data in the stores are modelled in Coq as `nat`, however, to ease its use in finite maps using functors in Coq, they are placed in module `Ref` with a field of type `nat`. In the Coq code, a reference is referred to as `Ref.t` (where `Ref` is a module and `t` (type) is a member set to type `nat` in this case) which is transparently equal to `nat`. To leverage Coq type classes to facilitate further operations on references, the finite map from references to a parametric type are wrapped in record types. There are four such finite parametric maps (`Ref.t A`) where the type `A` is instantiated with types for windows, pages, nodes and activation records respectively.

In the browser, script expressions may evaluate to windows and document nodes which may persist indefinitely even after windows and nodes are rearranged. To identify particular windows and nodes such that their identities remain stable throughout browsing session, fresh references to windows and nodes are used. The activation records can be generated dynamically and are associated with fresh references. Similarly, the data for pages, which represents transient data associated with windows, are placed in store and assigned fresh generated reference. The last two fields of type `browser`, `browser_next_w_p_ref` and `browser_next_n_a_ref`, represent the fresh reference allocation to windows, pages, nodes and activation records.

One reference allocator would be sufficient, however, having more than one allocator simplifies the proof. There could be a separate reference allocator for each of these types, however, it would have duplicated the code unnecessarily. Having two allocators seems a reasonable choice and hence are kept as in Featherweight Firefox Coq model. The rest of the fields of type `browser` are discussed in more

detail as the following.

Figure 4.1: `browser` data type

```
Record browser: Type :=
  build_browser {
   browser_open_wins: list win_ref;
   browser_wins: win_ref_map win;
   browser_pages: page_ref_map page;
   browser_nodes: node_ref_map (node_ref * node);
   browser_node_forest: node_ref_map node_tree;
   browser_acts: act_ref_map act;
   browser_cookies: CookieMap.t cookie_flags_value;
   browser_doc_conns: UrlMap.t (list doc_conn);
   browser_scr_conns: UrlMap.t (list scr_conn);
   browser_xhr_conns: UrlMap.t (list xhr_conn);
   browser_next_w_p_ref: Ref.t;
   browser_next_n_a_ref: Ref.t
  }.
```

4.1.1 Windows

The first field `browser_open_wins` is the list of references to windows that are visible to the user. When a new window is opened, it is placed at the head of this list indexed by the identifier `user_win_id`. As mentioned above, a window reference is a number `nat` but wrapped in the record type `win_ref`.

Figure 4.2: `win` data type

```
Record win: Type :=
  build_win {
   win_name: option String.t;
   win_opener: option win_ref;
   win_page: page_ref
  }.
```

The second field `browser_wins` of type `browser` is the store for windows: a mapping from window references to type `win` (Figure 4.2). This models only a small

subset of DOM window object properties. The fields `win_name` and `win_opener` correspond to the window object properties: *name* of the window and *opener* of the current window. The last field `win_page` represents a page reference to the `page` currently in the window. After the addition of the *Referer* header, a way was needed to track the opener of a window when user opens a link in a new window. For this purpose, the parameter `user_win_id` was added to the user input event `user_load_in_new_window_event` (see Section 3.1). The field `user_win_id` in this input event points to the *opener* (previous) window, containing a page where the user clicked on a link. The `url` of the page (`win_page`) in the *opener* window, which is pointed to by the `user_win_id`, is used as the referrer of the current page document. When windows are closed, their references are removed from the store, hence, the opener window and the page in that window can not be retrieved if the window is closed.

Figure 4.3: `page` data type

```
1 Record page: Type :=
2   build_page {
3    page_label: label;
4    page_location: url;
5    page_document: option node_ref;
6    page_environment: act_ref;
7    page_expr_queue: list queued_expr
8   }.
```

4.1.2 Pages

The third field `browser_pages` of type `browser` is the store for pages. It is a mapping from page reference to type `page` (Figure 4.3). The field `page_label` is the label associated with the page and the field `page_location` represents the source URL of the page currently being displayed. Using the function `url_label`

(Figure 3.14), a security label could be associated with the page based on its source URL, however, the `page_label` differs from the source URL label when a window is opened and a document is not yet loaded. A window without page still holds blank page with URL *about:blank*, which is assigned the `top_label`. This is later changed to the label of HTTP(S) URL when the page is loaded. The `page_document` represents the optional root node of the document tree of the page in the window. Initially, when the window holds a blank page, the root node of the document tree is set to `None`. The global environment of the scripts loaded into a page is stored as an activation record `act_ref` in the field `page_environment`.

The scripts loaded for a page are stored in the field `page_expr_queue`. This is a list of scripts, including both in-line and remote scripts, corresponding to the *<script>* and *<script src>* tags in the document tree of the page. Remote scripts must be retrieved through HTTP(S) requests. Scripts are executed, in the order they appear in the document, soon after the page is loaded. A page in the window is removed from the store when a different page is loaded in its container window or when the window is closed.

4.1.3 Document Nodes

The type `browser` contains two different node stores: the `browser_nodes` and `browser_node_forest`. The first node store is a map from node reference to a pair of `node_ref` and `node` and is used for data associated with the node. The second store keeps track of the parent-child relationship of nodes and is a map from node reference to `node_tree`. The `node_ref` is the reference to the root node of the tree where the node is found. The data types `node_tree` and `node` are shown in Figure 4.4 and 4.5.

The DOM views documents as tree structures called *node tree* and is modelled as type `node_tree` with each node referred to by `node_ref`. Each document `node`

Figure 4.4: `node_tree` data type

```
Inductive node_tree: Type :=
  | node_tree_branch:
     node_ref → list node_tree → node_tree.
```

Figure 4.5: `node` data type

```
Inductive node: Type :=
  | inl_script_node:
     option elt_id → script → bool → node
  | rem_script_node:
     option elt_id → url → bool → node
  | textbox_node:
     option elt_id → String.t → list expr → node
  | div_node: option elt_id → node.
```

represents either in-line script, remote script, a text box or an HTML *div* element.

4.1.4 Activation Records

The field `browser_acts` is the store for activation records: a map `act_ref_map` from activation references to type `act`. The definition of activation record type `act` is shown in Figure 4.6. An activation record represents the execution environment of a function and is created each time when the function is applied.

Figure 4.6: `act` data type

```
Record act: Type :=
  build_act {
   act_parent: option act_ref;
   act_depth: nat;
   act_vars: list (var * expr)
  }.
```

The field `act_vars` is a mapping from the values `expr` of the function parameter

to each of the function's local variables `var`. A script may read or update a variable not in the associated activation record. In such cases, the browser will attempt variable access from the parent record stored in the field `act_parent`. The set of activation records in the browser forms a forest of trees structure. The field `act_depth` indicates the number of ancestors that the activation record `act` has in the tree structure. The global environment of pages consists of activation records that forms the roots of these trees. Unlike windows and pages, activation records are not garbage-collected and hence they are accessible even after their associated pages are removed from the store.

4.1.5 Cookies

Figure 4.7: `cookie_id` data type

```
1 Record cookie_id: Type :=
2   build_cookie_id {
3    cookie_id_domain: domain;
4    cookie_id_path: path;
5    cookie_id_key: String.t
6   }.
```

The field `browser_cookies` represents the store of cookies in the browser. It is a finite map from the type `cookie_id` to `cookie_flags_value`. The definition of type `cookie_id` is shown in Figure 4.7. In the browser, a cookie store is a mapping (key-value pairs) from string key to string value. However, the browser must keep track of cookie mappings for each file path (represented by the field `cookie_id_path`) registered by each domain (`cookie_id_domain`). The field `cookie_id_key` of type `String` is the cookie name (key). In Featherweight Firefox model, the cookie store is a mapping from the `cookie_id` to `String` value. In EFF, instead, the value of type `String` is wrapped in a record type `cookie_flags_value`

(Figure 3.6) with two additional fields for flags `Secure` and `HttpOnly`. These flags are used to define and enforce the information-flow policy (Section 3.3) to protect attacks on web session confidentiality.

4.1.6 Network Connections

The fields `browser_doc_conns`, `browser_scr_conns` and `browser_xhr_conns` represent the three different network connection stores corresponding to document, script and *XMLHttpRequest* (AJAX) network requests, respectively. Each store is a mapping form the URL to the list of opened network connections to that URL.

A security label is associated with each network connection based on the protocol used in the corresponding network request, therefore, in addition to the domain, the protocol is also required. Both, the domain and protocol, could be added as separate fields to these record types, however, instead the type `url` is used as it contains both. There is a different store for each kind of network connection in the browser, each corresponds to the type of the network request (document, script or AJAX). Similarly, there is a separate handler in the model for each kind of network input. The URL of the `net_conn_id` in each network input response is used to access a list of network connections to that URL. The numerical part of the `net_conn_id` is then used to uniquely identify a specific network connection in that list. This is useful to match-up the response to the corresponding request. The browser, for example, will not accept a document input from a domain in response to a script request to that domain.

The network connections for accessing top level HTTP(S) documents are stored in the field `browser_doc_conns`. It is a mapping from the URL to the list of document connections `doc_conn`. The definition of type `doc_conn` is shown in Figure 4.8. The first field of this type is the `req_uri` which is needed to update the cookies in the browser through the *Set-Cookie* header in the network response.

Figure 4.8: `doc_conn` data type

```
Record doc_conn: Type :=
  build_doc_conn {
   doc_conn_req_uri: req_uri;
   doc_conn_page: page_ref
  }.
```

The second field is `doc_conn_page` which stores the reference to the page on which the document request was originated. This field is added to populate the *Referer* header of a redirected document request[2]. It is either the page where an HTTP(S) link was clicked or the previous (parent) page in case a link is opened in a new tab. If a new window (with blank page and blank URL) is navigated to a URL, there is no parent page and hence `blank_url` is used as the referrer. Note that, the `blank_url` as referrer is interpreted as disabled or no referrer header. The window into which the document is to be loaded is provided as parameter in the document response input (Section 3.1).

Figure 4.9: `scr_conn` data type

```
Record scr_conn: Type :=
  build_scr_conn {
   scr_conn_req_uri: req_uri;
   scr_conn_page: page_ref;
   scr_conn_node: node_ref
  }.
```

The document tree may contain remote script nodes which correspond to the *<script src>* DOM tag. Remote scripts are loaded immediately through HTTP(S) script requests when the page is being rendered. Network connections corresponding to the script requests are stored in the store `browser_scr_conns`. It is a

[2]Note that, keeping the original referrer is a common practice used by most of the browsers and is used in EFF, but it is not recommended by the corresponding RFCs.

mapping from URL to the list of script connections `scr_conn` (Figure 4.9). The first field `req_uri` is the path of the script file. The second field `scr_conn_page` stores the page on which the script should be run when it is received. The location of this page serves as the referrer when the script requests is redirected. The field `scr_conn_node` stores the node of the document tree that triggered the script request.

Figure 4.10: `xhr_conn` data type

```
1 Record xhr_conn: Type :=
2   build_xhr_conn {
3    xhr_conn_req_uri: req_uri;
4    xhr_conn_page: page_ref;
5    xhr_conn_handler: expr
6   }.
```

The network connections corresponding to the AJAX requests are stored in the store `browser_xhr_conns`: a map from URL to the list of AJAX connections `xhr_conn` (Figure 4.10) to that URL. Similar to `doc_conn` and `scr_conn`, the first two fields represent the path file and the source page of the request. The field `xhr_conn_handler` stores the expression (handler) to handle the AJAX response on that page if it still exists.

4.1.7 Waiting and Running States

Similar to the *Consumer* and *Producer* states of the standard reactive system [19], EFF accepts inputs in the *waiting* state and produces a (possibly empty) sequence of outputs at the *producer* states. The waiting state of the browser consists of just the core browser data and the running state include one additional component: a list of tasks. In the Featherweight Firefox model [17], the *security mode* is set to the security level of input event needed to securely execute the scripts in the task

list. In EFF setting, all the input events are visible at any label, therefore, the security mode is not associated with input event and hence is not included in the running states. The definitions of `waiting_state`, `running_state` and `state` are given in the Figure 4.11. The infix notations `*` and `+` are used to define *product* and *sumbool* types, respectively.

Figure 4.11: `waiting` and `running` data types

```
1 | Definition waiting_state := browser.
2 | Definition running_state := browser * list task.
3 | Definition state := waiting_state + running_state.
```

The browser in a waiting state steps to a running state with a list of tasks (scripts) when it receives an input event (e.g., document response). The definition of type `task` is shown in Figure 4.12. A task consists of a field for the script `expr` that is to be executed in running state and a reference to the window containing the page where the scripts are to be executed. All the scripts in the list of tasks must be executed before the browser can return to the waiting state.

Figure 4.12: `task` data type

```
1 | Record task: Type :=
2 |   build_task {
3 |    task_win: win_ref;
4 |    task_expr: expr
5 |   }.
```

The scripts in task list are evaluating using a standard small-step semantics with the top-level multi-step reactive system progressing to a new `running_state` after each step of the evaluation. The small-step semantics for the script evaluation is defined using Coq function definition. Unlike inductive definition, deterministic[3]

[3]Recall that, only deterministic reactive systems are considered in this dissertation.

systems are easy to reason about when defined as functions in Coq. The `script` data type is the surface language of scripts defined in the Section 3.1. However, script evaluation needs to be defined in terms of internal language with syntax defined by the type `expr` which contains some expression that are not allowed in the surface language. The definition of type `expr` is given in Appendix A.1.

The `expr` data type has a constructor corresponding to each constructor in type `script` plus five additional constructors. The first one is `error_expr` which is a string representing a run-time error during expressions evaluation. This error message is later propagated to the user as `ui_error_event` output event. The window and node references are treated as values in the language and the constructors, `win_expr` and `node_expr`, are used to convert window and node references to expressions. The fourth expression is `scoped_expr` which should be evaluated in a specific context. The last additional expression is `closure_expr` which is resulted from the evaluation of `function_expr`. It, in addition to the data of the `function_expr`, includes a copy of the `context` in which the `function_expr` was evaluated. The application of closures evaluate to a `scoped_expr` which inherits its context from the closure.

The expression `get_cookies_expr` corresponds to `get_cookies_script` (Section 3.1). The addition of this script is very important and hence is discussed in more detail. As discussed in Section 3.1, the main objective of the expression `get_cookies_expr` is to access the non-`HttpOnly` cookies, registered by the owner of the document, from the store. Using this expression and the `HttpOnly` flag, interesting security policies were defined, enforced in EFF and are proved secure.

When the expression `get_cookies_expr` is evaluated, it internally executes a function `get_site_cookies_httponly` (Figure 4.13) that accesses all the non-`HttpOnly` cookies. The function gets the source URL of the page where the script is residing and the cookies store as the input and returns all the non-`HttpOnly` cook-

Figure 4.13: `get_site_cookies_httponly` function

```
1  Definition get_site_cookies_httponly (u: url)
2    (ckm: CookieMap.t cookie_flags_value) : StringMap.t String.t :=
3     match u with
4     | blank_url ⇒ StringMap.empty
5     | http_s_url prot d ru ⇒
6        CookieMap.fold
7         (fun cki ck zm ⇒ if cki.(cookie_id_domain) =?= d &&
8           cki.(cookie_id_path) =?= ru.(req_uri_path) &&
9           !ck.(cookie_flags_value_httponly)
10          then StringMap.add cki.(cookie_id_key)
11          ck.(cookie_flags_value_value) zm else zm)
12       ckm
13       StringMap.empty
14    end.
```

ies, as a mapping (`StringMap.t String.t`), registered for that URL. It is using the Coq function `fold` which in turn is applying another function to each key-value pair in the cookie store (a mapping from `cookie_id` to `cookie_flags_value`). The body of the later function is matching the `domain` and `path` of the URL with the `domain` and `path` of each cookie in the store. If a match is found and the `HttpOnly` flag is not set (in Coq code, type `bool` is used for flags instead of `string`), the cookie is added to the accumulator. As a result, the function returns all non-`HttpOnly` cookies registered for that `domain` as map from cookie name to value. When the expression `get_cookies_expr` evaluates, all the returned cookies are stored as a string expression (see below).

These cookies could have been stored as a mapping (`StringMap.t String.t`) as returned by the `get_site_cookies_httponly`, however, an extra `expr` constructor would be needed to make expression from type (`StringMap.t String.t`). Moreover, it is a tedious job to define and prove Coq type class definition and relation (*setoid* and *equivalence*) for the whole type `expr` with twenty-three constructors. To prove that, *similar* browser states remain in sync if they both get

similar inputs, Coq type classes are used in the proofs to ease *rewriting*. A *setoid* consists (is record) of a type A together with a binary relation (equiv: relation A) on A and proof (proposition) that this relation is an *equivalence* relation. The existing *equivalence* relation over type `expr` is defined in terms of *equality* which is easy to deal with in Coq. The simple equality relation seems quite restrictive, however, it behaves similar to the flexible equivalence relation *equiv* (see Coq code) for simple types such as `nat`. On the other hand, the equality is overly restrictive for maps while in this case flexible relation was needed. For example, two states with exactly the same data structures but *equivalent* cookie stores behave exactly the same in real browsers. Converting the string map to a string of key-value pairs makes the proof quite easy as *equality* can be used as the relation required to define *setoid* for the type `expr`.

This conversion requires some effort, though, to prove that two *equivalent* (cookie) string maps result *equal* strings. To achieve this, the string map, returned by the `get_site_cookies_httponly`, is converted to a list of strings (key-value pairs). The list of strings (key-value pairs) are sorted by lexicographic order using keys. This sorting is not required for functionality reasons but for the proofs. The sorted list of cookies is then converted to a string of semi-colon separated cookies, similar to the result of command *document.cookie* at the console. Finally, it is proved that for any two *equivalent* cookie maps, the corresponding sorted strings of cookies are exactly the same.

4.2 Proof of Session Confidentiality

The input and output events were extended with the additional facilities to support communication over HTTPS and cookie flags (Section 3.1 and 3.2) to define meaningful information-flow security policies and the internal data structures of

the browser model were updated with additional features (Section 4.1) to enforce these security policies. In Section 3.3, the security policy protecting session cookies was defined in terms of noninterference. It appears that, the model should now behave *securely* as the standard browser according to the protection mechanisms based on flags `Secure` and `HttpOnly`, however, its security guarantees according to the security policy have not been proved yet. To give mathematical guarantees of the security, a proof, that the (model of) the browser is secure according to this security policy, is needed. This is stated in terms of the main theorem as the following.

Theorem 1. *The Extended Featherweight Firefox (EFF) browser model is noninterferent according to the security policy defined in Section 3.3.*

Proof. The machine-checked proof of this theorem is given in the accompanying Coq code. □

This theorem is proved using Bohannon's ID-bisimulation proof technique [19]. Using Coq module system as used by Bohannon [17], the proof of this theorem would be to build a module that conforms to a signature `SecureReactiveSystem` with its components `ReactiveSystem` and `Policy` instantiated with the EFF and the security policy defined in Section 3.3. As web browser is formalized as a multi-step reactive system, this is achieved by first building a module conforming to the signature of `SecureMultiStepReactiveSystem` with its components `ReactiveSystem` and `Policy` instantiated with the EFF and the security policy. To build such a module, similar to `SecureReactiveSystem`, first the *lockstep*[4] unwinding relation [19] `sim` is defined and then the five theorems, corresponding to the five axioms in Bohannon's ID-bisimulation definition, are proved. This

[4] A *lockstep* version of the standard unwinding relation is used where both states always take step together.

would be the proof of security of multi-step reactive system. The unwinding relation `sim` is defined as the inductive type `sim_wrq` (Figure 4.14), which consists of only two constructors each comparing two states of the same type (two waiting or two running states). The relation requires that if one state take a step, the other must also take a step, hence *lockstep*. After the security of multi-step reactive system is proved, the `SecureMultiStepReactiveSystem` is transformed to a `SecureReactiveSystem` which would be the proof of security of standard reactive system.

Figure 4.14: `sim_wrq` relation

```
1  Inductive sim_wrq (l: label): state → state → Prop :=
2    | sim_wrq_wq_wq: ∀ bL bR,
3       sim_b l bL bR →
4       sim_wrq l (inl bL) (inl bR)
5    | sim_wrq_rq_rq: ∀ bL bR tlL tlR,
6       sim_b l bL bR →
7       bL.(browser_next_w_p_ref) ## tlL →
8       bR.(browser_next_w_p_ref) ## tlR →
9       bL.(browser_next_n_a_ref) # tlL →
10      bR.(browser_next_n_a_ref) # tlR →
11      tlL == tlR →
12      sim_wrq l (inr (bL, tlL)) (inr (bR, tlR)).
```

The result proved in the Theorem 1 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. The actual adoption of these flags in existing systems needs to be determined and, if the deployment in existing systems is not satisfactory, countermeasures should be taken to secure modern web browsers. This is formalized, implemented and experimentally analysed in the next chapters.

5

Web Session Confidentiality: Browser-Side Enforcement

The Coq proofs provide certified guarantee of the effectiveness of the native protection mechanisms, based on `Secure` and `HttpOnly` flags, used in modern web browsers. In the formal browser model, the assumption is that session cookies have been *properly* flagged by the web developers. That is, to protect session cookies, servers set them with both `Secure` and `HttpOnly` flags if registered over HTTPS and only `HttpOnly` if registered over HTTP, however, this is not happening in practice. The experiments shows that most of the existing web applications do not deploy these protection mechanisms properly and hence fail to protect session cookies.

To fill this gap between the security that these mechanisms can actually provide and their actual adoption in the web applications in practice, a browser extension CookiExt is proposed and developed for the Chrome web browser. In this chapter, the experimental results motivating towards client-side protection, are given (Section 5.2) and then the design and implementation of CookiExt (Section 5.3) is discussed. At the end of the chapter, CookiExt is evaluated and its effectiveness at protecting session cookies is discussed (Section 5.4).

5.1 Session Cookies Protection in Existing Systems

To assess the actual adoption of the security flags in existing systems, a survey of the top 1000 websites of Alexa was conducted. In the survey, all of the cookies, registered using HTTP headers, were collected from these sites and then a heuristic was applied to isolate session cookies. The adopted heuristic simply marks a cookie as a session cookie if it satisfies either of the following two 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[1] is below 0.04.

The heuristic used is consistent with previous proposals [90] 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 the heuristic used in CookiExt, however, just after the work was published [21], Calzavara et al. [25] analysed and compared it with few other solutions and found it with satisfactory results. Moreover, as will be discussed, however, there are other strong evidences that the survey is reliable enough.

The heuristic is evaluated using two experiments. The first experiment provides an estimate of the false positives identified by the heuristic. First, the percentage

[1] This is a statistical measure which can be effectively employed to understand how likely a given text was randomly generated [55].

of cookies identified by the heuristic is computed which are *weak* session identifiers, i.e., fail to satisfy condition 2 above and thus are probably not used for authentication purposes. It turns out that only 8.75% contain weak session identifiers. Interestingly, a manual inspection shows that most of these failures are concentrated in very few websites (e.g., Amazon), which adopt the string 'sess' in an unanticipated way. If only Amazon is excluded, the false positive rate lowers to 6.01%, which is considered acceptable for the investigation.

The second experiment aims at estimating the false negative rate and counts 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 likely contain session information. As it turns out, only 3.21% cookies ignored by the heuristic have at least one security flag set. Again a manual inspection reveals that this is a very localized behaviour, typically adopted by some high-security websites (e.g., PayPal), which seem to enforce very strong protection for the large majority of their cookies, irrespective of the nature of their content. Further experimental evaluation of the heuristic on top 100 Alexa websites, where personal accounts were created, is given in the Section 5.4.

5.2 The Need for Client-side Defence

Table 5.2 provides some statistics which highlight that the large majority (71.35%) of the session cookies, identified in the experiment, has no flag set. Although, 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 evidences are provided by other experiments (Section 5.4).

The `HttpOnly` flag appears to be adopted much more widely than `Secure` (a cookie set with a particular flag is represented with $\surd$ in the table). The

Table 5.1: Statistics about cookie flags

`HttpOnly`	`Secure`	Cookies	Percentage
√	√	32	2.81%
√	×	284	24.96%
×	√	10	0.88%
×	×	812	71.35%

conjectured reasons are: 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; secondly, `Secure` cookies presuppose an HTTPS implementation, which is not available for all websites.

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 [90, 106]. This idea is pushed 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, a simple experiment was conducted to estimate the extent of the actual HTTPS deployment. The experiment showed that 192 out of the 443 (43.34%) websites registering at least one session cookie support HTTPS transparently, i.e., they can be successfully accessed simply by replacing `http` with `https` in their URL[2]. Notice that the real percentage of websites supporting HTTPS can be possibly higher, since encrypted variants of existing websites are sometimes hosted on a different domain and no redirect to this domain is performed when the homepage is accessed over HTTPS.

Moreover, it was observed that only 8.33% websites set the `Secure` flag for at least one session cookie. Remarkably, it turns out that 73.44% of these websites contain at least one HTTP link to the same domain hard-coded in their homepage,

[2]In this count, a number of websites which automatically redirect HTTPS connections over HTTP were excluded.

hence session cookies which are not marked `Secure` are at risk of being disclosed to a network attacker when navigating these websites. These data suggest that the `Secure` flag for session cookies could (and should) be more largely deployed in practice, which motivates towards the client-side defence proposal, discussed in the next section.

5.3 CookiExt: Enforcing Session Confidentiality

The implementation of the required protection mechanisms should be designed so as to minimize their impact on the user experience: this is a difficult task, which requires careful design based on the search of the best possible trade-off between security and usability. Moreover, the design should lend itself to an implementation as a browser extension, to ease its deployment. When that is not possible, it is important to fine-tune the proposed security mechanisms so that they are still consistent with the theoretical model.

CookiExt is an extension for Google Chrome aimed at enforcing robust client-side protection for session cookies. The reason to choose Chrome for the extension development is that 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.

In the initial CookiExt proposal [21], it was designed as the following: when the browser received an HTTP(S) response, CookiExt inspected its headers trying to identify the session cookies based on the heuristic discussed earlier. If a session cookie was found, CookiExt behaved as follows:

1. If the response was sent over HTTPS, all the identified session cookies were marked `Secure` and `HttpOnly`.

2. If the response was sent over HTTP, all the identified session cookies were

erased from the HTTP headers.

In both cases, all subsequent requests to the website were automatically redirected over HTTPS, even for sites that were not supporting HTTPS at all. As the extension was white-listing individual links, it had to redirect each request at least once to check if HTTPS was supported. Each time, when an HTTP request was redirected to HTTPS, a small timeout delay was set. If the website, that was not supporting HTTPS, returned a network error or the timeout expired, it was added to a white list. The approach of setting a timeout before sending an HTTP request was not very robust in practice and CookiExt often adopted a fallback to HTTP when it was not actually required.

Websites such as `www.bbcurdu.com` redirect HTTPS requests over HTTP automatically if HTTPS is not supported. Some of these websites, responded with network error `ERR_TOO_MANY_REDIRECTS` if the redirected requests exceeded a certain threshold number[3]. These redirected requests over HTTPS for non-supporting sites unnecessarily add to network traffic: for some websites, the browser redirected more than hundred HTTP requests (e.g., for sub-resources such as images and scripts) for a single web page, which were redirected back to HTTP by the server. To address these issues, along with many others, the design of the CookiExt was further improved with a number of necessary changes and extensive experiments were carried to analyse many interesting features of the web.

5.3.1 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

[3]The number varies from site to site but normally it ranges between 50-70

based on the heuristic discussed earlier. If a session cookie is found, CookiExt behaves as follows:

1. If the response was sent over HTTP, all the identified session cookies are marked as `HttpOnly`.

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

This simple picture of CookiExt, however, is significantly complicated by a number of issues which arise in practice and must be addressed to devise a usable implementation. The extension maintains a number of stores for the record, most notable of them are the lists of *auto-redirect* domains, *secured* cookies and *white-listed* URLs. When login operation is successful[4] over HTTPS, the base domain of the request is added to the auto-redirect list. There are two possible scenarios: 1) a site (e.g., `linkedin.com`) may deliver the login page over HTTPS, but as soon the login is successful, it immediately redirects the browser over HTTP, or 2) it keeps delivering all the subsequent pages (e.g., `facebook.com`) or at least the page following login operation over HTTPS (e.g., `yahoo.com`). In both cases, the domain of the request is added to the *auto-redirect* list so that all of the subsequent requests to the same domain (or its sub or upper domains) may be redirected over HTTPS. The reason behind this rule is cookies: a cookie set by a domain is included with the requests to any of its sub-domain and vice versa.

As the main goal of the CookiExt is to flag session cookies, if they are not already flagged, to secure them, however, due to usability reasons, this rule can not be applied all the time. In practice, some times (see below) cookie flags need to be *reverted* back to their original values, therefore, the CookiExt keeps track of

[4]A simple heuristic is used to check if a login operation is successful over HTTPS.

the cookie flags that are altered. The list of *secured* cookies is used exactly for this purpose. The list *white-listed* stores URLs with minimum (or no) constraints applied by CookiExt on subsequent requests to any URL in this list.

The design of CookiExt and the issues that arises in practice are discussed in more detail.

5.3.2 Flagging Session Cookies

As mentioned above, whenever a session cookie is identified in HTTP(S) response, the cookie headers are modified before the cookies are stored in Chrome local store. The CookiExt is listening through the `chrome.webRequest.onHeadersReceived` API for incoming HTTP(S) responses. In `chrome.webRequest` API, each request-response pair is represented with a unique identifier `requestId` which is used to match-up the response with the corresponding request. The *Set-Cookie* HTTP header is parsed and session cookies are identified according to the heuristic discussed in Section 5.1.

If the response is received in clear (over HTTP protocol), the CookiExt marks the identified session cookies as `HttpOnly` and if the encrypted response is received (over HTTPS protocol), all the identified session cookies are marked both as `Secure` and `HttpOnly`. In addition, the changed flag values are recorded in the list of *secured* cookies.

5.3.3 White-listing URLs

A website may support HTTPS only for some of the requests while delivers the rest of the contents over HTTP. Redirecting requests over HTTPS for non-supported parts of the websites will obviously fail. An HTTPS request may be redirected by the server over HTTP and is intercepted using Chrome API

`chrome.webRequest.onHeadersReceived` or the server returns an TLS/SSL related network error which is intercepted using `chrome.webRequest.onErrorOccurred` API. This way, a website can be checked if HTTPS is supported for the requested URL. If HTTPS is not supported, the CookiExt falls-back to HTTP to preserve functionality and adds the request URL to the *white-listed* URLs. Any future request to the same URL is not redirected over HTTPS.

In addition, to prevent a network attacker from tapping with outgoing HTTPS connections and disabling the client-side defence altogether, 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 activities on the network.

5.3.4 Redirecting HTTP Requests

When the browser sends an HTTP request, the CookiExt redirects it to HTTPS based on the following rules:

1. If the request is a page, frame or AJAX, it is redirected over HTTPS if the base domain of the request URL is in *auto-redirect* list, at least a session cookie is registered for the request and the URL is not *white-listed* for HTTP communication.

2. If the request is for a sub-resource, it is redirected over HTTPS only if the source page is over HTTPS.

As mentioned before, websites are added to *auto-redirect* list if the login operation is successful over HTTPS. This means, the site is listening at port 443 for HTTPS requests. Now, if at least a session cookie is previously registered for the request URL, it is upgraded to HTTPS to secure the session cookie. To find if the

request should be redirected and then to redirect it, each HTTP request is intercepted using Chrome API `chrome.webRequest.onBeforeRequest` listener. The request to the same URL might have been redirected in the past but had failed and was added to *white-listed* URLs. In such cases, redirecting the same request over HTTPS is useless, therefore, the CookiExt do not redirect requests that are *white-listed* for HTTP communication.

If a cookie, with its `Secure` flag set, is found in the *secured* cookies, CookiExt checks if it may be included with the request using the *domain-match* [9] rule. A website can register a cookie for itself and for any of its sub-domain and vice versa. For example, `yahoo.com` can set a cookie with domain `.yahoo.com` and the browser will include it in any request transmitted therein (e.g., to `mail.yahoo.com`). Such cookies are informally referred to as *domain* cookies.

If a website relies on HTTP requests to a sub-domain for session tracking, a naive implementation of CookiExt would break it. Assume, in fact, that the website registers a domain cookie over HTTPS to authenticate the user, the CookiExt will mark this cookie as `Secure`. Since any request sent to the sub-domain will be transmitted over HTTP and as the cookie has been marked as `Secure`, the browser will not send the cookie to the intended sub-domain. In the CookiExt implementation, whenever a domain cookie is identified as a session cookie, a redirection over HTTPS is enforced also for any sub-domain of the interacting website.

The general rule is that a request is redirected over HTTPS if the login operation to the same website was successful over HTTPS and at least a session cookie was registered for the request URL. As the request is upgraded to HTTPS, cookies with `Secure` flag set are automatically included by the browser from Chrome cookie store. As mentioned in the Section 5.3.2, a request can only be redirected to HTTPS if the URL is not *white-listed* for HTTP communication either by the

extension, or the server itself. Note that, sub-domains are not dealt with in the formal model (Chapters 3 and 4).

The rule for sub-resource requests is simple: if the source page of the request is loaded over HTTP, the cookies are already exposed in clear and the request is not redirected, but if it is over HTTPS, resource requests are redirected over HTTPS to secure the session cookies. If the page was loaded over HTTPS, any sub-resource request in clear may leak the session cookies as the browser enclose all the cookies registered by the page domain with request to the same domain.

5.3.5 Challenges in Practice

The security policy adopted in CookiExt seems straight forward, however this simple picture is complicated by a number of issues in practice. There are many websites that support HTTPS only for some content and delivers the rest of parts over HTTP. Some websites need scripts to access cookies for the desired functionality, however, this behaviour conflicts with the security policy enforced in CookiExt. A website may accept HTTPS requests but redirect them back over to HTTP and if the extension redirect them to HTTPS, it results into looping. All these different web behaviours lead to different problems discussed below.

Supporting Mixed-Content Websites Mixed-content websites have support for 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 catalogues 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, also the adoption of a fallback to HTTP would eventually break the user's session. Since session

cookies are by default marked `Secure` by the extension, they will not be sent to the HTTP portion of the website.

It was 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. Thus, any session cookie registered by a mixed-content website is initially marked as both `HttpOnly` and `Secure`. The later flag, however, is removed when a lack of HTTPS support is detected for a portion of the website and the session is proceeded over HTTP. Clearly, implementing this behaviour requires to safely detect when HTTPS is not supported. The key design choice for ensuring usability is to mark a session cookie as `Secure` and start redirecting HTTP requests to HTTPS only whenever the initial login operation happened over HTTPS. This guarantees that the remote web server is listening on port 443, thus there is no need to set a timeout and the Chrome API is leveraged to detect a number of network connection errors which may arise when HTTPS is not supported.

Preventing Redirection Loops Several websites accept HTTPS requests, but enforce a redirection over HTTP in the response. This is a bad security practice, since inattentive users may be fooled in believing to operate over HTTPS even though the website is actually deployed over HTTP.

In CookiExt setting, however, this may additionally lead to redirection loops and negatively impact on the user experience, since CookiExt may end-up forcing a redirection from HTTP to HTTPS for a given website, while the website may conversely try to perform a redirection from HTTPS to HTTP. The implementation of CookiExt breaks these loops by processing incoming redirects and blocking further redirects. When the destination URL is HTTP and at the same time included in the list of *white-listed* URLs, it is not redirected over HTTPS: a URL is added to

the white list if the request to that URL is redirected by the server back to HTTP or a network error related to TLS/SSL is received.

Enabling Web Features As mentioned in the Section 5.3.1, CookiExt sets `HttpOnly` flag of every cookie to thwart web attacks by preventing client-side scripts to access cookies. However, some websites use scripts to access cookies to track user's activities (e.g., user chat enable or disable in `facebook.com` and `gmail.com` or displaying user's name on the page from `fifa.com`). To enable such web features, CookiExt allows the user to relax the security policy enforced for that particular web site only for `HttpOnly` flag. When the user choose to white-list a website only for `HttpOnly` flag, all the altered `HttpOnly` flags for that website are reverted back to their original status and no `HttpOnly` flag is set for that website in future.

5.4 Analysis of CookiExt

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 and on the other, false positives may hinder the usability of the browser. For instance, if a cookie is improperly flagged as `HttpOnly` by the extension, no script may be able to access it, which may break the website. In this section, the additional protection enabled by CookiExt as well as its impacts on the user experience are analysed.

5.4.1 Methodology

To study the protection provided by CookiExt and its impact on the user experience, personal accounts on the top 100 websites from Alexa were created which

provide a private area. After authenticating to each website, an in-depth navigation was performed to collect data and evaluated the user experience. The data collection was restricted to the websites of interest to make the analysis more precise: for instance, third-party cookies or cross-domain requests are not included in the analysis.

Overall, a total of 733 cookies registered through HTTP(S) headers were collected: 303 of them were marked as session cookies by the heuristics used. For each session cookie, a record of the original flags as well as of the new flags which were assigned by CookiExt, was kept. To understand the usefulness of the idea of redirecting HTTP requests over HTTPS, 32,622 HTTP requests were analysed. The details of the experiments are reported as the following.

5.4.2 Evaluating Protection

Table 5.2 shows the original flags assigned to the session cookies registered by the websites that were considered in the experiments. These data are more precise than those in Section 5.2, since they were collected after authenticating to the websites and navigating them. Notice, though, that they confirmed a worrying lack of protection: more than a half (58.1%) of the session cookies had no security flag set and were vulnerable to web threats and/or network attacks.

Table 5.2: Original session cookie flags

`HttpOnly`	`Secure`	Cookies	Percentage
√	√	48	15.8%
√	×	66	21.8%
×	√	13	4.3%
×	×	176	58.1%

To evaluate the protection CookiExt provides, the original flags of the cookies received over HTTP headers are compared with the new flags which were set

(represented with √) by the extension. Table 5.3 details which additional flags were granted by CookiExt with respect to the original website behaviour.

Table 5.3: Secured session cookies

`HttpOnly`	`Secure`	Cookies	Percentage
√	√	92	30.4%
√		97	32.0%
	√	19	6.3%
	reverted	23	3.1%

The improvement in cookies protection is clear from the table. For instance, 30.4% cookies were originally completely unprotected (without any flag set), while they could be secured against both web threats and network attacks using security flags. It was noticed that 32.0% cookies were unprotected, but they could be protected at least against XSS attacks using the `HttpOnly` flag. Finally, 6.3% cookies were originally protected against web attacks, but they could be effectively secured also against network threats by setting the `Secure` flag. As a side-note to the table, 3.1% session cookies were originally marked as both `HttpOnly` and `Secure` by the extension, but they were eventually downgraded to `HttpOnly` when navigating the website to keep the session alive. There was at least a request that could not be redirected successfully to HTTPS and hence the session cookie, with `Secure` flag set by the extension, could not be included with the request which breaks the session. For functionality reasons, the cookie is reverted back to non-`Secure` cookie and then can be included with requests over HTTP.

To understand the impact of CookiExt in terms of successful HTTPS redirections, any redirection attempt performed by the extension was logged. The results are summarized in Table 5.4, where the requests are discriminated based on their types: main page and sub-resource[5].

In these experiments, in total 1344 pages, with 322 page requests originally over

[5]The Chrome API provides facility to identify these request types.

Table 5.4: Redirected requests

Type	Total	Redirected	Success	Percentage
Page	1344	322	166	51.5%
Resource	31278	158	154	97.5%

HTTP, were navigated among which 51.5% were successfully redirected to HTTPS. This suggests that many websites do not provide their pages over HTTPS, even though HTTPS support is available. Given the poor deployment of the `Secure` flag, these data suggest that many major websites do not provide a satisfactory level of protection against network attacks on web authentication. It was also observed that 158 HTTP requests for sub-resources (e.g., images and scripts) were redirected on HTTPS by CookiExt and only four redirections failed. Interestingly, 138 (87.3%) of these redirected requests were 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 as, otherwise, the sub-resource requests in clear will include the session cookie and will void the security provided by the secure page.

On further investigating the numbers above, it was 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 an HTTP request was successfully redirected to HTTPS and no redirection failed. Such websites mostly benefit from the use of CookiExt, since they can be completely secured against network attacks.

5.4.3 Evaluating the Heuristic

To evaluate the effectiveness of the session cookie detection heuristic, some more experiments were carried out[6]. To understand the practical impact of false negatives, the survey of top 100 websites was analysed and the cookies flagged either `Secure` or `HttpOnly` were isolated which were *not* identified as session cookies by the heuristic. The intuition here was that cookies which were explicitly protected by web developers were likely to contain session information and they were deemed as potential false negatives. Only 62 of the 733 cookies ignored by the heuristic had at least one security flag set. In addition, 16 of these 62 cookies were already flagged `Secure` and `HttpOnly`, hence missing them was completely harmless.

To further evaluate the security of CookiExt, for each website, all the cookies which have been marked as session cookies by the extension were deleted: a logout implied that all the real session cookies had been identified by the heuristic. In 90 out of 100 cases, the user logged out, which indicated that most of the time the heuristic approximated the real set of session cookies correctly.

Web applications may be produced using web application frameworks (e.g., ASP.NET, JavaEE) and off-the-shelf content management systems (e.g., Drupal, Wordpress). As these frameworks also handle web aspects such as those concerning session cookies, the heuristics used may not correctly reflect the every day usage of the average user. It would be interesting to complement the above study with analysis of the performance of the heuristics when used to identify session cookies stored by websites developed with these frameworks, however, it is left to future work.

[6]In a parallel study, a more systematic analysis was performed. The interested readers are referred to the work by Calzavara et al. [25] for further information.

5.4.4 Evaluating Usability

To understand the practical impact of the false positives, the CookiExt was tested and hands-on experience was observed. In these experiments, the most serious concern was about the web session being broken by the security policy applied by CookiExt. While complete sessions break never happened in practice, however, it was noticed that some 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 `www.gmail.com`, email deletion in `www.gmail.com` and appearing user's name on page from `www.fifa.com` after the user is logged in, were disabled. On further investigation, it was observed that each of these sites registered non-`HttpOnly` session cookies through HTTPS (cookies `c_user`, `GMAIL_AT` and `FIFACom` were registered by `www.facebook.com`, `www.gmail.com` and `www.fifa.com`, respectively) and the CookiExt set their `HttpOnly` flags, making them inaccessible to JavaScript opposed to as intended behaviour.

To enable features such as in these sites, CookiExt allows the user to add such sites to the list of sites white-listed for `HttpOnly` cookies and non-`HttpOnly` cookies received over HTTP(S) are not marked as `HttpOnly` afterwards. Even though a slight performance degradation was observed when CookiExt was activated, however, at the time of this writing, no major usability issue was found in the updated prototype of CookiExt.

6

Security of CookiExt-Patched Web Browser

Reading the noninterference result (Theorem 1) carefully, one can argue that it 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. In other words, the noninterference property has been proved for the Coq model of the browser without the features in CookiExt enabled. 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 correct flagging to cookies received through network input events, *de facto* enforcing the similarity condition on the input streams that constitutes the hypothesis of the noninterference definition. A formal proof of the browser augmented with the security policy in CookiExt, however, is required to provide mathematical guarantees of the CookiExt-patched web browser. In this chapter, the underlying design intuition of CookiExt is formalized and then the security of the model patched with it is proved in Coq.

6.1 Interpretation of CookiExt in Coq

The behaviour of CookiExt is stateful: that is, the extension dynamically updates an internal database to keep track of HTTP links that can be rewritten (redirected) to HTTPS and websites that should fall back to HTTP (as HTTPS is not supported). Moreover, CookiExt generates additional network traffic with respect to a standard web browser, since HTTP links must be tested to determine if an HTTPS redirection is supported. On the other hand, it was observed that these dynamic aspects disappear when the internal database of CookiExt grows large enough, for example, after the CookiExt user has fully navigated an arbitrarily large set of known websites. Since users tend to navigate always the same websites, the dynamic aspects described above are abstracted in the proof and the semantics of CookiExt is defined in terms of a *stateless* (two-step) translation from input events to *more secure* input events. This approach makes the soundness proof in Coq simpler, modular and more elegant.

For the universe of URLs navigated by the user in the past using a CookiExt-extended browser, on-going session is supposed to span over a subset of the universe and the universe of URLs is stipulated to partition into two subsets of working HTTP and HTTPS URLs, respectively. A domain is fully supporting HTTPS if any HTTP URL comprised of this domain can be successfully redirected over HTTPS. In Coq, this is defined as a predicate relation `support` in Figure 6.1.

Figure 6.1: `support` relation

```
1 | Definition support (S: DomainSet.t) (d: domain) : Prop :=
2 |   DomainSet.In d S.
```

The relation `support` states that the `domain` in argument is in the set of domains with full HTTPS deployment.

6.1.1 Rewriting URLs

The first step of the translation is given by the HTTPS rewriting of an input event. This translation step corresponds to CookiExt upgrading HTTP requests to HTTPS for supporting URLs. The rewriting upgrades an HTTP URL to HTTPS, provided that the latter protocol is supported. The rewriting process is extended to network connections by performing the HTTPS rewriting of the URL in network connection and to network responses by performing the HTTPS rewriting of any URL syntactically occurring in the response (e.g. the *scr* attributes of the *<script>* tags). All these URLs in the response eventually result HTTP requests which are redirected to HTTPS by the CookiExt if HTTPS is supported. In CookiExt formalisation, this is modelled as rewriting all the HTTP URLS to HTTPS.

A rewrite function `rewrite_url` (`rewrite_urlo` for optional `url`) just replaces the `http_protocol` of the `url` given as argument with `https_protocol`. A network response `resp` may include HTML document `list doc_tree` which in turn include remote script nodes `rem_script_doc` each with `url` from where the scripts are to be loaded. The function `rewrite_list_doc_tree` is defined in the Figure 6.2. It takes the list of document trees `list doc_tree`, which represent the HTML document received, and rewrites the HTTP URLs to HTTPS in the remote script nodes of all the trees.

The script in the response `resp` may also contain a script `url`, which is also rewritten to HTTPS. The function `rewrite_script` in Figure 6.3 gets the `script` as argument and rewrites the `url` in the `url_script`.

The body of the response `resp_file` is modelled as inductive type `file` (Figure 3.7), which include HTML document `html_file` and scripts `script_file`. The function `rewrite_file` (Figure 6.4) takes a `file` as input and rewrites all the URLs of type `url`, in both the `html_file` and `script_file`, using functions `rewrite_list_doc_tree` and `rewrite_script` respectively.

Figure 6.2: `rewrite_list_doc_tree` function

```
Fixpoint rewrite_list_doc_tree (ldt: list doc_tree)
  {struct ldt} : list doc_tree :=
   match ldt with
   | dt::ldt' ⇒
      match dt with
      | rem_script_doc eio u ⇒
         rem_script_doc eio (rewrite_url u) ::
         (rewrite_list_doc_tree ldt')
      | _ ⇒ ldt
      end
   | _ ⇒ ldt
   end.
```

Figure 6.3: `rewrite_script` function

```
Definition rewrite_script (sc: script) : script :=
  match sc with
  | url_script u ⇒ url_script (rewrite_url u)
  | _ ⇒ sc
  end.
```

Figure 6.4: `rewrite_file` function

```
Definition rewrite_file (f: file) : file :=
  match f with
  | html_file ldt ⇒ html_file (rewrite_list_doc_tree ldt)
  | script_file sc ⇒ script_file (rewrite_script sc)
  | _ ⇒ f
  end.
```

Finally, the function `rewrite_ie` (Appendix A.2) takes an input event as argument and rewrites all the URLs occurring anywhere in the event. This include the request/response URL (the `url` parameter in the `net_conn_id`), redirect URL (the `resp_redirect_uri` value), the URLs in the HTML document (such as the `url` parameters in the remote scripts `rem_script_doc`) and the URLs in the script (`url_script`).

6.1.2 Updating Cookies

In the second step of the translation, security flags of the cookies in a network input event are upgraded. This translation step corresponds to the CookiExt setting the correct security flags for incoming (session) cookies (Section 5.3.2): the cookie flags are set with both `Secure` and `HttpOnly` flags if the website is supporting HTTPS, otherwise, only `HttpOnly` flag is set. The function `flag_cookies` defined in Figure 6.5 does exactly that. The network response events set cookies in the browser through HTTP *Set-Cookie* header, which is modelled as the field `resp_set_cookies` of the type `resp` (Figure 3.5) representing the body of network response.

Figure 6.5: `flag_cookies` function

```
1  Definition flag_cookies (S: DomainSet.t) (u: url)
2    (rssc: StringMap.t cookie_flags_value)
3    : StringMap.t cookie_flags_value :=
4    match u with
5    | http_s_url prot d p ⇒
6       if support_dec_b S d
7       then StringMap.map (fun z ⇒ set_flags z) rssc
8       else StringMap.map (fun z ⇒ set_httponly_flag z) rssc
9    | _ ⇒ StringMap.empty
10   end.
```

The function `flag_cookies` gets the list of cookies (the value of the field

`resp_set_cookies` defined as a map) and the URL of the response and sets the flags if the website is supporting HTTPS. Both, `Secure` and `HttpOnly`, flags are set (the function `set_flags` at line 7) if the response is received over HTTPS, otherwise, it sets only the `HttpOnly` flag (the function `set_httponly_flag` at line 8). This corresponds to setting the `Secure` flag only for cookies registered by websites with full HTTPS support. The HTTPS support checking is modelled as the proposition `support` (Figure 6.1) and is checked using the boolean function `support_dec_b` (Figure 6.5, line 6).

The function `upgrade_cookies_ie` (Figure 6.6) updates the flags of the cookies, using the function `flag_cookies`, received in the *Set-Cookie* headers of the network response input event while the user input event is left unchanged.

Figure 6.6: `upgrade_cookies_ie` function

```
1  Definition upgrade_cookies_ie (S: DomainSet.t) (ie: input_event)
2    : input_event :=
3     match ie with
4     | network_document_response_event nci uwi rs ⇒
5        network_document_response_event nci uwi
6        (build_resp rs.(resp_del_cookies)
7         (flag_cookies S nci.(net_conn_id_url)
8         rs.(resp_set_cookies))
9        rs.(resp_redirect_uri) rs.(resp_file))
10    | network_script_response_event nci rs ⇒
11       network_script_response_event nci
12       (build_resp rs.(resp_del_cookies)
13        (flag_cookies S nci.(net_conn_id_url)
14        rs.(resp_set_cookies))
15       rs.(resp_redirect_uri) rs.(resp_file))
16    | network_xhr_response_event nci rs ⇒
17       network_xhr_response_event nci
18       (build_resp rs.(resp_del_cookies)
19        (flag_cookies S nci.(net_conn_id_url)
20        rs.(resp_set_cookies))
21       rs.(resp_redirect_uri) rs.(resp_file))
22    | _ ⇒ ie
23    end.
```

6.1.3 Translating Input Events

Given the definitions of rewriting URLs and updating cookies, the semantics of CookiExt is characterized in terms of the two-step translation. The translation include performing the HTTPS rewriting of any URL syntactically occurring in network response and updating all the cookies in the response *Set-Cookie* headers. This is defined by the function `translate_ie` in Figure 6.19, which simply is using the two functions `rewrite_ie` and `upgrade_cookies_ie` defined above. The translation is extended to the stream of events using the function `translate_iel` (Figure 6.19).

Figure 6.7: `translate_ie` function

```
1 | Definition translate_ie (S: DomainSet.t) (ie: input_event)
2 |   : input_event := upgrade_cookies_ie S (rewrite_ie ie).
```

Figure 6.8: `translate_iel` function

```
1 | Fixpoint translate_iel (S: DomainSet.t) (iel: list input_event)
2 |   : list input_event :=
3 |    match iel with
4 |    | nil ⇒ iel
5 |    | ie::iel′ ⇒ translate_ie S ie::(translate_iel S iel′)
6 |    end.
```

6.2 Confidentiality Policy

To generalize the noninterference result in Theorem 1 to a CookiExt-extended browser, a new similarity relation for input events is introduced. This similarity relation is different than `sim_ie` introduced in the Figure 3.21 as it does not depend on the cookie flags, since CookiExt applies them automatically.

Figure 6.9: `cookie_label_plus` function

```
1 Definition cookie_label_plus (S: DomainSet.t) (u: url) :=
2   if support_dec_b S (domain_url u)
3   then https_label (domain_url u)
4   else http_label (domain_url u).
```

Figure 6.10: `erase_invis_cookies_plus` function

```
1 Definition erase_invis_cookies_plus (l: label) (S: DomainSet.t)
2   (u: url) (rssc: StringMap.t cookie_flags_value)
3   : StringMap.t cookie_flags_value :=
4    if cookie_label_plus S u <?= l
5    then rssc else StringMap.empty.
```

This difference can be seen when labels are assigned to cookies using the function `cookie_label_plus` (Figure 6.9). This function, like many others defined in this chapter, is parametric over the set of websites with full HTTPS support. As mentioned in the Section 6.1, flags have already been updated in the input translation phase depending on the URL of the response and the website. A cookie is assigned `https_label` if the website which registered it (the `domain` of the response `url`) is fully supporting HTTPS, otherwise, it is assigned `http_label`. The function `cookie_label_plus` only checks if the website is with full HTTPS support and if it is, then CookiExt must have already been successfully redirected the request over HTTPS and the cookie flags must have been set to `Secure` and `HttpOnly`, otherwise, to `HttpOnly`. The function `cookie_label_plus` is then used in the function `erase_invis_cookies_plus` (Figure 6.10) to erase confidential cookies from the set of cookies received in the response.

After defining the pieces of security policy for CookiExt, all that is needed to put these pieces together in the required form. An updated version of the binary relation `same_form_ie` (Figure 3.19), `same_form_ie_plus` (Appendix A.3), does

exactly that by combining these pieces in one equivalence relation. The only differences between these functions are that the later is parametric over the set of websites with full HTTPS support and replaces the function `cookie_label` with `cookie_label_plus` to assign labels to the cookies received in the input response.

Figure 6.11: `sim_ie_plus` relation

```
1 | Inductive sim_ie_plus (l: label) (S: DomainSet.t)
2 |   : input_event → input_event → Prop :=
3 |    | sim_ie_same_form: ∀ ieL ieR,
4 |       same_form_ie_plus l S ieL ieR →
5 |       sim_ie_plus l S ieL ieR.
```

Finally, the predicate `sim_ie_plus` in the Figure 6.11 defines the important relation on input events. Two input events are considered to be *similar* if they are in the same form according to the new definition `same_form_ie_plus`. In other words, two input events are similar if they differ only in the confidential cookies *after* the cookies have been flagged by the CookiExt. The relation on input events can be easily extended to similarity on stream of events `sim_iel_plus` (Figure 6.12). In addition, stream similarity relation over original input events `sim_iel` is also added (Figure 6.13).

Figure 6.12: `sim_iel_plus` relation

```
1 | Definition sim_iel_plus (l: label) (S: DomainSet.t)
2 |   : list input_event → list input_event → Prop :=
3 |   Forall2 (sim_ie_plus l S).
```

It is observed that the relation `sim_ie_plus` deems as similar much more inputs than `sim_ie` (Figure 3.21) for any label `l`, hence it can be leveraged to prove a stronger noninterference result. Specifically, it is easy to find two input events `ieL` and `ieR` such that (`sim_ie_plus l S ieL ieR`) holds for any label `l` and set

Figure 6.13: `sim_iel` relation

```
Definition sim_iel (l: label)
  : list input_event → list input_event → Prop :=
  Forall2 (sim_ie l).
```

`S` of websites with full HTTPS support, but (`sim_ie l ieL ieR`) does not. For instance, two identical document responses `ieL` and `ieR` over HTTP from domain `d`, with the only difference that `ieL` sets the `HttpOnly` flag on its cookies, while `ieR` does not. In this case, for a web attacker `l = http_label d`, the proposition (`sim_ie_plus l S ieL ieR`) holds, but (`sim_ie l ieL ieR`) does not. The reason is that in the former case, CookiExt sets the `HttpOnly` flags of all the cookies in `ieR` and hence the web attacker at `l = http_label d` will not be able to draw a distinction between the corresponding outputs, but in the later case, the attacker can differentiate between the outputs by providing a script, exploiting a XSS vulnerability, to read a cookie set by `ieR`. A similar reasoning applies to the HTTPS case.

Figure 6.14: `ie_no_secure_cookie` function

```
Definition ie_no_secure_cookie (ie: input_event) : Prop :=
  match ie with
  | network_document_response_event nci uwi rs ⇒
     rs.(resp_set_cookies) ==
      StringMap_key_filter_plus (fun b ⇒ !b) rs.(resp_set_cookies)
  | network_script_response_event nci rs ⇒
     rs.(resp_set_cookies) ==
      StringMap_key_filter_plus (fun b ⇒ !b) rs.(resp_set_cookies)
  | network_xhr_response_event nci rs ⇒
     rs.(resp_set_cookies) ==
      StringMap_key_filter_plus (fun b ⇒ !b) rs.(resp_set_cookies)
  | _ ⇒ True
  end.
```

In addition, an axillary function is needed to check that the input event has no

cookie with `Secure` flag set. The proposition `ie_no_secure_cookie` defined in the Figure 6.14 gets an input event and holds if the *Set-Cookie* headers in the network response have no `Secure` cookie. The function `StringMap_key_filter_plus` uses the Coq function `filter` that erases all the cookies, from the set of cookies received in the network response, with `Secure` flags set.

Given the similarity on input events `sim_ie` (Figure 3.21), a relation with similar-plus `sim_ie_plus` can be established. The relation `sim_ie_plus` is coarser than `sim_ie`. As a consequence, showing noninterference property with respect to `sim_ie_plus` rather than `sim_ie` provides stronger security guarantees.

6.2.1 Relation `sim_ie` Versus `sim_ie_plus`

To show that the relation `sim_ie_plus` is actually larger than `sim_ie`, it was first observed that the two relations coincide on user input events, as such events do not set cookies in the browser. As to network events, where the two relations differ, it can be easily noticed that `sim_ie` implies that both, `ieL` and `ieR`, come from the same URL (by the definition of `sim_ie` in the Figure 3.21). In the next two lemmas, the relation between these two relations on input events is defined and proved.

Figure 6.15: `sim_ie_sim_ie_plus_equiv_https` lemma

```
1 | Lemma sim_ie_sim_ie_plus_equiv_https: ∀ l S ieL ieR,
2 |   sim_ie l ieL ieR →
3 |   support_dec_b S (domain_ie ieL) = true →
4 |   sim_ie_plus l S ieL ieR.
```

The first lemma `sim_ie_sim_ie_plus_equiv_https` (Figure 6.15) states that two similar events are similar-plus if the websites (domains) are fully supporting HTTPS. As the events are similar, this implies their protocols and domains are

the same, so the hypothesis that the source website of the event `ieR` is fully supporting HTTPS can be derived[1]. As both the events are similar (with the same domains, protocols and HTTPS support), they will be successfully redirected by the CookiExt over HTTPS and hence both the flags will be set, implies they will be similar-plus. This is further elaborated with the following examples.

For an attacker at `l = http_level d`, two similar events at HTTP contains the same set of cookies without any or with `HttpOnly` flags. After redirecting over HTTPS, the flags of the same set of cookies in both events will change to `Secure` and `HttpOnly`, resulting two similar-plus events. Similarly, for an attacker at `l = https_level d`, two similar events will have the same set of cookies regardless of the flags (as each time the cookie flag will be at or below `https_level d`), and with the CookiExt, all the cookies will have all the flags set, again resulting similar-plus events. The proof is achieved by case analysis on the structure of input event `ieL`.

Figure 6.16: `sim_ie_sim_ie_plus_equiv_http` lemma

```
1 | Lemma sim_ie_sim_ie_plus_equiv_http: ∀ l S ieL ieR,
2 |   sim_ie l ieL ieR →
3 |   support_dec_b S (domain_ie ieL) = false →
4 |   ie_no_secure_cookie ieL →
5 |   ie_no_secure_cookie ieR →
6 |   sim_ie_plus l S ieL ieR.
```

According to the lemma `sim_ie_sim_ie_plus_equiv_http` in Figure 6.16, two similar events are similar-plus if the websites are not fully supporting HTTPS and the events do not contain any `Secure` cookie. Notice, however, that this lemma does not involve any loss of generality in practice, since websites without full HTTPS support do not mark their cookies as `Secure` (this would break the session when navigating the HTTP portion of the website as `Secure` cookies will

[1]It could be added as an extra hypothesis, but its just a matter of taste and is instead proved.

not be attached with HTTP requests).

6.3 Patching the Browser With CookiExt

A CookiExt-patched Extended Featherweight Firefox (EFF$^+$) model is obtained from the EFF model (Chapters 3 and 4) by applying the translation `translate_ie` (Figure 6.19) to any input event before processing it. A proof technique, consisting of a number of axillary lemmas, is developed to ease the proof of information-flow security of EFF$^+$ (Section 6.4) with respect to the new similarity relation `sim_ie_plus` on input events. The main approach used in this technique is to prove that given two similar-plus streams of input events, they are similar after their translation: all the URLs are rewritten and the cookies are updated. The key to this proof is to prove two lemmas corresponding to these transformations first.

The first lemma `sim_iel_plus_rewrite_equiv` (Figure 6.17) corresponding to URL rewriting, states that two similar-plus input event streams are still in the same relation after all the URLs in the events are rewritten using the `rewrite_ie` function. The Coq function `map` (lines 3 and 4) maps the input streams by rewriting each event in the stream using the function `rewrite_ie`. This lemma is straightforward as the corresponding events in both streams before rewriting have the same protocols by the definition of `sim_ie_plus` and rewriting their protocols to HTTPS (if they are HTTP) will keep them in the relation.

Similarly, the second lemma `sim_iel_plus_update_cookies_equiv` (Figure 6.18) corresponds to updating the cookies in input streams. Two input event streams related by `sim_ie_plus` are also related by `sim_ie` after the cookie flags of the corresponding individual events in both streams are updated. The function `upgrade_cookies_iel` (lines 3 and 4) operating on list is used to update cookies in

Figure 6.17: `sim_iel_plus_rewrite_equiv` lemma

```
1 | Lemma sim_iel_plus_rewrite_equiv : ∀ l S IL IR IL′ IR′,
2 |   sim_iel_plus l S IL IR →
3 |   IL′ = map rewrite_ie IL →
4 |   IR′ = map rewrite_ie IR →
5 |   sim_iel_plus l S IL′ IR′.
```

the stream instead of `map` (as used in the lemma `sim_iel_plus_rewrite_equiv`) where each element is transformed by applying the translator function. This is just a mater of enjoying different Coq programming tasts and does not affect the proof.

Figure 6.18: `sim_iel_plus_update_cookies_equiv` lemma

```
1 | Lemma sim_iel_plus_update_cookies_equiv: ∀ l S IL IR IL′ IR′,
2 |   sim_iel_plus l S IL IR →
3 |   IL′ = upgrade_cookies_iel S IL →
4 |   IR′ = upgrade_cookies_iel S IR →
5 |   sim_iel l IL′ IR′.
```

Finally, the main lemma of input translation `cookie_translation` (Figure 6.19) relating two relations, the `sim_iel_plus` and the `sim_iel`, is proved. It simply states that two similar-plus event steams can safely be translated into two similar event streams by rewriting the URLs and updating the cookies in all events of the streams. This is proved easily by applying the corresponding two lemmas proved earlier: the lemma `sim_iel_plus_rewrite_equiv` and the lemma `sim_iel_plus_update_cookies_equiv`.

Figure 6.19: `cookie_translation` lemma

```
1 | Lemma cookie_translation: ∀ l S IL IR IL′ IR′,
2 |   sim_iel_plus l S IL IR →
3 |   IL′ = translate_iel S IL →
4 |   IR′ = translate_iel S IR →
5 |   sim_iel l IL′ IR′.
```

6.4 Proof of Session Confidentiality

The EFF model patched with the model of CookiExt (EFF^+) is proved to be noninterferent to ensure augmenting the web browser with CookiExt still preserves end-to-end web session security.

Theorem 2. *EFF^+ is noninterferent according to the security policy defined in Section 6.2.*

Proof. Let $I \approx_l^+ I'$, then by lemma in Figure 6.19 we know that $[\![I]\!]_{\mathcal{U}} \approx_l [\![I']\!]_{\mathcal{U}}$, where $\approx_l^+$ is the relation `sim_iel_plus` (Figure 6.12), $\approx_l$ is the relation `sim_iel` (Figure 6.13) and $[\![I]\!]_{\mathcal{U}}$ represents the two-step translation (Figure 6.8). Let EFF_{init} be the initial state of the original EFF model, then by Theorem 1 we know that $\text{EFF}_{init}([\![I]\!]_{\mathcal{U}}) \Rightarrow O$ and $\text{EFF}_{init}([\![I']\!]_{\mathcal{U}}) \Rightarrow O'$ imply $O \approx_l O'$. Now let EFF^+_{init} be the initial state of EFF^+, by construction we know that $\text{EFF}^+_{init}(I) \Rightarrow O$ and $\text{EFF}^+_{init}(I') \Rightarrow O'$, hence we conclude. □

7

Web Session Integrity: Access Control Enforcement

There are a number of ways used by the attackers to force the browser to send authenticated requests to victim website impersonating the user, for example, through the well-known class of attacks CSRF (Chapter 1). In this chapter, a formal notion of session integrity is introduced and a security enhanced version of EFF with a mechanism to enforce session integrity is developed. The integrity security policy is enforced using access control model where the network connections and web pages are dynamically tainted. The enforcement mechanism captures attacks on session integrity such as (classic, local and login) CSRF, session fixation, password theft and reflected XSS attacks. In addition, the design of a prototype web browser extension is given that uses, in practice, a relaxed version of security policy enforced in EFF.

7.1 Web Session Integrity

The formal notion of session integrity [23] defines how an attacker can influence execution traces of the reactive system (browser) and then session integrity is defined as the property that the attacker has no effective way of interfering with an authenticated session.

Assume a lattice of security labels $(\mathcal{L}, \sqsubseteq)$, with bottom and top elements $\bot$ and $\top$, respectively. With each output event of a reactive system, a label in $\mathcal{L}$ is associated by way of a *trust* mapping[1] $\tau : \mathcal{O} \to \mathcal{L}$ where each label in the lattice corresponds to an interaction point for the reactive system (an *origin*, in the context of web systems) and $\tau(o) = l$ indicates that $o \in \mathcal{O}$ is a message output by the reactive system (the browser) in an authenticated session with l's endpoint (e.g., network requests to website A when the user is signed into A). Whenever o does not belong to any authenticated session (e.g., network requests to website A when the user is *not* signed into A), then $\tau(o) = \bot$ ($\tau_\bot$ henceforth). The trust changes dynamically, noted $\tau \xrightarrow{o} \tau'$, upon certain output (authentication) events (e.g., output sent after the user is signed into the website). Finally, the $O \downarrow l$ denotes the stream that results from O by considering only the events at trust level l.

Definition 2 (Session integrity)**.** *A reactive system preserves* session integrity *for its trace* (I, O) *iff for all* $l \in \mathcal{L}$*, and all its attacked traces* (l, I, O') *one has:*

$$\forall l' \not\sqsubseteq l : O' \downarrow l' \textit{ is a prefix of } O \downarrow l'.$$

Where the stream O'*, in the attacked trace, include messages generated by the attacker at label* l*. A reactive system preserves session integrity if and only if it preserves session integrity for all its traces.*

Session integrity ensures that the attacker has no effective way to interfere with any authenticated session within the set of traces. In particular, if the trust mapping remains constant at $\tau_\bot$ along the trace, no authentication event occurs in O and the attacker may only initiate its own authenticated sessions, at level l or

[1]The intuition here is that output event to a website A gets different labels depending on whether or not the user is signed into A. The trust mapping captures this change in trust level.

lower. If instead the trust mapping does change, to include authenticated output events at level $l' \not\sqsubseteq l$, then the requirement that $O' \downarrow l'$ be a prefix of $O \downarrow l'$ ensures that the attacker will at best be able to interrupt the on-going sessions, but not otherwise intrude into them.

7.2 Flyweight Firefox Browser Model

Extended Featherweight Firefox model provides precise and faithful abstraction of current web browsers and just like the browsers it models, it is vulnerable to a variety of attacks on session integrity. In Chapter 3 and 4, the security guarantees of the protection mechanisms to protect session confidentiality were proved using noninterference, however, the model still does not enforce any integrity policy.

Enforcing session integrity in EFF and then proving its soundness according to those policies would be challenging. There were two choices; 1) extend EFF with the new features sufficient for defining meaningful integrity policies and the integrity enforcement mechanism for these policies and then prove its soundness in Coq; or 2) build a new simplified version of EFF, extend it with security mechanism and prove its soundness. The first choice is not feasible, as the current version of EFF model and the confidentiality proofs are very huge (consists of more than 45,000 lines of code). Defining integrity policies, enforcing them in the model and then proving their soundness appears to be more tedious than the confidentiality. Adding more complex security policies to a large model would simply have over complicated the model and would be difficult to close the proof. Therefore, the later approach was taken and, after carefully choosing the features required to enforce integrity policies, a lightweight version of EFF called Flyweight Firefox was introduced [23, 22].

Flyweight Firefox is a core model of a web browser distilled from the Feath-

erweight Firefox Coq model [17, 18]. It consists of almost the same features as EFF but simplified: for example, it consists of fewer scripts than in EFF and include few new scripts that are required to enforce meaningful integrity policies. A security-enhanced extension of Flyweight Firefox (FF^+) was introduced that provides a full-fledged enforcement of web session integrity based on access control. The runtime mechanisms underlying FF^+ are robust against both web threats and network attacks, and the resulting model is concrete enough to be amenable for an almost direct implementation, while at the same time being fit for a rigorous formal treatment and a security proof. To prove the security of the lightweight model FF^+, unlike the bi-simulation based proof technique [19, 17] as used in Chapters 4 and 8, a *simulation* based proof technique was introduced.

To describe the integrity policy and the corresponding enforcement mechanism, the EFF model in Coq was further extended with necessary features fit for enforcing the notion of session integrity. These features and the enforcement mechanism are exactly the same as used in FF^+ but are added to and applied in a detailed model EFF. This approach is taken to synchronize with Coq formalisms in Chapter 3 and 4 to ease understanding of this dissertation and build the initial foundation of mechanising session integrity in future. The readers are referred to Bugliesi et al. [23, 22] for further detail of the formal notion of integrity, FF, its security enhanced version FF^+ and security proofs[2].

7.3 Enforcement in EFF

In this section, the design of Extended EFF (EEFF), a security-enhanced extension of EFF aimed at enforcing web session integrity, is discussed. As with the Featherweight Firefox model, EFF is also missing some features and contextual

[2]None of the integrity proofs is mechanized in Coq. The integrity enforcement mechanism in EFF has not been proved sound and is left to future work.

information needed to apply a sound security policy for session integrity.

The CSRF attacks may be caused either by the 1) cross-origin requests generated by the scripts or HTML elements (e.g., <*script*> tag) with the *src* attributes set to a URL of victim website, 2) requests (same or cross-origin) initiated by the injected scripts or 3) redirecting requests to the victim website [23].

The first case can be easily captured by not allowing session cookies with cross-origin requests or blocking such requests at all. The latter approach has been used by Samuel [102] where non-white-listed cross-origin traffic is blocked, however, it results usability issues and is not used here. Striping cookies from the *malicious* cross-origin requests, on the other hand, is one of the popular approaches adopted to protect against CSRF attacks [40, 100]. The same approach is used here, however, both the legitimate and malicious cross-origin requests are stripped. Striping the legitimate cross-origin requests creates usability problems in the third-party widgets (e.g., the Facebook *like* button) and collaborative web applications such as e-payment systems (e.g., PayPal). The problem could be solved by separating *expected* (legitimate) and malicious cross-origin requests and then limiting stripping only to the later. This technique has been used in [40, 100], however, it is heuristic-based and hence needs to be replaced by a more rigorous and precise method, for example, getting help from server side (see Chapter 8 for a proposal). Moreover, striping only cross-origin requests is not enough in many cases as the attacker can also create authentic same-origin requests, for example, by injecting scripts, an issue addressed in this chapter (see below).

The second case, however, is challenging as the script injected by an attacker can trigger the same-origin authentic requests. The problem is that the injected script gets the origin of the page where it is injected according to the same-origin policy and hence the request generated by that script to the origin of the page is considered as same-origin, but it is initiated by the attacker. Enforcing a mech-

anism to capture such attacks is tricky and hence requires to carefully look at different ways that may be used to inject a malicious script. In the attack scenarios existed in the literature such as reflected XSS and session fixation and the new one local CSRF (Figure 1(b), Figure 2(c) and Figure 1(c), respectively in Bugliesi et al. [22]), the attacker is able to launch the attack by redirecting the requests.

Attacks caused by cross-origin redirection (cause 2 and 3 above) can be captured by tracking the origin redirecting the requests. The password theft and login CSRF (Figure 2(a) and Figure 2(b), respectively in [22]) attacks are captured using different techniques, such as security contexts. The classic CSRF attack (Figure 8.3), however, can be launched without the script injection and is dealt with by erasing session information from cross-origin requests. Readers are referred to Bugliesi et al. [22] for the detailed description and encoding of how these attacks are captured.

The existing EFF model does not have any mechanism to track origin changes across network requests or capture the password theft attack. In the next subsections, the additional features are discussed that are added to store contextual information, maintain security contexts, extend scripts to introduce meaningful security policies and perform secure cookie operations.

7.3.1 Contextual Information

Whenever a network document request is redirected by an origin, the request and hence the page loaded is influenced by that origin, as the origin can inject script which upon execution can send authenticated same or cross-origin requests or the origin can redirect the request to anywhere he wants. Therefore, the simple strategy is to track the origin that redirects the request.

In EFF, to map each network response to the corresponding request, a network connection is used which is represented by a record type `net_conn_id` (Figure 3.4)

with two fields: the request `url` and an integer to uniquely identify the connection. A new filed `net_conn_id_taint` of type `bool` as the *qualifier* is added to the type `net_conn_id` to track origin change during a network request. The qualifier of the network connection changes as the following: when the user navigates the browser to a URL, a new network connection `net_conn_id` is created and it is assigned the qualifier `false` (untainted). If a cross-origin redirect is received over the connection, it is given the qualifier `true` (becomes tainted) and it will never be restored to an untainted state. The updated definition of `net_conn_id` is given in Figure 7.1, with the new field added at line number 4.

Figure 7.1: `net_conn_id` data type (with qualifier)

```
1 | Record net_conn_id: Type :=
2 |   build_net_conn_id {
3 |     ...
4 |     net_conn_id_taint: bool
5 |   }.
```

Further requests sent over a tainted network connection (`net_conn_id` with `net_conn_id_taint` set to `true`) will never include cookies to thwart CSRF attacks performed through a redirect: this policy is applied also to the same-origin requests to prevent local CSRF attacks [22]. When a document response is eventually received over the network connection, the connection is closed and a new page is stored in the browser page store `browser_pages`.

Figure 7.2: `page` data type (with qualifier)

```
1 | Record page: Type :=
2 |   build_page {
3 |     ...
4 |     page_taint: bool
5 |   }.
```

As the content loaded over a tainted connection is also untrusted, it is necessary to track the qualifier for the page as well. The page loaded should inherit the qualifier assigned to the network connection, however, when the page is loaded, the network connection used can not be retrieved later as it is closed and hence the qualifier is lost. To track the origin that may have influenced the contents of the page, a boolean field `page_taint` as the qualifier is added to the record type `page` (Figure 7.2, line 4).

Further constraints are put on setting cookies through network responses received over tainted network connections. The cookie store is updated only if the network connection was marked as untainted. This is needed to prevent the attacker from corrupting the cookies stored in the browser, for instance, by forcing a redirect on a page of a trusted website which overwrites the existing cookies with some default values (see secure cookie update operation in Section 7.3.4).

7.3.2 Security Contexts

When the user enters a password into a login form, the script running under the event handler registered on the page can steal the password and leak it to the attacker. To prevent password theft, each script is run inside a security context, i.e., a sandbox represented by a pair: script and the label of execution. This is modelled as the extended running state of the browser with each script executing with a label, as defined in the figure 7.3. The added label parameter resembles to the *security mode* in Featherweight Firefox, however, is used here for different purpose. When a password is disclosed to a script expression, a new security context is instantiated which provides EEFF with the information needed to protect the password. In the implementation of the integrity security policy as the browser extension SessInt, an internal password manager keeps track of the security context (Section 7.7).

Figure 7.3: `running_state` data type (with context)

```
1 | Definition running_state := browser * list task * label.
```

For example, when the user enters password into a login form on the page from URL `u` = (`http_s_url p d ru`), the execution label of the script under the event handler gets the value (`url_label u`) if the action URL of the login form matches with the one stored with the password in the internal password manager, otherwise, the script gets the `bot_label`. In the first case, the script is allowed to communicate only with domain `d` on the protocol `p`, while in the second case, the script can not send the password, hence protecting password theft (see Section 7.7).

7.3.3 Extending Scripts

In the session fixation attack, a web attacker can inject a script which registers a cookie chosen by the attacker [22]. To model such attack scenarios, a script setting cookie value is needed, however both, the Featherweight Firefox and its extended version EFF are missing script to set cookie values. To define and then enforce meaningful integrity policies, the `set_cookie_script` (Figure 7.4, line 3) is added to the `script` (Figure 3.9) type. The `set_cookie_script` constructor has two parameters. The first one is of type string which represents the name of the cookie to be overwritten and the second one is the value – a record type `cookie_flags_value` (Figure 3.6), representing the two security flags and a cookie value.

The login operation is modelled using a new script `auth_script` (Figure 7.4, line 4). The first parameter of this script is the password string and the second parameter is the URL where the password is sent.

Figure 7.4: `script` data type (extended)

```
1 | Inductive script :=
2 |  ...
3 |  | set_cookie_script: String.t → cookie_flags_value → script
4 |  | auth_script: String.t → url → script
```

7.3.4 Secure Cookie Operation

Many attacks [21, 23] against web sessions are enabled by the browser failing to ensure the confidentiality or integrity of authentication cookies. Updates to the cookie store in EEFF adopt a strong security policy: as in CookiExt, authentication cookies received over HTTP are marked `HttpOnly`, while authentication cookies received over HTTPS are flagged both `HttpOnly` and `Secure`. If a `Secure` cookie is sent from the server to the browser over HTTP, which is one of the many quirks allowed on the Web, it is discarded by EEFF. Moreover, EEFF ensures that `Secure` cookies are never overwritten by cookies set through HTTP responses, thus protecting their integrity against network attacks: this is not guaranteed by standard web browsers [9].

A web attacker may fixate the session by injecting the script `set_cookie_script` to set the non-`HttpOnly` session cookie to a value already known to the attacker. The standard procedure employed by the web browsers is to select the cookies to be attached to a given network request, including those fixated by the attacker. A secure counterpart of the standard procedure employed by the web browsers is introduced in EEFF to ensure that no outgoing cookie can have been fixated by an attacker. For HTTP requests, the protection against web attacks is enforced by requiring that only `HttpOnly` cookies are sent to the web server. Since these cookies cannot be set by a script, they can only be fixated by network attacks. For HTTPS requests, instead, a higher level of protection is targeted that ensures

that any cookie attached to them cannot have been fixated, even by a network attacker. Since `Secure` cookies do provide integrity guarantees against such an attacker, in EEFF, only cookies which are marked as both `Secure` and `HttpOnly` are attached to outgoing HTTPS requests.

7.4 Threat Model

In the threat model, it is assumed that all HTTPS traffic is signed using trusted certificates (unsigned HTTPS traffic is represented using HTTP). The attacker's power is characterized by a security label, with the understanding that higher labels provide additional capabilities. A novel aspect of the threat model is the assumption that the attacker has full control over compromised sessions, i.e., authenticated sessions established using the attacker's credentials.

If a network request belongs to a compromised session, it is pessimistically assumed that all the data included in the request are stored by the server in the attacker's account and later made available to him: this is useful to capture login CSRF attacks [10]. For example, search engines including Yahoo and Google allow users to opt-in to saving search history for later retrieval. As search queries may contain sensitive details about user's activities [92], they could be used by an attacker to steal the secrete data or to spy on the user (e.g., searching the word 'bomb' can motivate intelligence agencies to spy on the user).

In attacks such as login CSRF where the attacker forges a login request to an honest site using the attacker's user name and password at that site [10], the password submitted identifies the attacker's account. To model such scenario, an additional label `evil_label` is added to the type `label` as shown in the Figure 7.5 (line 3). The label `evil_label` represents the password identifying the attacker's account: for simplicity, it is assumed that this password can be used to establish

authenticated sessions on any website.

Figure 7.5: `label` data type (extended)

```
Inductive label: Type :=
  ...
  | evil_label: label
  | bot_label: label.
```

Formally, the threat model results from instantiating the definitions of *interception*, *eavesdropping* and *synthesis*. The relation `interception` is inductively defined in Figure 7.6. The first argument of the relation is a function that assigns security labels to output events. The type `event` (Figure 7.7) represents either the `input_event` or `output_event`, while the function `event_label` assigns `url_label` to network events sent or received and `top_label` to user events.

Figure 7.6: `interception` relation

```
Inductive interception:
  (output_event → label) → label → event → Prop :=
  | ii_net: ∀ tau l evt,
     event_label evt <= l → interception tau l evt.
```

Figure 7.7: `event` data type

```
Definition event := input_event + output_event.
```

The only rule of the relation states that a web attacker at a label, say `http_label d`, can intercept only the network traffic, either in clear or with no trusted certificates, sent to domain `d` while a network attacker can intercept all the traffic over HTTP (and any HTTPS message directed to him). Moreover, a net-level attacker cannot intercept arbitrary HTTPS traffic: indeed, since signed HTTPS

communication ensures both freshness and integrity [45], the attacker cannot replay encrypted messages or otherwise tamper with HTTPS exchanges without breaking the communication session. Hence, preventing the interception of arbitrary HTTPS traffic ultimately amounts just to discarding denial of service attacks, which is not dealt with in this dissertation.

The definition of the relation `eavesdropping` (Figure 7.8) consists of two rules. According to the rule `ih_net`, an HTTPS exchange can still be overheard by a net-level attacker. The network attackers are thus aware of all network traffic, even though they may be unable to access its payload. Finally, rule `ih_evil` makes any request sent over compromised sessions available to the attacker, as discussed above.

Figure 7.8: `eavesdropping` relation

```
1 | Inductive eavesdropping:
2 |   (output_event → label) → label → event → Prop :=
3 |   | ih_net: ∀ tau l evt,
4 |     event_label evt <= net_label ∧ net_label <= l →
5 |     eavesdropping tau l evt
6 |   | ih_evil: ∀ tau l oe,
7 |     tau oe = evil_label → eavesdropping tau l (inr oe).
```

The third relation of the threat model `synthesis` is defined in the Figure 7.9. There is an additional argument of type `messages` which is just a list of `event` (messages) under the control of an attacker. The relation `generate_event` (line 5) models the ability of an l-attacker that can generate any name in a name partition indexed by a label bounded above by `l`. This relation holds if all the free names in the event, such as domain names, URL parameters, names in the script and HTML document and cookie values, can be generated by the attacker at the label `l`. Moreover, the attacker may generate the free names of any network event previously intercepted or overheard, provided that the attacker can inspect

its payload. The attacker has the capability to generate any name communicated over compromised sessions.

Figure 7.9: `synthesis` relation

```
1  Inductive synthesis:
2    (output_event → label) → label → messages → event → Prop :=
3    | is_gen: ∀ tau l M ie,
4       event_label (inl ie) <= l →
5       generate_event (inl ie) (event_label (inl ie)) →
6       synthesis tau l M (inl ie)
7    | is_rep: ∀ tau l M evt,
8       List.In evt M →
9       event_label evt <= net_label ∧ net_label <= l →
10      synthesis tau l M evt.
```

The first rule `is_gen` models when an attacker can forge an input event. An l-attacker can forge an input event `ie`, provided that he can generate all the free names in `ie` and the event label of `ie` is bounded above by `l`. The latter condition ensures, for instance, that a net-level attacker cannot forge signed HTTPS traffic and that a web attacker `http_label d` cannot provide responses for another web server at `d'`. This rule also allows the attacker to send arbitrary output events to any server, provided that he is able to compose the request contents.

Finally, the rule `is_rep` allows an attacker with network capabilities (side-condition `net_label <= l`) to replay previously intercepted or overheard traffic. Since HTTPS ensures freshness, the side-condition `event_label evt <= net_label` similarly guarantees that encrypted traffic cannot be replayed.

7.5 Well Formed Traces

Proving integrity for a session with ill-formed input events is extremely challenging, therefore, the input events are restricted only to the well-formed input events. A well-formed input event does not contain names (e.g., cookie values) anywhere in

the input that can be guessed by an attacker. An attacker, however, is not forced to produce only well-formed inputs.

Well-formed input events are precisely defined by ensuring that the URL, headers and body of the input events do not contain any secret value, except the cookie values in the *Set-Cookie* headers which can only be guessed at the level of the `url_label`. This property of the contents of the input event is defined in Coq as a polymorphic inductive proposition `guessable` (Figure 7.10) which captures in the model the inability of an attacker to guess the random secrets like passwords and authentication cookie values. It states that the value of a type `A` can be guessed by an attacker at label `l`, precluding the existence of any secrete value. The type `A` is included as an implicit argument which can be instantiated with any other type of sort `Type`.

Figure 7.10: `guessable` relation

```
1 | Inductive guessable {A:Type}: A → label → Prop :=
2 |   ran: ∀ l u, guessable u l.
```

As observed by Akhawe et al. [7], the user behaviour needs to be constrained to ensure his or her password is not sent in clear as the URL parameter, otherwise, this will make most web security mechanisms ineffective. For this purpose, the well-formedness of the URLs is checked. A URL is well-formed if its `domain` is public and its `path` can be guessed by the attacker at or below `url_label`. The relation `wf_url` in Figure 7.11 defines well-formed URL. Well-formedness ensures that the URL carries no secret values (e.g., password) with it.

The body of the network response input contain in-line scripts and HTML document. JavaScript code may also be loaded as remote files from third-parties or from the same origin. In either case, they are checked to ensure they do not contain any secret values. The recursive function `wf_script` in Figure 7.12 checks

Figure 7.11: `wf_url` relation

```
Definition wf_url (u: url) : Prop:=
  guessable (url_domain u) bot_label ∧
  (∃ l, l <= url_label u → guessable (url_path u) l).
```

each parameter of each script constructor with at least one parameter. For the rest of scripts with no parameters (e.g., `null_script`), the proposition always holds (returns `True`). This seems abusing the use of `True` as in such scripts, there is no content and hence can not be guessed, however, the well-formed property ensures that the scripts do not contain any secret values which is always true for type constructors with no parameters. The same approach is adopted for constructors in other types as well such as `empty_file` in Figure 7.14.

Figure 7.12: `wf_script` relation

```
Fixpoint wf_script (e: script) (l: label) : Prop :=
 match e with
  | nat_script n ⇒ guessable n l
  | str_script z ⇒ guessable z l
  | url_script u ⇒ wf_url u
  | code_script e ⇒ wf_script e l
  | app_script e1 e2 ⇒ wf_script e1 l ∧ wf_script e2 l
  | var_script v ⇒ guessable v l
  | fun_script v vl e ⇒
     guessable v l ∧ wf_list vl l ∧ wf_script e l
  | eval_script e ⇒ wf_script e l
  | seq_script e1 e2 ⇒ wf_script e1 l ∧ wf_script e2 l
  | set_cookie_script z cfv ⇒ guessable z l ∧ guessable cfv l
  | auth_script z u ⇒ guessable z l ∧ guessable u l
  | set_var_script v e ⇒ guessable v l ∧ wf_script e l
  | xhr_script e1 e2 e3 ⇒
     wf_script e1 l ∧ wf_script e2 l ∧ wf_script e3 l
  | get_win_root_node_script e ⇒ wf_script e l
  | remove_node_script e ⇒ wf_script e l
  | insert_node_script e1 e2 e3 ⇒
     wf_script e1 l ∧ wf_script e2 l ∧ wf_script e3 l
  | _ ⇒ True
 end.
```

The network document response consists of HTML document, represented as list of document trees `list doc_tree`, with HTML elements for in-line and remote scripts, text boxes and the HTML * * elements. The relation `wf_doc_tree` in Figure 7.13 ensures that the body of the HTML document is well-formed, by checking each HTML element of the document. The constructor `div_doc` of the type `doc_tree` (Figure 3.8, line 5) has one parameter `list doc_tree` that recursively refers to itself. To check that the `doc_tree` in the input event is well-formed, two *mutually depended* proposition functions are required: functions that call each other. In programming languages, defining such functions is tricky as the function defined earlier has to make a call in its body to the function defined after it.

Figure 7.13: `wf_doc_tree` data type

```
1  Inductive wf_doc_tree : doc_tree → label → Prop :=
2    | wf_inl_script_doc: ∀ eio e l,
3       (guessable eio l ∧ wf_script e bot_label) →
4       wf_doc_tree (inl_script_doc eio e) l
5    | wf_rem_script_doc: ∀ eio u l, (guessable eio l ∧ wf_url u) →
6       wf_doc_tree (rem_script_doc eio u) l
7    | wf_textbox_doc: ∀ eio z l, (guessable eio l ∧ guessable z l) →
8       wf_doc_tree (textbox_doc eio z) l
9    | wf_div_doc: ∀ eio ldt l,
10      (guessable eio l ∧ wf_list_doc_tree ldt l) →
11      wf_doc_tree (div_doc eio ldt) l with
12        wf_list_doc_tree: list doc_tree → label → Prop :=
13         | wf_nil_doc_tree: ∀ l, wf_list_doc_tree nil l
14         | wf_cons_doc_tree: ∀ x tl l,
15            wf_doc_tree x l ∧ wf_list_doc_tree tl l →
16            wf_list_doc_tree (x::tl) l.
```

In Coq, a proposition relation can be defined as an inductive type. This solves the complexity of mutually dependent proposition functions by defining mutually depended inductive types: types that refer to each other. The type `wf_doc_tree` in Figure 7.13 refers to another inductive type `wf_list_doc_tree` (line 10), which in turn is referring to the parent type at line 15. The (induc-

tive) type `wf_list_doc_tree` (line 12) is defined internal to the inductive type `wf_doc_tree` using the `with` keyword. Both of these types can be separately referred to from inside other types (as in Figure 7.14).

The body of the network response event is represented using the type `file` (Figure 3.7): an inductive type with constructors for empty, scripts and HTML document files. The well-formed property of the `file` is checked in relation `wf_file` (Figure 7.14) by combining the well-formed properties of the `list doc_tree` (HTML file) and the `script` (script file), using the corresponding well-formed relations `wf_list_doc_tree` (at line 3) and `wf_script` (at line 4) respectively.

Figure 7.14: `wf_file` relation

```
1  Definition wf_file (f: file) (l: label) : Prop :=
2    match f with
3    | html_file ldt ⇒ wf_list_doc_tree ldt l
4    | script_file e ⇒ wf_script e l
5    | _ ⇒ True
6    end.
```

The headers and body of the network response event are represented by the type `resp` (Figure 3.5), which is a record type with fields for the set of cookie names to delete, *Set-Cookie* header representation, optional redirect URL and the body (script and HTML document) of the response. The relation `wf_resp` (Figure 7.15) defines the well-formed property of the `resp` type of the `input_event`. It uses an axillary function `wf_resp_cookies` that ensures that cookie values in the response can only be guessed by the domain that registered them (at `url_label` of the response URL). For the `path` of the redirect URL (if there is any), there exists a label at or below the response `url_label` at which it is guessable. The rest of the components of the response (scripts and HTML document) in the `file` are public and can be guessed at the `bot_label` – that is, they contain no secrete values.

Figure 7.15: `wf_resp` relation

```
Definition wf_resp (rs: resp) (u: url) : Prop :=
  wf_resp_cookies rs.(resp_set_cookies) u ∧
  guessable (url_domain u) bot_label ∧
  (∃ l', l' <= (url_label u) →
    guessable (urlo_path rs.(resp_redirect_uri)) l') ∧
  wf_file rs.(resp_file) bot_label.
```

Figure 7.16: `wf_input_event` relation

```
Inductive wf_input_event: input_event → Prop:=
  | wf_load_new_win: ∀ uwi u, wf_url u →
     wf_input_event (user_load_in_new_window_event uwi u)
  | wf_load_win: ∀ uwi u, wf_url u →
     wf_input_event (user_load_in_window_event uwi u)
  | wf_close_win: ∀ uwi,
     wf_input_event (user_close_window_event uwi)
  | wf_text: ∀ uwi k pwd l u,
     guessable pwd (text_input_label pwd u) →
     wf_input_event (user_input_text_event uwi k pwd l)
  | wf_doc_resp: ∀ nci uwi rs,
     wf_url nci.(net_conn_id_url) →
     guessable nci.(net_conn_id_value) bot_label →
     wf_urlo rs.(resp_redirect_uri) →
     wf_resp rs nci.(net_conn_id_url) →
     wf_input_event (network_document_response_event nci uwi rs)
  | wf_scrip_resp: ∀ nci rs,
     wf_url nci.(net_conn_id_url) →
     guessable nci.(net_conn_id_value) bot_label →
     wf_urlo rs.(resp_redirect_uri) →
     wf_resp rs nci.(net_conn_id_url) →
     wf_input_event (network_script_response_event nci rs)
  | wf_xhr_resp: ∀ nci rs,
     wf_url nci.(net_conn_id_url) →
     guessable nci.(net_conn_id_value) bot_label →
     wf_urlo rs.(resp_redirect_uri) →
     wf_resp rs nci.(net_conn_id_url) →
     wf_input_event (network_xhr_response_event nci rs).
```

After defining the well-formed propositions for each individual component of the `resp` and `url` types, what is needed now is to combine all of these relations in one relation `wf_input_event` as shown in Figure 7.16. This relation is defined as inductive type proposition with constructors one for each input event. The first two constructors (lines 2−5) correspond to user input navigating a URL in a new window or in an existing window. Both of these events are well-formed if the URL is well-formed – it ensures that the user never types in the address bar a URL containing a password (or an authentication cookie value) which should not be disclosed to the remote server. The user input closing a window (lines 6−7) does not contain any value entered by the user and hence is always well-formed.

The `wf_text` constructor at lines 8−10 rules out text inputs containing names conflicting with authentication cookie values. In other words, the assumption is that the user always enters either a password or some public data. The function `text_input_label` (line 9) models the internal password manager (Section 7.7) which gets two parameters: the password entered in a login form and the page source URL. It returns the `url_label` if the action URL of the login form matches with the corresponding action URL stored in the manager for the entered password, otherwise, `bot_label` is returned.

The rest of three constructors (lines 11−28) correspond to the last three network responses of the type `input_event` (Figure 3.1). They ensure that cookies set by an honest server are picked from the correct name partition and only occur in the standard HTTP header. Furthermore, it is required that confidential data never appear in the body of a response or in the cookie names. Enforcing session integrity whenever any of the previous constraints fails would require the browser to implement a full-fledged information flow control policy [38], so as to identify secret data inside a web page or in the address bar and prevent their leakage. Notice that it is not assumed that the intruder is forced to produce well-formed

inputs.

Figure 7.17: `wf_input_stream` function

```
Fixpoint wf_input_stream (lie: list input_event) : Prop :=
  match lie with
  | x::tl ⇒ wf_input_event x ∧ wf_input_stream tl
  | _ ⇒ True
  end.
```

The definition of well-formed input event in Figure 7.16 is extended to input streams using the function `wf_input_stream` defined in Figure 7.17. To leverage the security definitions, defined so far to enforce integrity, to stream of events, the data type `trace` (Figure 7.18) is defined, which is a pair of input and output event streams [19, 23]. The definitions of trace and well-formed input streams are combined together to define well-formed trace relation `wf_trace` as shown in Figure 7.19.

Figure 7.18: `trace` data type

```
Definition trace := (list input_event * list output_event).
```

Figure 7.19: `wf_trace` relation

```
Definition wf_trace (tr: trace) : Prop :=
  match tr with
  | (lie, loe) ⇒ wf_input_stream lie
  end.
```

7.6 Proof of Session Integrity

Theorem 3. *Flyweight Firefox Plus (FF$^+$) enforces session integrity for any well-formed trace (with respect to the threat model in Section 7.4).*

Proof. Complete proof of this theorem is given in the technical report [22][3]. □

The proof of this integrity result is very challenging. It draws on a label-indexed family of simulation relations, which connect the original trace and the attacked trace (original trace with attacker added messages). The nature of the simulation is non-standard, due to the significant differences which may arise between the two traces.

7.7 SessInt: Enforcing Session Integrity

SessInt is a proof-of-concept implementation of the integrity enforcement mechanisms in FF$^+$[4] as a browser extension for Google Chrome. The formal model FF$^+$ and the soundness proof of the integrity policy enforced in it, as discussed (by describing the mechanism enforced in EFF) in the previous sections, gives provably sound browser-based enforcement mechanisms of web session integrity. In this section, their enforcement into real web browser is discussed, however, their implementation as a browser extension is challenging due to the usability issues.

The design of SessInt is based on the same mechanisms as adopted in CookiExt, however, there are some notable additional features added to enforce the integrity policy enforced in FF$^+$. Similar to CookiExt, it redirects HTTP requests over HTTPS if the login operation is successful over HTTPS and fall-back to HTTP

[3]The proof of correctness of Theorem 3 has not been mechanized and is left to future work.

[4]The term EEFF is used to refer to the description of enforcement mechanism in Coq and FF$^+$ to the enforcement in Flyweight Firefox with proofs. As the enforcement mechanisms used in both are exactly the same, they can be used as synonyms when referring to the mechanism.

for the parts of the website that does not support HTTPS, such as in mixed-content websites. In this later case, the set of cookies is extended with those upgraded by the extension (with `Secure` flags set) to keep the session alive for usability reasons, compromising the security of such cookies. Following are the major additional features adopted in the design of SessInt.

7.7.1 Pages and Network Connections Stores

To thwart CSRF attacks, in particular those caused by cross-origin redirects, SessInt maintains stores for the pages and network connections similar to those as described in the formal model EFFF. The network connection is tainted if a cross-origin request is sent over it and the page loaded over tainted connection inherits the value of the connection taint. A conservative policy is then adopted for the requests sent over a tainted connection or from a tainted page. The essence of such policy will be highlighted in the next subsections when cookies and network requests are discussed.

7.7.2 User Clicks

When the user follows a link by clicking it, this action by the user is not trusted. The rationale here is that, such actions might have been performed by malicious JavaScript code injected on the page. Moreover, it is unrealistic to assume that the users carefully check the link and decide whether or not to follow based on the URL and its parameters. Therefore, to prevent cross-origin forgeries by malicious scripts, a more conservative policy is used and all authentication cookies are stripped before sending cross-origin requests.

As an example, suppose the user is signed into a website `www.example.com` and opens a vulnerable website `www.vulnerable.com` where the attacker is able to inject a link pointing to `www.example.com`. When the user clicks the link, a

cross-origin request is sent to `www.example.com` and, in the absence of SessInt, the browser will automatically attach all the cookies (including session cookies) with the request and will be successfully authenticated by the server. As the SessInt strips all the session cookies from such requests, the attack will fail.

7.7.3 Implicit Loads

Modern web applications load third-party contents such as JavaScripts, images and styles using HTML elements with the source *src* attributes set to the remote URLs. When the HTML document is loaded, these remote resources are *implicitly* loaded from third parties and the same origin. A similar attack as described above can be launched, for example, by including a <*script*> tag with *src* attribute set to the URL pointing to `www.example.com`. To counter such attacks, SessInt also strips cookies from cross-origin requests initiated by implicit loads.

Session cookies are stripped from all the (same or cross-origin) requests, originated through implicit loads or user clicks, from a tainted page. There is an exception when the request URL is in the white-list for HTTP communication.

7.7.4 Passwords

When the user receives a login page and enters the password in the login form in the page, the password becomes part of the DOM and any script on the page can access it and leak it to the attacker, for example by changing the action URL of the login form. To protect passwords, the login form is sand-boxed into an isolated pop-up window without scripts and an internal password manager is implemented by the SessInt. The password manager checks if the entered password is correct before sending it. When the user enters the password for the first time, the manager asks the user for confirmation that the password is being sent to an unknown URL. If the user confirms the operation, the manager stores the password

and is associated with the page and the action URL of the login form to enforce the runtime discipline adopted by FF$^+$.

Next time when the user enters the password for the same site, if the action URL is modified by the malicious script, it will be detected by the password manager and hence the password will not be sent. This corresponds to the function `text_input_label` in the formal model (EEFF) which returns `bot_label` in this case. The input is treated with low integrity and hence more restrictive policy is adopted and the password is not sent with the request. In case, the action URL of the login form where the password is entered matches with the one stored in the password manager, the password is sent only if the page is not tainted.

A user may use the same password for many websites, which is a bad but common practice by many users. In such cases, the password entered for one site can be sent by the script to the other website and it will be not detected by the password manager as the fabricated action URL is also associated with the same password. However, as the password is already known to the later website, it can not be considered as a failure of the SessInt protection mechanism.

An ideal way would be to check the action URL of the login form before the scripts are executed when the page is being loaded, however, the Chrome API does not allow to inspect the page content before in-line scripts are executed. However, if these scripts modify the action URL before the extension creates the sand-boxed form, the password manager will detect it by a comparison with the stored action URL and will warn the user before proceeding.

7.7.5 Cookies

A request received over the tainted connection is not trustworthy, hence cookies received with such requests are not stored. As in FF$^+$, cookies are only updated if they are received over an untainted network connection. Similar to Cook-

iExt, SessInt marks any authentication cookie received by the browser on HTTP connections as `HttpOnly`. Authentication cookies received over HTTPS, instead, are marked as both `HttpOnly` and `Secure`. This prevents leakage by malicious JavaScript programs and protects cookies in case HTTP links are injected into HTTPS websites.

To preserve functionality, SessInt forces a redirection on HTTPS for the entire website when a login form is submitted over HTTPS. Indeed, if the website contains some hard-coded HTTP links, marking some authentication cookies as `Secure` would break the session when navigating these links. As done by other extensions including CookiExt [21, 41, 91], authentication cookies are detected based on standard naming conventions (e.g., PHPSESSID) and a heuristic that measures the degree of entropy of the cookie value. In a recent paper by Calzavara et al. [25], the authors show how the authentication cookie detection process can be improved significantly using machine learning techniques.

For every cookie in the response header, the extension checks if the same cookie exists in the store and whether or not its `Secure` flag was set by the extension. Cookie originally with `Secure` flag set (its `Secure` flag is not set by SessInt but `HttpOnly` may be set by it) over HTTPS has high integrity and hence is not overwritten with the cookie received over HTTP (low integrity). If a `Secure` cookie gets expired, the browser automatically removes it and hence the new cookie will be added to the store.

A non-`Secure` cookie received over HTTPS could be assumed of low integrity than the corresponding cookie in the store with `Secure` flag set. However, this rule will break some scenarios where the website updates a `Secure` cookie with a non-`Secure` over HTTPS. For example, when the user logs into `www.yahoo.com`, a cookie Y is registered with `Secure` flag set. Later on, when the user tries to login again to `www.yahoo.com`, during the login phase, the response received

from the URL `https://login.yahoo.com/config/login` sets the same cookie with the same value over HTTPS but without `Secure` flag. If the new cookie is not added, it does not create any problem in this case (as the values are the same) and the user will be logged into `www.yahoo.com` successfully. However, sometimes unpredictably, the new cookie value received with the response from `https://login.yahoo.com/config/login` is different than the one already in the store, invalidating the existing cookie. If the existing cookie is not updated with the new one, the browser will attach the old one with the request while the website is expecting the new value and hence the login operation will fail.

As SessInt sets the flags anyway, the policy adopted in SessInt, therefore, considers two cookies of different flags of the same integrity. Hence a cookie with originally `Secure` flag set is replaced with a new cookie (possibly originally non-`Secure` but secured by SessInt) cookie regardless of the value. If the cookie is non-`Secure` and is received over HTTPS, it is protected during transmission, and as the SessInt sets its `Secure` flag, it is protected from network attackers by not including it with subsequent requests in clear.

7.7.6 Protection vs Usability

The security policy adopted in FF$^+$ is very strict and there are a few situations where it would break too many websites, hence the policy is slightly relaxed in SessInt. Some websites only support HTTPS for a subset of their pages. If some portions of the website do not provide support for HTTPS, as in CookiExt, SessInt selectively allows a fall-back to HTTP, with the proviso that cookies which have been previously promoted to `Secure` by the extension must be included to preserve the session. If an HTTPS connection times out, SessInt do not force fall-back, to prevent a network attacker intercepting HTTPS traffic from forcing SessInt into leaking over HTTP authentication cookies of websites normally providing HTTPS

support. In such cases, it cannot provide session security against network attacks for websites which only partially support HTTPS and the user is warned when this is the case.

Many websites redirect the browser to HTTPS when an HTTP access is requested and set the cookies at the browser by the redirect response over HTTP which are endorsed by including them with redirected request over HTTPS. However, SessInt would not include authentication cookies upon HTTPS redirection, since this redirect could as well be exploited by a network attacker to point the browser to a sensitive HTTPS URL and carry out a forgery. This cookie stripping breaks many websites, e.g., Facebook. To regain functionality, in the specific case of a protocol redirection with *unmodified* URL (when the URLs differ only by protocols), the user is asked (once for each site) to confirm that the redirection is expected, so that authentication cookies can be sent to the website. If the redirection looks suspicious, the user can block it.

It is common to find websites where HTTPS login forms are embedded (e.g., as iframes) into HTTP pages. This is insecure, as the attacker can change the HTTP page so as to redirect forms to a server that he controls, but since this is a very common practice, it is permitted and the SessInt warns the user when this happens. Additionally, the password manager will give an extra warning in case the password is going to be sent to a URL that is not yet known (as the modified action URL will not match with the one associated with the password stored in the password manager). The combination of these two warnings should make the user well aware of a possible attack.

The extension SessInt deals the event, typing a URL in the address bar and loading a page, with the highest integrity and hence the least restrictive policy is used. Network requests, triggered by typing in the address bar, are dealt with according to the normal browser behaviour – HTTP requests are not redirected

over HTTPS and cookies are included. Chrome API (by the time of this writing) does not allow to differentiate between a redirect, possibly caused by JavaScript and request triggered by the user typing in the address bar. As a temporary solution used in SessInt, the user has to add a special character 'g' before inserting the URL, which is captured by the Chrome omnibox API and detect that the URL has been typed in the address bar.

Secure sessions may be linked to sub-domains (e.g., `www.login.yahoo.com` and `www.yahoo.com`) or to external sites such as in e-payments systems (e.g., Paypal) or single sign on web applications. To avoid breaking websites, cookies are not stripped when moving into a sub-domain, though, this could sometimes be exploited by a web attacker with scripting capabilities in the sub-domain [20, 41].

The external sites scenarios are tricky to deal with, in particular, identifying legitimate and malicious cross-origin requests is challenging. Stripping cookies from cross-origin requests would simply break a number of web applications such as Paypal. To accommodate such applications, a simple idea might be to include a white list of trusted sites, e.g., for e-payments, that are needed to be reached by other websites, so that when the navigation comes back to the original site, authentication cookies are correctly sent and the session is preserved. Another option could be to use a heuristic-based approach as suggested by Philippe et al. [100] to differentiate between legitimate and malicious cross-origin requests and then strip cookies from the malicious requests. As this approach is based on heuristic, a more precise approach would be to get help from the server, such as through using Content Security Policy (CSP) headers. A proposal based on the later approach is given in Chapter 8.

The testing results of SessInt on existing vulnerable web applications, such as OWSAP Mutillidae [3] and Damn Vulnerable Web Application [2], confirms that even the relaxed security policy can prevent significant examples of attacks.

Consider the classic CSRF attack: a link to domain A on a page from a different domain E, that performs an action inside an active session with A. If the user is signed in to A and clicks the link to A on page from E (or opens a page from E which sends implicit request to A (Figure 8.3)), the attack will be launched. In the presence of SessInt, as the request is cross-origin, all the authentication cookies are stripped and the action will have no effect.

The policy adopted in SessInt, however, result compatibility issue with a number of web applications. Moreover, the enforcement of session integrity as access control model does not provide end-to-end security. In order to enforce integrity policies in a more precise and rigorous way and to make the collaborating web applications compatible, information-flow control model and content-security policy headers are considered in the next chapter.

8

Web Session Integrity: Information-Flow Control Enforcement

In the previous chapter, web session integrity was formalized and enforced using access control/tainting mechanism and as a proof-of-concept, a Chrome extension was designed and developed to enforce web session integrity at the client side. In this chapter, an information-flow control technique is designed to enforce web session integrity in a more permissive and fine-grained way than access control mechanisms. A prototype of the enforcement mechanism is implemented as an extension to the FlowFox web browser.

8.1 Session Protection at Different Layers

Web sessions can can be attacked at the network layer through network sniffing or man-in-the-middle attacks breaking the confidentiality or integrity of web sessions. This is a well-understood problem with well-understood solutions: by appropriate use of transport level security techniques such as encrypting the communication using TLS/SSL protocols, these attacks can be stopped. At the session implementation layer, script injection or again network level attacks can be used to

steal a session cookie and hijack the session or to impose a session cookie on a client already known to the attacker (called session fixation attack [41, 69]). Such attacks can be prevented by ensuring that session cookies are submitted only using TLS/SSL, prohibiting script access to session cookies (by setting the `Secure` and `HttpOnly` attributes on session cookies) and enforcing renewal of a session on authentication. The extensions CookiExt and SessInt as discussed in previous chapters are used to protect against attacks at the network and session implementation layer.

In addition to the attacks at the network and session implementation layer, web sessions can also be attacked at the application layer. Since cookies are attached to HTTP requests by the browser automatically, without any web application involvement, any page in the browser can send malicious requests (for example, by including *<script>* tags) to any of the servers that the browser currently has a session with, and that request will automatically get the session cookie attached and hence will be considered as part of a (possibly authenticated) session by the server. If the page sending the malicious request is from a different origin, such attacks are called CSRF attacks [10]. Malicious requests can also be sent by scripts included in or injected by an attacker into a page from the same origin. Since both inclusions of third-party scripts [89] and script injection vulnerabilities are common [68], these are important attack vectors.

The formal model given by Akhawe et al. [7] is an excellent definition for the purpose of studying CSRF attacks and countermeasures, but the underlying model does not have a sufficiently detailed representation of scripts to study other application-level session integrity issues. The formal notion of session integrity defined and enforced in Chapter 7 is browser-centric and amenable for client-side enforcement, however, the enforcement at the client side is based on access control/tainting mechanism. To enforce session integrity in a more permissive and

fine-grained way than access control, an information-flow technique needs to be designed to enforce session integrity. For this purpose, the session integrity definitions in Chapter 7 are refined to a classical noninterference property. The underlying assumption, however, is that appropriate defences against both network-level and cookie-level attacks are put in place.

8.2 Login History Dependent Noninterference: Definition and Enforcement

A common way to formalize integrity properties, such as the one given in Figure 1.2, is based on the concepts from information-flow security. One defines a partially ordered set of security labels that represent integrity levels (in the simplest case, two labels $\top$ and $\bot$ for high and low integrity, respectively). All inputs and outputs from the program under consideration (in this case, the browser) are labelled. Inputs are labelled $\top$ if they come from a trustworthy source and $\bot$ otherwise. Outputs, on the other hand, are labelled $\top$ if their integrity is important and $\bot$ otherwise. A program is information-flow secure (noninterferent) if low integrity inputs do not influence high integrity outputs (i.e. no information flows from low integrity sources to high integrity targets).

A complication in the case of web session integrity, however, is that both the set of integrity labels, as well as the labelling function, evolve over time as the user logs into more sites. The same message sent by site E to site A (for instance if the page from E sends a request to load a resource from A) will be of low integrity level if the browser is currently not logged into A, and it will be of a higher integrity level if the browser *is* logged into A. Exactly this kind of *login history-dependent noninterference* is formalized and then instantiated to the web context in the following sections.

As in Chapters 3, 4 and 7, the browser is modelled as a general notion of reactive systems (Section 2.1.5) and then the property of *login history-dependent reactive noninterference* and an enforcement mechanism for it, is introduced.

Assume given a set of web domains $\mathcal{D}$, and the set of input events $\mathcal{I}$ contains an event $\mathsf{login}(d)$ for all $d \in \mathcal{D}$. This event models a successful login of the browser into domain d. It is further assumed that the set of output events $\mathcal{O}$ contains an event $\bullet$ that represents a *silent* output, i.e. an internal computation step of the reactive system. A stream is defined by the coinductive interpretation of the grammar $S ::= [] \mid s :: S'$, where s ranges over individual stream elements. Unlike, Bohannon et al. who defines the behaviour of a reactive system in a state Q as a relation between input and output streams, the reactive behaviour, instead, is defined here as a relation between input streams and event streams where the later contain both input and output events, appropriately interleaved. This approach is needed, in particular, to handle the noninterference that is dependent on the login history.

Definition 3 (Reactive behaviour)**.** *A reactive system state Q generates the event stream S from the input stream I if the judgment $Q(I) \rightsquigarrow S$ holds, where this judgment is coinductively defined by:*

$$\frac{}{C([]) \rightsquigarrow []} \qquad \frac{C \xrightarrow{i} P \qquad P(I) \rightsquigarrow S}{C(i :: I) \rightsquigarrow i :: S} \qquad \frac{P \xrightarrow{o} Q \qquad Q(I) \rightsquigarrow S}{P(I) \rightsquigarrow o :: S}$$

8.2.1 Login History Dependent Noninterference

The lattice of possible integrity levels $\mathcal{L}$ has elements $\top$ (highest integrity), $\bot$ (lowest integrity) and d for all $d \in \mathcal{D}$ (integrity level of authenticated communication with domain d). Since higher integrity information can flow to lower integrity

levels but not vice-versa, the ordering relation on $\mathcal{L}$ is defined as $\top \leq d \leq \bot$, and for different d and d', d and d' are incomparable.

The key idea of login history dependent noninterference (LHDNI) is to make the labelling function that assigns integrity levels to events dependent on the login events that have occurred. Initially, all network events are low integrity ($\bot$), but after a $\mathsf{login}(d)$ event, network communication with d will have level d. This models the behaviour of a web browser: because of the automatic attaching of cookies (including the session cookie), the integrity of network communication to domain d becomes more important after a login to d. It also models the assumption that the server will be more careful with HTTP(S) responses for authenticated sessions (integrity level of these responses is higher).

The login history is represented as a finite sub-lattice L of $\mathcal{L}$, where L is initially $\{\top, \bot\}$, and L evolves with inputs processed as follows (where $L \oplus d$ is written when L is extended with element d):

$$\frac{i = \mathsf{login}(d)}{L \xrightarrow{i} L \oplus d}\ (\tau\text{-LOGIN}) \qquad \frac{i \neq \mathsf{login}(d)}{L \xrightarrow{i} L}\ (\tau\text{-NIL})$$

In words, whenever the user logs into a domain d, the label d is added to the set of integrity labels L.

The function $lbl_L(e) : \mathcal{I} \uplus \mathcal{O} \to \mathcal{L}$, that labels events, depends on the login history L. The intuition is that interactions that belong to a session with a domain d will get label d iff $d \in L$, otherwise, they get label $\bot$ (i.e. once logged into d, the user cares about the integrity of messages to d). It is stipulated that $lbl_L(\mathsf{login}(d)) = d$ for any d. The notation $L^{|l}$ is used for the list of labels $l' \in L$ such that $l \leq l'$ and $L_{|l}$ for the list of labels $l' \in L$ such that $l' \leq l$. For an input

i, for simplicity, $L^{|lbl_L(i)}$ is written as just $L^{|i}$.

The LHDNI is defined in terms of the relation *LHD-similarity*, which defines when two streams look the same to an observer at level l while taking the login history into consideration.

Definition 4 (LHD-similarity)**.** *Under login history L, two streams S and S' are LHD-similar at level l (l-similar) if the judgement $L \vdash S \approx_l S'$ holds, where this judgement is coinductively defined by:*

$$\frac{}{L \vdash [] \approx_l []}\ (\text{LHD-NIL}) \qquad \frac{s = \mathsf{login}(d) \qquad d \leq l \qquad L \oplus d \vdash S \approx_l S'}{L \vdash s :: S \approx_l s :: S'}\ (\text{LHD-LOGIN})$$

$$\frac{s \neq \mathsf{login}(d) \qquad lbl_L(s) \leq l \qquad L \vdash S \approx_l S'}{L \vdash s :: S \approx_l s :: S'}\ (\text{LHD-SIM}) \qquad \frac{lbl_L(s) \not\leq l \qquad L \vdash S \approx_l S'}{L \vdash s :: S \approx_l S'}\ (\text{LHD-L})$$

$$\frac{lbl_L(s) \not\leq l \qquad L \vdash S \approx_l S'}{L \vdash S \approx_l s :: S'}\ (\text{LHD-R})$$

Now, a state is LHDNI if l-similar inputs lead to l-similar outputs. Formally, this is defined as the following:

Definition 5 (LHDNI)**.** *A state Q of a reactive system is LHDNI if $Q(I) \rightsquigarrow S$ and $Q(I') \rightsquigarrow S'$ imply that $\forall l \in \mathcal{L}, \emptyset \vdash I \approx_l I' \Rightarrow \emptyset \vdash S \approx_l S'$.*

Notice that it is important that S and S', the event streams that contain interleaved input and output events, are compared because of the history dependence of the definition of LHD-similarity. If only the output events were considered, as classic noninterference definitions do, then there would be no login event present

in the output streams; but login events need to be kept there as they influence the labelling function.

8.2.2 Enforcement

An enforcement mechanism based on secure multi-execution [44, 15, 96] is built to enforce the security policy defined in terms of LHDNI. The basic idea is to construct a new reactive system that is a *wrapper* [15] around multiple copies (sub-executions) of the original reactive system, one for each level in the login history L. When the wrapper consumes an input event, it is passed to the copies at or higher than the level of the input. When a sub-execution produces an output, if its level matches the level of the execution, the output is produced by the wrapper, otherwise it is suppressed.

A state of the wrapper is a triple (L, R, L_q), where

- L is the login history,
- R is a function mapping security labels in L to states, i.e. $R(l)$ is the sub-execution at level l, and
- L_q is a waiting queue of levels that still need to process the last input consumed. It is initially empty and when an input is consumed it is set to all levels that should process this input. These labels are ordered from low integrity to high integrity such that the sub-execution at the low integrity (label $\bot$) is always executed first.

States $(L, R, [])$ are consumer states and states (L, R, L_q) with $L_q \neq []$ are producer states. The initial state of the wrapper is a state $(\{\top, \bot\}, R, [])$ with $R(\top)$ and $R(\bot)$ being the initial state of the original reactive system.

The semantics is shown in Figure 8.1. The main extension with respect to standard SME for reactive systems [15] is the way in which login events are handled:

Figure 8.1: Basic semantics for secure multi-execution of a reactive system

$$\text{(LOGIN)}\quad \frac{i = \mathsf{login}(d) \qquad d \notin L \qquad L' = L \oplus d \qquad L_q = L'^{|d} \qquad R(l) \xrightarrow{i} P_l \qquad R'(d) = P_\top \qquad R'(l) = P_l \text{ for } l \in L_q \setminus \{d\} \qquad R'(l) = R(l) \text{ for } l \notin L_q}{(L, R, []) \xrightarrow{i} (L', R', L_q)}$$

$$\text{(LOAD)}\quad \frac{i \neq \mathsf{login}(d) \vee (i = \mathsf{login}(d) \text{ with } d \in L) \qquad R(l) \xrightarrow{i} P_l \qquad L_q = L^{|i} \qquad R'(l) = P_l \text{ for } l \in L_q \qquad R'(l) = R(l) \text{ for } l \notin L_q}{(L, R, []) \xrightarrow{i} (L, R', L_q)}$$

$$\text{(OUT-P)}\quad \frac{R(l) \xrightarrow{o} P \qquad lbl_L(o) = l}{(L, R, l :: L_q) \xrightarrow{o} (L, R[l \mapsto P], l :: L_q)}$$

$$\text{(OUT-C)}\quad \frac{R(l) \xrightarrow{o} C \qquad lbl_L(o) = l}{(L, R, l :: L_q) \xrightarrow{o} (L, R[l \mapsto C], L_q)}$$

$$\text{(DROP-P)}\quad \frac{R(l) \xrightarrow{o} P \qquad lbl_L(o) \neq l}{(L, R, l :: L_q) \xrightarrow{\bullet} (L, R[l \mapsto P], l :: L_q)}$$

$$\text{(DROP-C)}\quad \frac{R(l) \xrightarrow{o} C \qquad lbl_L(o) \neq l}{(L, R, l :: L_q) \xrightarrow{\bullet} (L, R[l \mapsto C], L_q)}$$

these update the login history L, and hence also the number of sub-executions in the wrapper, and (implicitly) the labelling function lbl_L. Note how the newly created sub-execution at level d is initialized: $P_\top$ is the resulting state after giving i to $R(\top)$, i.e. essentially the sub-execution at level $\top$ is cloned and the event i is fed to it. This is the right thing to do as the newly created sub-execution must have seen all the events of higher integrity than d. The [LOAD] rule handles other input events than initial login events. It essentially feeds the input to all sub-executions with a level $\leq lbl_L(i)$, by updating the appropriate sub-executions in R to a state where they have received i and by setting the waiting queue to contain all levels that have to process this input event. The other four rules implement the SME output rules, making sure that output of level l is only performed by

the execution at level l. They also make sure that, as sub-executions return to a producer state, the next sub-execution in the waiting queue gets a chance to run.

These rules effectively block *all* cross-origin requests to authenticated domains. For instance, if a page received from an unauthenticated domain (a $\perp$ event) loads an image from an authenticated domain d, the corresponding HTTP(S) request (a d-level event) will be suppressed (by either of the rules [DROP-C] or [DROP-P]). However, substantially better can be achieved: instead of dropping such requests, the session cookie is stripped from the request as in other client-side CSRF protection systems [23, 100]. Assume a function $strip^L(o)$ exists, that for any o with $lbl_L(o) = d$ (for some d) strips the session cookies from o, and for all other o returns o.

To erase session cookies from the requests, the projection functions π_l^L is defined as follows:

$$\pi_l^L(o) = \begin{cases} strip^L(o) & \text{if } l = \perp \\ o & \text{otherwise} \end{cases}$$

The assumption is that the event labelling function lbl_L checks for the presence of authentication cookies to deem a network output as a high integrity event. As the projected output event does not include any authentication cookie, it is deemed of low integrity, hence $lbl_L(strip^L(o))$ is always $\perp$.

The basic semantics (Figure 8.1) released an output o from a sub-execution at label l only if $lbl_L(o) = l$. This concept is now generalized: a sub-execution at label l can release $\pi_l^L(o)$ if the following predicate holds:

$$release_{L,l,L_q}(o) \quad = \quad lbl_L(o) = l \vee (l = \perp \wedge lbl_L(o) \notin L_q)$$

That is, an output is *released* from a sub-execution if its label matches the label l of the sub-execution, or when $l = \perp$ and there is no sub-execution at the

Figure 8.2: Semantics for secure multi-execution of a reactive system (updated)

$$\frac{R(l) \xrightarrow{o} P \qquad release_{L,l,L_q}(o)}{(L, R, l :: L_q) \xrightarrow{\pi_l^L(o)} (L, R[l \mapsto P], l :: L_q)} \text{(OUT-P)}$$

$$\frac{R(l) \xrightarrow{o} C \qquad release_{L,l,L_q}(o)}{(L, R, l :: L_q) \xrightarrow{\pi_l^L(o)} (L, R[l \mapsto C], L_q)} \text{(OUT-C)}$$

$$\frac{R(l) \xrightarrow{o} P \qquad \neg release_{L,l,L_q}(o)}{(L, R, l :: L_q) \xrightarrow{\bullet} (L, R[l \mapsto P], l :: L_q)} \text{(DROP-P)}$$

$$\frac{R(l) \xrightarrow{o} C \qquad \neg release_{L,l,L_q}(o)}{(L, R, l :: L_q) \xrightarrow{\bullet} (L, R[l \mapsto C], L_q)} \text{(DROP-C)}$$

label of the output in the waiting queue. Since the sub-executions are processed in the order from low integrity to high integrity, this means that this output is being sent in response to an input that was *not* of level $lbl_L(o)$, and hence is a cross-domain request to an authenticated domain. Starting processing sub-executions from the one at the low integrity (at $\bot$ label) implies all the cross-origin requests (requests with no sub-execution in the queue to release them) are released from the execution at $\bot$ label. As such requests are projected, hence session cookies are erased[1]. The updated rules are shown in Figure 8.2.

If the predicate $release_{L,l,L_q}(o)$ does not hold, then the output is *suppressed* (not released by the wrapper) from the sub-execution at level l. An output is suppressed from a sub-execution if its label l does not match the domain of the output. There is an exception to this rule: the output from sub-execution at $\bot$ level is suppressed only if there exists a sub-execution at the label of the domain of the output. That is, the output is suppressed if the following predicate holds:

$$suppress_{L,l,L_q}(o) \qquad = \qquad lbl_L(o) \neq l \wedge (l \neq \bot \vee lbl_L(o) \in L_q)$$

These two predicates are mutually exclusive, that is $\neg release_{L,l,L_q}(o) \iff suppress_{L,l,L_q}(o)$. Let P stands for the proposition $lbl_L(o) = l$, Q for $l = \bot$

[1]This will make a number of web applications incompatible, an issue addressed in Section 8.4.

and R for $lbl_L(o) \in L_q$, then the predicate $release_{L,l,L_q}(o)$ can be rewritten as $P \vee (Q \wedge \neg R)$. The implication $\neg release_{L,l,L_q}(o) \Rightarrow suppress_{L,l,L_q}(o)$ can be proved as the following:

$$\begin{aligned}
\neg release_{L,l,L_q}(o) &\Rightarrow \neg(lbl_L(o) = l \vee (l = \bot \wedge lbl_L(o) \notin L_q)) \\
&\Rightarrow \neg(P \vee (Q \wedge \neg R)) \\
&\Rightarrow \neg P \wedge \neg(Q \wedge \neg R) \quad (\textit{DeMorgan's Law } (\textit{negating OR})) \\
&\Rightarrow \neg P \wedge (\neg Q \vee R) \quad (\textit{DeMorgan's Law } (\textit{negating AND})) \\
&\Rightarrow lbl_L(o) \neq l \wedge (l \neq \bot \vee lbl_L(o) \in L_q) \\
&\Rightarrow suppress_{L,l,L_q}(o)
\end{aligned}$$

8.2.3 Security

The enforcement mechanism defined above provides session integrity, however, this claim has not been proved yet. It is now necessary to prove that this enforcement mechanism guarantees LHDNI.

Theorem 4 (Security)**.** *All the initial states of the wrapper are LHDNI.*

Proof. The proof of this theorem is given in the Appendix **??**. □

The theorem is proved using Bohannon's ID-bisimulation proof technique [19]. It suffices to prove that there exists an ID-bisimulation $\approx_l$ such that for every state of the wrapper (L, R, L_q), we have $(L, R, L_q) \approx_l (L, R, L_q)$. The proof of security consists of two steps: first the relation $\approx_l$ is defined and then it is shown that this relation is indeed an ID-bisimulation relation. Note the overloading of the $\approx_l$ notation. When used between streams, it is interpreted as LHD-similarity (Definition 4), when used between reactive system states, it refers to the definition below.

Definition 6 (*l-similarity* relation $\approx_l$)**.** *The state (L_1, R_1, L_{q1}) is l-similar to the state (L_2, R_2, L_{q2}) (written $(L_1, R_1, L_{q1}) \approx_l (L_2, R_2, L_{q2})$) iff:*

- $L_{1|l} = L_{2|l}$, *and*
- $R_1 \approx_l R_2$, *meaning* $\forall l' \leq l$: $R_1(l') = R_2(l')$, *and*
- $L_{q1|l} = L_{q2|l}$.

Lemma 1. *The l-similarity relation is an ID-bisimulation relation.*

Proof. The proof of this lemma is given in the Appendix **??**. □

8.3 Instantiation to Web Session Integrity

In this section, it is shown by example how LHDNI protects browsers from typical attacks on session integrity. Recall that the best practices for session security (i.e. the use of TLS/SSL and the use of the `Secure` and `HttpOnly` attributes on session cookies) are assumed in place. Assume the login events are recognizable by the browser: they are triggered for instance by a bookmarklet or password manager [23] and the response page of the site that one is logging into is shown in a separate top-level frame (tab) in the browser. The browser should enforce that logins to these known and trusted domains *must* happen through these bookmarklets, to avoid attacks such as login CSRF [10].

The examples show how remaining attacks such as classic CSRF and malicious script inclusion are countered by the enforcement mechanism defined in Section 8.2.2. A similar example can be constructed for client-side or reflected XSS.

Applying the enforcement mechanism described by the semantics in Figure 8.2 to web browsers requires to define the sets of input and output events for a browser. The events are limited to a simple set that can model the attacks considered in this dissertation. These events are described in Table 8.3 (the first four events are input events and the last four are output events). The table also shows the value of the security label assigned by the lbl_L function. All these events are standard

browser events and easy to recognize by the browser (for the $\mathsf{login}(d)$, recall the assumptions made above).

Table 8.1: User actions, input/output events and their labels

User actions	I/O events	lbl_L	
		$d \in L$	$d \notin L$
typing URL to domain d in the address bar	$\mathsf{ui\_load}(d)$	$\top$	$\top$
network response from domain d with header h	$\mathsf{net\_resp}(d, h)$	d	$\bot$
clicking link on the page from domain d	$\mathsf{ui\_link\_click}(d)$	d	$\bot$
entering password on the page from domain d	$\mathsf{login}(d)$	d	d
network request to domain d (including cookie)	$\mathsf{net\_req}(d)$	d	$\bot$
network request to domain d (no cookie)	$\mathsf{net\_req}(d)$	$\bot$	$\bot$
loading a page at the screen	$\mathsf{ui\_page\_loaded}$	$\bot$	$\bot$
dummy	$\bullet$	$\bot$	$\bot$

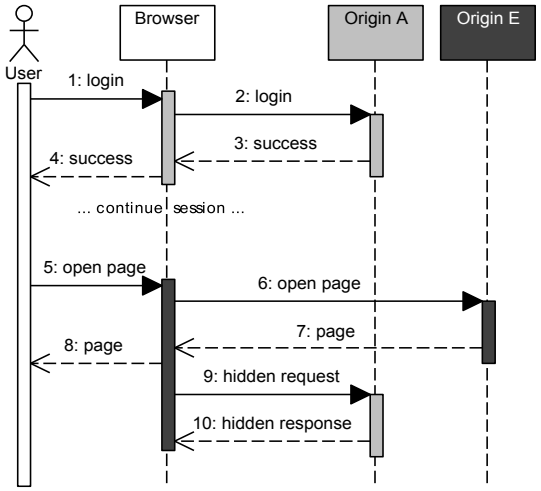
8.3.1 CSRF

Figure 8.3 gives a schematic overview of a classic CSRF attack[2]. The user signs into website A (messages 1−4) and opens a page in another tab from malicious website E (messages 5−8), which implicitly sends a cross-origin request to load remote content (e.g., an image) from A (message 9). As the browser will attach all the cookies with this request to A, it will lead to a CSRF attack on A.

Figure 8.4 shows an encoding of this attack in the browser model and shows how the enforcement mechanism stops the attack. Each line of the encoding is of the form $(E, [Rule]) : (L, R, []) \xrightarrow{n} (L', R', L_q)$, where E is the input or output event, *Rule* is the semantics rule (Figure 8.2), (L, R, L_q) represents the state of the wrapper and n is the message number in the corresponding interaction diagram figure. Outputs are shown slightly indented, so that it is easy to see by which input event they are caused. To simplify, L_0 is written for the set $\{\bot, \top\}$ and L_A for the

[2]It is exactly the same figure 1.2 but re-sketched here.

Figure 8.3: Classic CSRF



set $\{\bot, A, \top\}$. For simplicity, the finite list $l_1 :: l_2 :: []$ is denoted with $l_1 :: l_2$. If a specific component of the browser state is not taken care of, _ is instead used.

Events and semantics rules corresponding to each event in Figure 8.3 are shown in Figure 8.4. In this scenario, using a standard web browser, the attack would happen in message 9, where the request to A (initiated in response from E) would include cookies. However, under the wrapper, the attack is prevented. Specifically, the basic semantics in Figure 8.1 would drop the request, since a low integrity sub-execution is not allowed to send A-labelled requests; the updated semantics in Figure 8.2, instead, would strip the cookies from the request for the very same reason. Both options are secure, but the second option will break less existing websites.

8.3.2 Malicious Script Inclusion

Figure 8.5 gives a schematic overview of a script inclusion attack. The user signs into the website A (messages 1−4) and then opens a page (messages 5−6). This

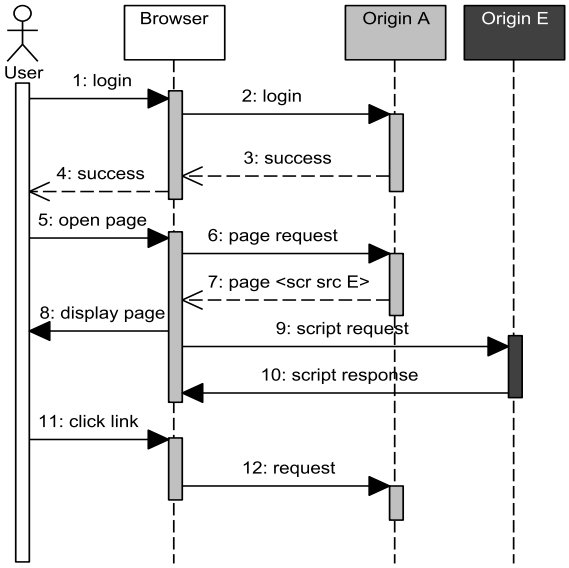
Figure 8.4: Classic CSRF attack encoding and prevention

1. ($\mathsf{login}(A)$, [LOGIN]): $(L_0, \_, []) \xrightarrow{1} (L_A, \_, \bot :: A)$
   2. suppress ($\mathsf{net\_req}(A)$, [DROP-C]): $(L_A, \_, \bot :: A) \xrightarrow{\bullet} (L_A, \_, A)$
   2. release ($\mathsf{net\_req}(A)$, [OUT-C]): $(L_A, \_, A) \xrightarrow{2} (L_A, \_, [])$
3. ($\mathsf{net\_resp}(A, h)$, [LOAD]): $(L_A, \_, []) \xrightarrow{3} (L_A, \_, \bot :: A)$
   4. release ($\mathsf{ui\_page\_loaded}$, [OUT-C]): $(L_A, \_, \bot :: A) \xrightarrow{4} (L_A, \_, A)$
   4. suppress ($\mathsf{ui\_page\_loaded}$, [DROP-C]): $(L_A, \_, A) \xrightarrow{\bullet} (L_A, \_, [])$
5. ($\mathsf{ui\_load}(u_E)$, [LOAD]): $(L_A, \_, []) \xrightarrow{5} (L_A, \_, \bot :: A :: \top)$
   6. release ($\mathsf{net\_req}(E)$, [OUT-C]): $(L_A, \_, \bot :: A :: \top) \xrightarrow{6} (L_A, \_, A :: \top)$
   6. suppress ($\mathsf{net\_req}(E)$, [DROP-C]): $(L_A, \_, A :: \top) \xrightarrow{\bullet} (L_A, \_, \top)$
   6. suppress ($\mathsf{net\_req}(E)$, [DROP-C]): $(L_A, \_, \top) \xrightarrow{\bullet} (L_A, \_, [])$
7. ($\mathsf{net\_resp}(E, h)$, [LOAD]): $(L_A, \_, []) \xrightarrow{7} (L_A, \_, \bot)$
   8. release ($\mathsf{ui\_page\_loaded}$, [OUT-P]): $(L_A, \_, \bot) \xrightarrow{8} (L_A, \_, \bot)$
   9. release without cookies ($\mathsf{net\_req}(A)$, [OUT-C]): $(L_A, \_, \bot) \xrightarrow{9} (L_A, \_, [])$

page includes a script tag that will include a third party script from E. When the page from A is being rendered (messages 7−8), the remote script is loaded from the website E (messages 9−10). The script can then, for instance, install an event handler that will trigger an (authenticated) request to A at a later time.

This example is encoded in Figure 8.6. The response input from E (message 10) gets a $\bot$ label, hence is fed only into the low integrity sub-execution. All the requests to A initiated by the user (in the context of A) or directly by input from A are released from the sub-execution at level A and hence are not affected by the script injected to the sub-execution at $\bot$. Requests released from the $\bot$ sub-execution, on the other hand, may be affected but as those outputs do not include cookies, they are safe. In the example, the request to A (message 12) as the result of the user input (message 11) is released from the execution at $\bot$, (release message 12). The sub-execution at A label never received the script from E, so it will not react to the link click and just output a silent event ($\bullet$).

Figure 8.5: Script inclusion attack



8.4 Extensions

The enforcement mechanism described by the formal semantics in Figure 8.2 enforces security policies to protect against attacks on session integrity, but by doing so it does break some common web scenarios that technically violate session integrity, but do so without malicious purposes. These scenarios can be handled in the approach adopted (Section 8.2) (by means of *endorsement* (the integrity variant of *declassification* [96, 109]).

Endorsements will typically have to be declared by the website that the browser has an authenticated session with. In the two approaches below, these declarations are done by means of request headers, similar to how Content Security Policy (CSP) [105] policies are communicated to the browser.

Figure 8.6: Script inclusion attack encoding and prevention

1. (login(A), [LOGIN]): $(L_0, \_, []) \xrightarrow{1} (L_A, \_, \bot :: A)$
   2. suppress (net_req(A), [DROP-C]): $(L_A, \_, \bot :: A) \xrightarrow{\bullet} (L_A, \_, A)$
   2. release (net_req(A), [OUT-C]): $(L_A, \_, A) \xrightarrow{2} (L_A, \_, [])$
3. (net_resp(A, h), [LOAD]): $(L_A, \_, []) \xrightarrow{3} (L_A, \_, \bot :: A)$
   4. release (ui_page_loaded, [OUT-C]): $(L_A, \_, \bot :: A) \xrightarrow{4} (L_A, \_, A)$
   4. suppress (ui_page_loaded, [DROP-C]): $(L_A, \_, A) \xrightarrow{\bullet} (L_A, \_, [])$
5. (ui_link_click(A), [LOAD]): $(L_A, \_, []) \xrightarrow{5} (L_A, \_, \bot :: A)$
   6. suppress (net_req(A), [DROP-C]): $(L_A, \_, \bot :: A) \xrightarrow{\bullet} (L_A, \_, A)$
   6. release (net_req(A), [OUT-C]): $(L_A, \_, A) \xrightarrow{6} (L_A, \_, [])$
7. (net_resp(A, h), [LOAD]): $(L_A, \_, []) \xrightarrow{7} (L_A, \_, \bot :: A)$
   8. release (ui_page_loaded, [OUT-P]): $(L_A, \_, \bot :: A) \xrightarrow{8} (L_A, \_, \bot :: A)$
   9. release (net_req(E), [OUT-C]): $(L_A, \_, \bot :: A) \xrightarrow{9} (L_A, \_, A)$
   8. suppress (ui_page_loaded, [DROP-P]): $(L_A, \_, A) \xrightarrow{\bullet} (L_A, \_, A)$
   9. suppress (net_req(E), [DROP-C]): $(L_A, \_, A) \xrightarrow{\bullet} (L_A, \_, [])$
10. (net_resp(E, h), [LOAD]): $(L_A, \_, []) \xrightarrow{10} (L_A, \_, \bot)$
    release ($\bullet$, [DROP-C]): $(L_A, \_, \bot) \xrightarrow{\cdot} (L_A, \_, [])$
11. (ui_link_click(A), [LOAD]): $(L_A, \_, []) \xrightarrow{11} (L_A, \_, \bot :: A)$
    12. suppress (net_req(A), [DROP-C]): $(L_A, \_, \bot :: A) \xrightarrow{\bullet} (L_A, \_, A)$
    12. release ($\bullet$, [OUT-C]): $(L_A, \_, A) \xrightarrow{\bullet} (L_A, \_, [])$

8.4.1 Endorsing Script Inclusions

A first, simple and common kind of endorsement is for script inclusion. The script inclusion example in Figure 8.5 is commonly *not* an attack: website A includes the script from E intentionally and trusts it to influence the session. While some scripts can be usefully included without having the possibility to influence the session (e.g., analytics scripts), inclusion of other scripts is only useful when these scripts have the right to influence the session (e.g., the jQuery library).

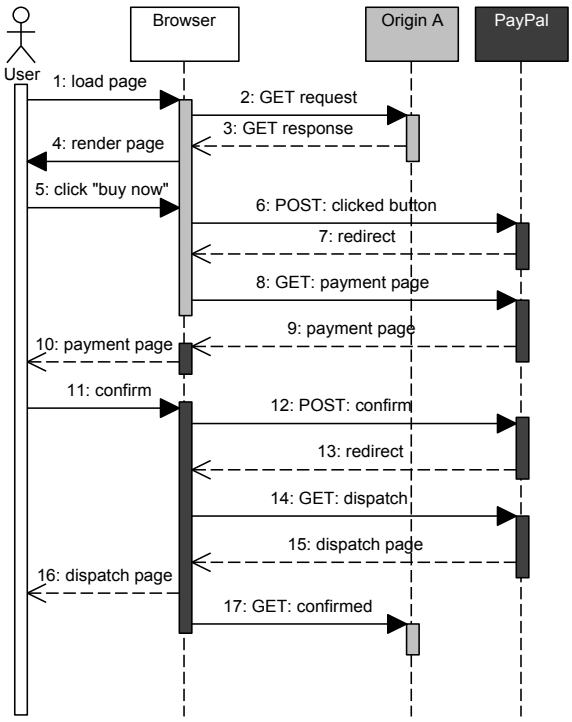
Fortunately, endorsing script inclusions is straightforward. The server A declares in an HTTP header which origins can provide trusted scripts and the browser uses this information to label outgoing and incoming requests to these white-listed origins from A's pages as being of level A. One could even argue that this should be the default interpretation of the CSP policy directives that allow script inclusions (e.g., the `script-src` directive).

8.4.2 Endorsements for Collaborating Applications

Endorsements are also required for collaborating web applications such as e-payment systems (e.g., Paypal). Consider, for example, a user who wants to buy an airline ticket at website A and pay via `www.paypal.com` (Figure 8.7).

In this example, the user opens a page from website A where he clicks the *buy* button and then confirms the payment on the `paypal.com` website. Messages 3−4 and 15−17 of Figure 8.7 are encoded in Figure 8.8 using the semantics. Assume the user is logged into both A and P (PayPal), i.e. L contains both A and P. The message 17 (*GET: confirmed*) at label A is a cross-origin request to A in response to the input at label E from website E and hence the wrapper will release it from the execution at $\bot$ (as there is no sub-execution in the queue at label A). As all the session cookies are erased, the payment operation will fail.

Figure 8.7: E-payment scenario [100]



To support such collaborating web applications, endorsement is needed. For these cases, the use of a response header is proposed which is used by the website to specify allowed entry points from different origins. A website s (source of white-list) sends a list of URLs url pointing to s specifying that another site w (white-listed site) is allowed to send cross-origin requests to these URLs, by setting a *connect-destination (cd)* header $<cd: \{W{:}w, U{:}url\}>$ in the response.

The wrapper will keep track of these headers by updating a set ω of key-value pairs of the form (w, url), where w is the white-listed website (the *who* part) and url is the list of URLs (the *how* part) specified as the allowed entry points white-listed for the w. The list of URLs url can also include URLs with wildcard character $*$ such as $s.com/*$, where the website w can send cross-origin (authenticated) requests to any URL of the site $s.com$.

Figure 8.8: E-payment application encoding

3. (net_resp(A, h), [LOAD]): $(L, \_, []) \xrightarrow{3} (L, \_, \bot :: A)$
   4. release (ui_page_loaded, [OUT-C]): $(L, \_, \bot :: A) \xrightarrow{4} (L, \_, A)$
   4. suppress (ui_page_loaded, [DROP-C]): $(L, \_, A) \xrightarrow{\bullet} (L, \_, [])$

...
(user clicks "buy" button, and confirms payment)
...

15. (net_resp(P, h), [LOAD]): $(L, R, []) \xrightarrow{15} (L, \_, \bot :: P)$
    16. release (ui_page_loaded, [OUT-P]): $(L, \_, \bot :: P) \xrightarrow{16} (L, R, \bot :: P)$
    17. release w/o cookies (net_req(u_A), [OUT-C]): $(L, \_, \bot :: P) \xrightarrow{17} (L, R, P)$
    16. suppress (ui_page_loaded, [DROP-P]): $(L, \_, P) \xrightarrow{\bullet} (L, \_, P)$
    17. suppress (net_req(u_A), [DROP-C]): $(L, \_, P) \xrightarrow{\bullet} (L, \_, [])$

As a simple example, assume two websites A and B send the endorsement headers $<$*cd:* $\{$*W:P, U:*$[a.com/*]$\}>$ and $<$*cd:* $\{$*W:P, U:*$[b.com/u1, b.com/u2]\}>$ in their responses. Initially, when the response from A is received, the wrapper will store in ω an entry $(P, [a.com/*])$ and when the other response from B is received, it will add the two URLs to the value bound to P, hence ω will become $(P, [a.com/*, b.com/u1, b.com/u2])$. The URL $a.com/*$ represents all the URLs of website A.

Now all the required information exists to decide if a cross-origin request should be endorsed. After receiving the example headers above, an output from P to any URL of A or to any of the two URLs $b.com/u1$ and $b.com/u2$ of website B should include cookies. On receipt of an input event i with label d, the wrapper will compute the set of URLs that d is allowed to send cross-origin requests to by looking it up in ω. Let us call the resulting set U_i.

The *release* predicate is generalized so that it takes U_i into account. An output is *released* from a state if 1) its label matches the label l of the current sub-execution or, 2) when $l = \bot$ and there is no sub-execution at the level of the output and

the request URL is not white-listed, or 3) when $l \neq \bot$ and the request URL is white-listed. The predicate $release_{L,l,L_q}(o, u, U_i)$ is defined as follows:

$$l = lbl_L(o) \vee (l = \bot \wedge lbl_L(o) \notin L_q \wedge u \notin U_i) \vee (l \neq \bot \wedge u \in U_i).$$

It is now shown how the PayPal example (Figure 8.7) works by encoding (Figure 8.9) using the model. (The ω and U_i is shown as the third and fourth component of the tuple representing the extended browser state.)

Assume the site A sends the header $<$*cd:* $\{$*W:P, U:*$[a.com/*]$\}>$ in the response input (Figure 8.7, message 3) and the wrapper creates the entry $\omega = (P, [a.com/*])$. Later on, when the input in message 15 is received from P, the corresponding list of URLs for P is retrieved, that is, $U_i = [a.com/*]$. The encoding in Figure 8.8 will now change as shown in Figure 8.9. The *GET: confirmed* cross-origin (legitimate) request to website A is now sent from the sub-execution at label P *with* its authentication cookie.

Figure 8.9: E-payment application encoding (updated)

...

15. (net_resp(P, h), [LOAD]): $(L, \_, \omega, [], []) \xrightarrow{15} (L, \_, \omega, [a.com/*], \bot :: P)$
    16. release (ui_page_loaded, [OUT-P]):
    $(L, \_, \omega, [a.com/*], \bot :: P) \xrightarrow{16} (L, \_, \omega, [a.com/*], \bot :: P)$
    17. suppress (net_req(u_A), [DROP-C]):
    $(L, \_, \omega, [a.com/*], \bot :: P) \xrightarrow{\bullet} (L, \_, \omega, [a.com/*], P)$
    16. suppress (ui_page_loaded, [OUT-P]):
    $(L, \_, \omega, [a.com/*], P) \xrightarrow{\bullet} (L, \_, \omega, [a.com/*], P)$
    17. release (net_req(u_A), [OUT-C]):
    $(L, \_, \omega, [a.com/*], P) \xrightarrow{17} (L, \_, \omega, [a.com/*], [])$

8.5 Implementation

The prototype implementation[3] is constructed as a modification of the FlowFox browser [38, 39]. Crucial for the implementation is the ability to keep track of all sites a user is logged into and to make sure that the labelling of JavaScript API calls can be dependent on this login history.

The biggest modification to FlowFox's core is the addition of a shared state variable, shared between all browser windows. This variable contains the login history log of the browser which is a list of strings and contains all domain names for which the browser has established an authenticated session. In the prototype, authentication to a web site has to happen by means of a bookmarklet that interacts with this login history log to add authenticated domains. The second modification is in the policy library that comes with FlowFox. This library now offers an API to query the login history log so that the labelling of JavaScript API calls can depend on this information.

The current prototype is just a proof-of-concept and has important limitations. The most important one is that FlowFox only performs multi-execution of JavaScript code and hence no policies can be enforced on network requests that are not triggered by scripts. If `attacker.com` tries to influence the session with `mail.com` via other means, e.g., an embedded image tag, thereby not relying on any JavaScript code, there is no way to intercept this in FlowFox. Removing this limitation is possible by multi-executing the entire browser, as proposed by Bielova et al. [15], but that would require a major overhaul of FlowFox and hence a substantial implementation effort. Despite this limitation, the prototype is evidence of the feasibility of the proposed mechanism in real browsers.

[3] Available online at `http://distrinet.cs.kuleuven.be/software/FlowFox/`.

Conclusion

In this dissertation, the formal verification of web sessions security in web browser and some browser-side mechanisms for enhancing this security were investigated. The security of web sessions was defined, both, in terms of information-flow property called noninterference and using access control model. The enforcement mechanisms defined for security policies are amenable to be enforced at the client side without requiring any help from the servers. Moreover, as the proof-of-concept, Chrome extensions were designed and implemented to enforce the relaxed versions of the security policies for protecting confidentiality and integrity of web sessions. The relaxed versions were carefully chosen to establish trade-offs between usability and security of web applications. The security policies and the level of protection they provided were analysed using extensive experiments on real life web applications while the extensions developed were installed in the Chrome browser.

8.6 Protecting Web Sessions

Web sessions can be attacked at network, session implementation and application layers. At the network layer, network sniffing or man-in-the-middle attacks can break the confidentiality or integrity of web sessions. This is a well-understood problem with may solutions and protection mechanisms proposed in the literature, for example, by *appropriate* use of transport level security techniques such as TLS/SSL, these attacks can be stopped. At the session implementation layer, script injection or again network level attacks can be used to steal a session cookie and hijack the session, or to fixate a session cookie on a client. Again, ensuring that

sessions only run over TLS/SSL, prohibiting script access to session cookies and enforcing renewal of a session on authentication, are appropriate countermeasures to such attacks.

Web sessions can be attacked at the application layer: since cookies are attached to HTTP requests by the browser automatically – without any web application involvement – any page in the browser can send malicious request to any of the servers that the browser currently has a session with, and that request will automatically get the session cookie attached and hence will be considered as part of a (possibly authenticated) session by the server. If the page sending the malicious request is from a different origin, such attacks are called CSRF (cross-site request forgery) attacks. But malicious requests can also be sent by scripts included in – or injected by an attacker into – a page from the same origin. Since both inclusions of third-party scripts and script injection vulnerabilities are common, these are important attack vectors.

The attacks at these three layers are roughly categorized as attacks on web session confidentiality and integrity. In the former category, the session cookie is either stolen using a JavaScript injected onto the page or intercepted in transit when sent in clear. In the later category, the attacker can force the browser to send malicious requests (e.g., by including a *<script>* tag or injecting a script) to any of the servers that the browser currently has a session with.

8.6.1 Protecting Attacks on Session Confidentiality

To tackle the attacks on confidentiality of web sessions, first a detailed survey was carried to assess the protection that the existing mechanisms, based on `Secure` and `HttpOnly` flags, provide to the web applications. The experiments showed that the actual adoption of these protection mechanisms was not satisfactory and hence suggested to 1) carry a formal analysis of the protection these mechanisms

can actually provide assuming they have been properly used by the web developers, and 2) then flag those sessions cookies at the client side which are not protected by the developers without breaking the web applications.

In the formal part [21], the web browser was modelled as a reactive system [19] by extending an existing model Featherweight Firefox [17] in the proof assistant Coq [32] and the security policy protecting web session confidentiality was defined in terms of reactive noninterference – the classical notion of noninterference but tailored towards reactive systems. Using the Coq facilities, the (model) of the web browser was proved secure according to the security policy. The noninterference security property proves strong security guarantees against, both implicit and explicit, information flows.

A prototype browser extension CookiExt [21] was developed to enforce the confidentiality policy at the client side. The extension first identifies session cookies using a heuristic and then set them with both `Secure` and `HttpOnly` flags if the websites are supporting TLS/SSL protocols and only `HttpOnly` for websites that do not support TLS/SSL protocols. Setting the flags with `HttpOnly` will protect them from JavaScript and the `Secure` flag will ensure their security in communication – all the subsequent requests over HTTP are redirected over HTTPS for supporting websites.

In the formal part, the security of the browser without the CookiExt is installed was proved, however, one may question the security of the CookiExt-patched web browser as the security guarantees of the browser without CookiExt does not ensure the security of the browser with CookiExt installed. A model of the browser patched with a model of the CookiExt was formalized in Coq and proved it secure according to a stronger security policy defined in terms of noninterference property.

8.6.2 Protecting Attacks on Session Integrity

Addressing the integrity of web sessions formally was much more challenging than confidentiality, in particular, using a detailed model as was used for proving confidentiality. However, a different approach was taken: a new lightweight browser model, called Flyweight Firefox [23], was built and extended with the enforcement mechanism for a security policy to protect session integrity against a number of already-known attacks and a new one. The security policy was enforced using access control model based on tainting network connections and pages loaded over tainted connections. Tainting network connections capture origin change during the network requests and responses. As the web pages may later initiate authenticated requests, the network connection taint is inherited to the page downloaded over it and hence a different security policy is adopted for the requests initiated from the tainted pages. The enforcement mechanism used in Flyweight Firefox was mathematically proved sound using a simulation-based technique.

Similar to CookiExt, a browser extension SessInt [23] was designed and implemented as the proof-of-concept of the enforcement mechanism used in the Flyweight Firefox. The experiments on real life web applications, when SessInt was installed, showed that it can provide protection against attacks on web session integrity and password theft.

To prove the session integrity in a more rigorous and precise way, web session integrity was formalized using information-flow model. As the trust level of network requests change while the user is surfing the web (e.g., when the user logs into a website), it was even more challenging to define information-flow policies based on noninterference. To track origin change when the user logs into websites, a *login history-dependent reactive noninterference* [73, 74] was defined and enforced using the secure multi-execution [44, 26] technique. The security of the enforcement mechanism was proved using the bi-simulation technique as was used

in the proof of session confidentiality.

8.7 Future Work

The formal web browser models are detailed enough to define interesting security policies and the enforcement mechanisms are based on, both information-flow control and access control, and the security policies can be enforced at the browser side without requiring server support. Similarly, the implementation as the browser extensions protect web sessions without breaking too many web applications. However, there is still room for further improvement, both, in the theoretical and implementation parts.

Both of the extensions, CookiExt and SessInt, are identifying session cookies using a heuristic. Although, the false positive and false negative rates are acceptable, however, the existing heuristic needs further refinement. A possible choice would be to integrate it with machine learning techniques as proposed by Calzavara et al. [25] in their recent work. Moreover, the effectiveness of the heuristic should also be analysed when used against web applications developed with web applications frameworks and content management systems. As these frameworks also handle aspects concerning session cookies, such analysis will help understanding the performance of the heuristic when used with applications that reflect correctly the everyday usage of the average user.

A further reasonable design improvement would be to combine both, CookiExt and SessInt, into a single browser extension. Both extensions enforce security policies that are not compatible with a number of applications such as collaborative web applications using SessInt. Although, a technique based on CSP headers was proposed [73], but that needs to be adopted by web applications which is not in practice now. To support such existing applications, the extensions (in

particular SessInt) need to be further reworked to solve the usability issues, without or minimum user intervention.

The security policy to protect web session integrity was enforced in a lightweight model Flyweight Firefox and was not directly mechanized in Coq. To ease the description in this dissertation, instead, the integrity security policy and enforcement mechanism was formalized in Coq as extension to the Extended Featherweight Firefox, however, the proofs were not carried mechanically. In future, carrying the proof in proof assistant Coq would be a nice contribution towards mechanizing web session integrity. The downside would be that it will require more effort than proving confidentiality but on the positive side, the existing Coq formalisation as extension of Featherweight Firefox (without proof) would provide the basic knowledge required to formalize integrity in proof assistants.

The integrity enforcement using the secure-multi execution is more precise and rigorous than the access control, however, it either misses or abstracts away a number of features needed to captures attacks (e.g., local CSRF attack). The initial investigation suggests that adding these features would be simple, however, it may further complicate the proof of login history-dependent noninterference. Furthermore, in the existing work, the security of the enforcement mechanism has been proved but not the precision. This later property would ensure that the enforcement mechanism change the behaviour of the web browser only in a *sensible* way.

A

Code Listings

A.1 expr Data Type

```
Inductive expr: Type :=
  | error_expr: String.t → expr
  | scoped_expr: context → expr → expr
  | null_expr: expr
  | nat_expr: nat → expr
  | str_expr: String.t → expr
  | url_expr: url → expr
  | closure_expr: context → var → list var → expr → expr
  | win_expr: win_ref → expr
  | node_expr: node_ref → expr
  | code_expr: script → expr
  | app_expr: expr → expr → expr
  | var_expr: var → expr
  | fun_expr: var → list var → expr → expr
  | eval_expr: expr → expr
  | seq_expr: expr → expr → expr
  | get_cookies_expr: expr
  | set_var_expr: var → expr → expr
  | xhr_expr: expr → expr → expr → expr
  | self_expr: expr
  | get_win_root_node_expr: expr → expr
  | new_div_node_expr: expr
  | remove_node_expr: expr → expr
  | insert_node_expr: expr → expr → expr → expr.
```

A.2 Rewriting Input Event

```
Definition rewrite_ie (ie: input_event) : input_event :=
```

```
  match ie with
  | user_load_in_new_window_event uwi u ⇒
     user_load_in_new_window_event uwi (rewrite_url u)
  | user_load_in_window_event uwi u ⇒
     user_load_in_window_event uwi (rewrite_url u)
  | network_document_response_event nci uwi rs ⇒
     match rs.(resp_redirect_uri) with
     | Some u ⇒ network_document_response_event
       (build_net_conn_id (rewrite_url nci.(net_conn_id_url))
       nci.(net_conn_id_value)) uwi
      (build_resp rs.(resp_del_cookies) rs.(resp_set_cookies)
      (Some (rewrite_url u)) (rewrite_file rs.(resp_file)))
     | None ⇒ network_document_response_event
       (build_net_conn_id (rewrite_url nci.(net_conn_id_url))
       nci.(net_conn_id_value)) uwi (build_resp rs.(resp_del_cookies)
       rs.(resp_set_cookies) None (rewrite_file rs.(resp_file)))
     end
  | network_script_response_event nci rs ⇒
     match rs.(resp_redirect_uri) with
     | Some u ⇒ network_script_response_event
        (build_net_conn_id (rewrite_url nci.(net_conn_id_url))
        nci.(net_conn_id_value))
        (build_resp rs.(resp_del_cookies) rs.(resp_set_cookies)
        (Some (rewrite_url u)) (rewrite_file rs.(resp_file)))
     | None ⇒ network_script_response_event
        (build_net_conn_id (rewrite_url nci.(net_conn_id_url))
        nci.(net_conn_id_value))
        (build_resp rs.(resp_del_cookies) rs.(resp_set_cookies)
        None (rewrite_file rs.(resp_file)))
     end
  | network_xhr_response_event nci rs ⇒
     match rs.(resp_redirect_uri) with
     | Some u ⇒ network_xhr_response_event
        (build_net_conn_id (rewrite_url nci.(net_conn_id_url))
        nci.(net_conn_id_value))
        (build_resp rs.(resp_del_cookies) rs.(resp_set_cookies)
        (Some (rewrite_url u)) (rewrite_file rs.(resp_file)))
     | None ⇒ network_xhr_response_event
        (build_net_conn_id (rewrite_url nci.(net_conn_id_url))
        nci.(net_conn_id_value))
        (build_resp rs.(resp_del_cookies) rs.(resp_set_cookies)
        None (rewrite_file rs.(resp_file)))
     end
  | _ ⇒ ie
  end.
```

A.3 same_form_ie_plus Relation

```
Definition same_form_ie_plus l (S: DomainSet.t) (ieL ieR: input_event)
: Prop :=
 match ieL, ieR with
  | network_document_response_event nciL uwiL rsL,
    network_document_response_event nciR uwiR rsR ⇒
      nciL == nciR ∧
      uwiL == uwiR ∧
      rsL.(resp_del_cookies) == rsR.(resp_del_cookies) ∧
      rsL.(resp_redirect_uri) == rsR.(resp_redirect_uri) ∧
      rsL.(resp_file) == rsR.(resp_file) ∧
      (forall k, StringMap.In k (resp_set_cookies rsL) ↔
        StringMap.In k (resp_set_cookies rsR)) ∧
      (erase_invis_cookies_plus l S nciL.(net_conn_id_url)
      rsL.(resp_set_cookies) == erase_invis_cookies_plus l S
      nciR.(net_conn_id_url) rsR.(resp_set_cookies))
  | network_script_response_event nciL rsL,
    network_script_response_event nciR rsR ⇒
      nciL == nciR ∧
      rsL.(resp_del_cookies) == rsR.(resp_del_cookies) ∧
      rsL.(resp_redirect_uri) == rsR.(resp_redirect_uri) ∧
      rsL.(resp_file) == rsR.(resp_file) ∧
      (forall k, StringMap.In k (resp_set_cookies rsL) ↔
         StringMap.In k (resp_set_cookies rsR)) ∧
      (erase_invis_cookies_plus l S nciL.(net_conn_id_url)
      rsL.(resp_set_cookies) == erase_invis_cookies_plus l S
      nciR.(net_conn_id_url) rsR.(resp_set_cookies))
  | network_xhr_response_event nciL rsL,
    network_xhr_response_event nciR rsR ⇒
      nciL == nciR ∧
      rsL.(resp_del_cookies) == rsR.(resp_del_cookies) ∧
      rsL.(resp_redirect_uri) == rsR.(resp_redirect_uri) ∧
      rsL.(resp_file) == rsR.(resp_file) ∧
      (forall k, StringMap.In k (resp_set_cookies rsL) ↔
         StringMap.In k (resp_set_cookies rsR)) ∧
      (erase_invis_cookies_plus l S nciL.(net_conn_id_url)
      rsL.(resp_set_cookies) == erase_invis_cookies_plus l S
      nciR.(net_conn_id_url) rsR.(resp_set_cookies))
  | _, _ ⇒ ieL == ieR
 end.
```

B

Proofs

B.1 Proof Technique

To prove our main result, we adapt Bohannon's ID-bisimulation proof technique [19] to support LHDNI. In the next definition, let C, C' range over wrapper consumer states of the form $(L, R, [])$; P, P' range over wrapper producer states of the form (L, R, L_q) with $L_q \neq []$; Q, Q' range over arbitrary wrapper states. Given a wrapper state Q, let L_Q represent the login history of Q.

Definition 7 (ID-bisimulation)**.** *An ID-bisimulation on a reactive system is a label-indexed family of binary relations on states (written $\sim_l$) with the following properties:*

(a) if $Q \sim_l Q'$, then $Q' \sim_l Q$;

(b) if $C \sim_l C'$ and $C \xrightarrow{i} P$ and $C' \xrightarrow{i} P'$ and $lbl_{L_C}(i) \leq l$, then $P \sim_l P'$;

(c) if $C \sim_l C'$ and $C \xrightarrow{i} P$ and $lbl_{L_C}(i) \not\leq l$, then $P \sim_l C'$;

(d) if $P \sim_l C$ and $P \xrightarrow{o} Q$, then $lbl_{L_P}(o) \not\leq l$ and $Q \sim_l C$;

(e) if $P \sim_l P'$, then either:

1. *$P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ imply $o = o'$ and $Q \sim_l Q'$, or*
2. *$P \xrightarrow{o} Q$ implies $lbl_{L_P}(o) \not\leq l$ and $Q \sim_l P'$, or*

3. $P' \xrightarrow{o'} Q'$ *implies* $lbl_{L_{P'}}(o') \not\leq l$ *and* $P \sim_l Q'$.

Lemma 2. *All the following properties hold true:*

1. *if* $C \xrightarrow{i} P$ *with* $i \neq \mathsf{login}(d)$, *then* $L_P = L_C$;
2. *if* $C \xrightarrow{i} P$ *with* $i = \mathsf{login}(d)$, *then* $L_P = L_C \oplus d$;
3. *if* $P \xrightarrow{o} Q$, *then* $L_Q = L_P$;

Proof. By a case analysis on the transition step in the antecedent. □

All the next results only hold true assuming some mild, implicit syntactic restrictions on the format of the history-dependant labelling function for input/output events. These restrictions are formalized in the next definition and assumed throughout all the proofs.

Definition 8 (Well-formed labelling function)**.** *A labelling function lbl is* well-formed *if and only if all the following properties hold true:*

1. $\forall L : ran(lbl_L) \subseteq L$;
2. $\forall L, d, s : lbl_{L\oplus d}(s) \neq lbl_L(s) \Rightarrow lbl_{L\oplus d}(s) = d \wedge lbl_L(s) = \bot$;
3. $\forall L, L', d, s : lbl_{L\oplus d}(s) = d \Rightarrow lbl_{L'\oplus d}(s) = d$;
4. $\forall L : lbl_L(\bullet) = \bot$.

Lemma 3. *Both the following properties hold true:*

1. *if* $L \vdash S \approx_l S'$ *and* $d \not\leq l$, *then* $L \oplus d \vdash S \approx_l S'$;
2. *if* $L \oplus d \vdash S \approx_l S'$, *then* $L \vdash S \approx_l S'$.

Proof. Both the points are proved by coinduction, exploiting the well-formation assumption for the labelling function (points 1 and 2 of the Definition 8). □

Lemma 4. *Suppose that* $Q \sim_l Q'$*, and that* $Q(I) \rightsquigarrow S$ *and* $Q'(I') \rightsquigarrow S'$*. Then* $L_Q \vdash I \approx_l I'$ *implies* $L_Q \vdash S \approx_l S'$*.*

Proof. We prove the result by coinduction. As both states, Q and Q', can be either consumer or producer states, we have four cases:

1. Assume the states are producer states, i.e., for some P and P', we have $Q = P$ and $Q' = P'$. By Definition 7, item (e), there are three cases:

   - By inverting the assumption that $P(I) \rightsquigarrow S$, we have $P \xrightarrow{o} Q_1$ for some Q_1 such that $Q_1(I) \rightsquigarrow S_1$ and $S = o :: S_1$. By inverting the assumption $P'(I') \rightsquigarrow S'$, we have $P' \xrightarrow{o'} Q'_1$ for some Q'_1 such that $Q'_1(I') \rightsquigarrow S'_1$ and $S' = o' :: S'_1$. By the assumption in this case, we know $o = o'$ and $Q_1 \sim_l Q'_1$. By Lemma 2, we know that $L_{Q_1} = L_P$, hence by the coinduction hypothesis we get $L_P \vdash S_1 \approx_l S'_1$. Since $o = o'$, either $lbl_{L_P}(o) \leq l$ and $lbl_{L_P}(o') \leq l$, or $lbl_{L_P}(o) \not\leq l$ and $lbl_{L_P}(o') \not\leq l$. In the first case, the conclusion follows using [ID-SIM], while in the second case, the conclusion follows using [ID-L] in sequence with [ID-R].
   - By inverting the assumption that $P(I) \rightsquigarrow S$, we have $P \xrightarrow{o} Q_1$ for some Q_1 such that $Q_1(I) \rightsquigarrow S_1$ and $S = o :: S_1$. By the assumption in this case, we know $lbl_{L_P}(o) \not\leq l$ and $Q_1 \sim_l P'$. By Lemma 2, we know that $L_{Q_1} = L_P$, hence by coinduction hypothesis we get $L_P \vdash S_1 \approx_l S'$. The conclusion then follows using [ID-L].
   - Same as above, but using [ID-R] to conclude.

2. Assume $Q = P$ and $Q' = C'$ for some producer and consumer states P and C' respectively. By inverting the assumption $P(I) \rightsquigarrow S$, we have $P \xrightarrow{o} Q_1$ for some Q_1 such that $Q_1(I) \rightsquigarrow S_1$ and $S = o :: S_1$. According to the item (d) in Definition 7, we know that $lbl_{L_P}(o) \not\leq l$ and $Q_1 \sim_l Q'$. By Lemma 2,

we know that $L_{Q_1} = L_P$, hence by using the coinduction hypothesis we get $L_P \vdash S_1 \approx_l S'$. The conclusion then follows using [ID-L].

3. Assume $Q = C$ and $Q' = P'$ for some consumer and producer states C and P' respectively. Using the symmetry case (Definition 7, item (a)), this case becomes similar to the case (2) above.

4. Assume $Q = C$ and $Q' = C'$ for some consumer states C and C'. By inverting the assumption $L_C \vdash I \approx_l I'$, we have five cases:

   - $I = I' = []$. Using inversion on $C([]) \rightsquigarrow S$ and $C'([]) \rightsquigarrow S'$, we have $S = S' = []$, and hence the case is closed using [ID-NIL].
   - $I = i :: I_1$, where $lbl_{L_C}(i) \not\leq l$ and $L_C \vdash I_1 \approx_l I'$. By inverting the assumption that $C(i :: I_1) \rightsquigarrow S$, we have $C \xrightarrow{i} P$ for some P such that $P(I_1) \rightsquigarrow S_1$ and $S = i :: S_1$. By Definition 7, item (c), we know that $P \sim_l Q'$. If $L_P = L_C$, by the coinduction hypothesis we get $L_C \vdash S_1 \approx_l S'$ and we conclude by using [ID-L]. Otherwise, by Lemma 2, we have $L_P = L_C \oplus d$ for some d such that $i = \mathsf{login}(d)$. By the first point of Lemma 3, observing that $lbl_{L_C}(i) = lbl_{L_C}(\mathsf{login}(d)) = d \not\leq l$ by hypothesis, we know that $L_C \vdash I_1 \approx_l I'$ implies $L_P \vdash I_1 \approx_l I'$. Hence, we can appeal to the coinduction hypothesis to get $L_P \vdash S_1 \approx_l S'$. By the second point of Lemma 3, we then get $L_C \vdash S_1 \approx_l S'$ and we conclude $L_C \vdash S \approx_l S'$ by using [ID-L];
   - $I' = i :: I'_1$, where $lbl_{L_C}(i) \not\leq l$ and $L_C \vdash I \approx_l I'_1$. This is analogous to the case above, except it is closed by using [ID-R].
   - $I = i :: I_1$ and $I' = i :: I'_1$, with $lbl_{L_C}(i) \leq l$, $i \neq \mathsf{login}(d)$, $L_C \vdash I_1 \approx_l I'_1$. By inverting the assumption that $C(i :: I_1) \rightsquigarrow S$, we have $C \xrightarrow{i} P$ for some P such that $P(I_1) \rightsquigarrow S_1$ and $S = i :: S_1$. By inverting the

assumption that $C'(i :: I'_1) \rightsquigarrow S'$, we have $C' \xrightarrow{i} P'$ for some P' such that $P'(I') \rightsquigarrow S'_1$ and $S' = i :: S'_1$. By the item (b) in Definition 7, we know that $P \sim_l P'$. By Lemma 2, we know that $L_P = L_C$, hence we get $L_C \vdash S_1 \approx_l S'_1$ by the coinduction hypothesis. We then conclude by using [ID-SIM].

- $I = i :: I_1$ and $I' = i :: I'_1$, where $lbl_{L_C}(i) \leq l$, $i = \mathsf{login}(d)$ and $L_C \oplus d \vdash I_1 \approx_l I'_1$. By inverting the assumption that $C(i :: I_1) \rightsquigarrow S$, we have $C \xrightarrow{i} P$ for some P such that $P(I_1) \rightsquigarrow S_1$ and $S = i :: S_1$. By inverting the assumption that $C'(i :: I'_1) \rightsquigarrow S'$, we have $C' \xrightarrow{i} P'$ for some P' such that $P'(I') \rightsquigarrow S'_1$ and $S' = i :: S'_1$. By the item (b) in Definition 7, we know that $P \sim_l P'$. By Lemma 2, we know that $L_P = L_C \oplus d$, hence we get $L_C \oplus d \vdash S_1 \approx_l S'_1$ by the coinduction hypothesis. We then conclude by using [ID-LOGIN].

□

Theorem 5 (Noninterference)**.** *Let Q be an initial wrapper state. If $Q \sim_l Q$ for all l, then Q is LHDNI.*

Proof. By a direct application of Lemma 4. □

B.2 Proof of the Main Result

We first recall that the wrapper is intended to protect *deterministic* reactive systems, according to the following definition.

Definition 9 (Determinism)**.** *A reactive system is* deterministic *iff:*

- *for all $P \in ProducerState$, $(P \xrightarrow{o} Q \wedge P \xrightarrow{o'} Q') \Rightarrow (o = o' \wedge Q = Q')$*
- *for all $C \in ConsumerState$, $(C \xrightarrow{i} P \wedge C \xrightarrow{i} P') \Rightarrow P = P'$*

Lemma 5. *Let L_1, L_2 be two login histories such that $L_{1|l} = L_{2|l}$. If $lbl_{L_1}(s) \leq l$ for some event s, then $lbl_{L_2}(s) = lbl_{L_1}(s)$.*

Proof. By exploiting the well-formation assumption for the labelling function. In particular, by point 2, we observe that the presence of a domain d in L_1 may only be relevant to assign a level d to some event s. Let then $lbl_{L_1}(s) = d \leq l$: given that $L_{1|l} = L_{2|l}$, we know that d belongs also to L_2. Hence, $lbl_{L_2}(s) = d$ by point 3 of the Definition 8. □

Lemma 1. *The l-similarity relation is an ID-bisimulation relation.*

Proof. To prove the result, we show that the five properties in Definition 7 hold true for the l-similarity relation in Definition 6.

a) We want to prove that, if $(L_1, R_1, L_{q1}) \approx_l (L_2, R_2, L_{q2})$, then we have $(L_2, R_2, L_{q2}) \approx_l (L_1, R_1, L_{q1})$.

Since $(L_1, R_1, L_{q1}) \approx_l (L_2, R_2, L_{q2})$, we have $L_{1|l} = L_{2|l}$ and $\forall l' \leq l$, $R_1(l') = R_2(l')$ and $L_{q1|l} = L_{q2|l}$ by definition of l-similarity, hence $(L_2, R_2, L_{q2}) \approx_l (L_1, R_1, L_{q1})$ by the symmetry of equality.

b) We want to prove that, if $(L_1^C, R_1^C, []) \approx_l (L_2^C, R_2^C, [])$ and $(L_1^C, R_1^C, []) \xrightarrow{i} (L_1^P, R_1^P, L_{q1}^P)$ and $(L_2^C, R_2^C, []) \xrightarrow{i} (L_2^P, R_2^P, L_{q2}^P)$ and $lbl_{L_1^C}(i) \leq l$, then we have $(L_1^P, R_1^P, L_{q1}^P) \approx_l (L_2^P, R_2^P, L_{q2}^P)$.

Before we start the proof, we observe that $(L_1^C, R_1^C, []) \xrightarrow{i} (L_1^P, R_1^P, L_{q1}^P)$ and $(L_2^C, R_2^C, []) \xrightarrow{i} (L_2^P, R_2^P, L_{q2}^P)$ must be derived by the same rule. Indeed, if $i \neq \mathsf{login}(d)$, then only rule [LOAD] is available. Assume instead that $i = \mathsf{login}(d)$: since $lbl_{L_1^C}(i) = d \leq l$, we know that $d \in L_1^C$ iff $d \in L_2^C$, given that the assumption $(L_1^C, R_1^C, []) \approx_l (L_2^C, R_2^C, [])$ implies $L_{1|l}^C = L_{2|l}^C$. Hence, rule [LOGIN] is either applied to both the input transitions or to none.

Since $(L_1^C, R_1^C, []) \approx_l (L_2^C, R_2^C, [])$, we have $L_{1|l}^C = L_{2|l}^C$ and $\forall l' \leq l$, $R_1^C(l') = R_2^C(l')$, therefore, $(L_1^P, R_1^P, L_{q1}^P) \approx_l (L_2^P, R_2^P, L_{q2}^P)$ because:

- There are two cases: either the reduction steps are triggered by [LOGIN] or they are triggered by [LOAD]. In the first case, we have $L_1^P = L_1^C \oplus d$ and $L_2^P = L_2^C \oplus d$. Since $d \leq l$, we have $L_{1|l}^P = L_{1|l}^C \oplus d$ and $L_{2|l}^P = L_{2|l}^C \oplus d$, hence $L_{1|l}^P = L_{2|l}^P$ from the assumption $L_{1|l}^C = L_{2|l}^C$. If instead the applied rule is [LOAD], no new label is added to the sets L_1^C and L_2^C, hence the login histories upto level l are still the same, that is $L_{1|l}^P = L_{1|l}^C = L_{2|l}^C = L_{2|l}^P$.

- There are two cases: either the reduction steps are triggered by [LOGIN] or they are triggered by [LOAD]. We show the first case in detail, the second one is easier. Let then $i = \mathsf{login}(d)$ with $d \notin L_1^C$ and $d \notin L_2^C$. Recall that $\forall l' \leq l$, $R_1^C(l') = R_2^C(l')$. Since the original reactive system is deterministic, $\forall l' \neq d : l' \leq l$, $R_1^C(l') \xrightarrow{i} P$ and $R_2^C(l') \xrightarrow{i} P'$ imply $P = P'$, hence $\forall l' \neq d : l' \leq l$, $R_1^P(l') = R_2^P(l')$. For $d \leq l$, since $R_1^C(\top) = R_2^C(\top)$ and the original reactive system is deterministic, we have $R_1^C(\top) \xrightarrow{i} P_\top$ and $R_2^C(\top) \xrightarrow{i} P_\top$, hence $R_1^P(d) = R_2^P(d)$. Therefore $\forall l' \leq l$, $R_1^P(l') = R_2^P(l')$, i.e., we have $R_1^P \approx_l R_2^P$.

- both of the lists, L_{q1}^P and L_{q2}^P, are populated by the same input i. As $L_{1|l}^C = L_{2|l}^C$ and $lbl_{L_1^C}(i) \leq l$, we know that $lbl_{L_2^C}(i) = lbl_{L_1^C}(i)$ by Lemma 5. Hence, both lists $L_{q1|l}^P$ and $L_{q2|l}^P$ are populated with the same labels l' such that $lbl_{L_1^C}(i) \leq l' \leq l$ and, regardless of the rule applied, we have $L_{q1|l}^P = L_{q2|l}^P$.

c) We want to prove that, if $(L_1^C, R_1^C, []) \approx_l (L_2^C, R_2^C, [])$ and $(L_1^C, R_1^C, []) \xrightarrow{i} (L_1^P, R_1^P, L_{q1}^P)$ and $lbl_{L_1^C}(i) \not\leq l$, then $(L_1^P, R_1^P, L_{q1}^P) \approx_l (L_2^C, R_2^C, [])$.

Since $(L_1^C, R_1^C, []) \approx_l (L_2^C, R_2^C, [])$, we have $L_{1|l}^C = L_{2|l}^C$ and $\forall\ l'$: $l' \leq l$, $R_1^C(l') = R_2^C(l')$. We show the three required conditions for similarity:

- There are two cases: either of the rules [LOGIN] or [LOAD] applies. If rule [LOGIN] is applied, then for $i = \mathsf{login}(d)$ with $lbl_{L_1^C}(i) \not\leq l$, domain $d \not\leq l$ is added, that is $L_1^P = L_1^C \oplus d$, hence $L_{1|l}^P = L_{1|l}^C = L_{2|l}^C$ from the assumption

$L^C_{1|l} = L^C_{2|l}$. If rule [LOAD] is applied, the login history does not change, hence $L^P_{1|l} = L^C_{1|l}$ and we have $L^P_{1|l} = L^C_{1|l} = L^C_{2|l}$.

- there are two cases: either of the rules [LOGIN] or [LOAD] applies. In both cases, since $lbl_{L^C_1}(i) \not\leq l$, the input i can only add new sub-executions at $l' \not\leq l$, hence $\forall\, l'$: $l' \leq l$, $R^P_1(l') = R^C_1(l') = R^C_2(l')$, which implies $R^P_1 \approx_l R^C_2$.

- by definition, L^P_{q1} contains levels l' such that $lbl_{L^C_1}(i) \leq l'$ and since we know that $lbl_{L^C_1}(i) \not\leq l$, we have $L^P_{q1|l} = []$.

Combining the information above, $L^P_{1|l} = L^C_{2|l}$ and $R^P_1 \approx_l R^C_2$ and $L^P_{q1|l} = []$ imply $(L^P_1, R^P_1, L^P_{q1}) \approx_l (L^C_2, R^C_2, [])$.

d) We want to prove that, if $(L^P_1, R^P_1, L^P_{q1}) \approx_l (L^C_1, R^C_1, [])$ and $(L^P_1, R^P_1, L^P_{q1}) \xrightarrow{o} (L^Q_1, R^Q_1, L^Q_{q1})$, then $lbl_{L^P_1}(o) \not\leq l$ and $(L^Q_1, R^Q_1, L^Q_{q1}) \approx_l (L^C_1, R^C_1, [])$.

We start by proving that $lbl_{L^P_1}(o) \not\leq l$. Since $(L^P_1, R^P_1, L^P_{q1}) \approx_l (L^C_1, R^C_1, [])$, we have $L^P_{1|l} = L^C_{1|l}$ and $\forall\, l'$: $l' \leq l$, $R^P_1(l') = R^C_1(l')$ and $L^P_{q1|l} = []$. In particular, this implies that $l \neq \bot$: indeed, assume by contradiction that $l = \bot$. Then we have $L^P_{q1|\bot} = L^P_{q1}$ and we know that $L^P_{q1|\bot} = L^P_{q1} \neq []$ by definition of producer state, which contradicts the assumption that $L^P_{q1|\bot} = []$. Having established this fact, we perform a case distinction on the rule applied to produce the output event: if either [DROP-P] or [DROP-C] is used, the dummy output $\bullet$ is released and we know that $lbl_{L^P_1}(\bullet) = \bot \not\leq l$ for any $l \neq \bot$. Otherwise, either [OUT-P] or [OUT-C] is used and a regular output o is released. Since $L^P_{q1|l} = []$, there is no sub-execution at or below l, hence the output o must have been released by a sub-execution at l' such that $l' \not\leq l$. Since the label of the sub-execution is the same as the label of the output, this implies $lbl_{L^P_1}(o) \not\leq l$.

To prove the second part of the property, as before, the three parts of the relation $\approx_l$ are proved one by one:

- we observe that $(L^P_1, R^P_1, L^P_{q1}) \xrightarrow{o} (L^Q_1, R^Q_1, L^Q_{q1})$ can be derived by [OUT-P],

[OUT-C], [DROP-P] or [DROP-C], but none of these rules changes the login history L_1^P, so $L_1^Q = L_1^P$ implies $L_{1|l}^Q = L_{1|l}^P = L_{1|l}^C$ by hypothesis.

- we observe that $(L_1^P, R_1^P, L_{q1}^P) \xrightarrow{o} (L_1^Q, R_1^Q, L_{q1}^Q)$ can be derived by [OUT-P], [OUT-C], [DROP-P] or [DROP-C]. In the first two cases, the output is released by a sub-execution at level $l' = lbl_{L_1^P}(o) \not\leq l$. Otherwise, the output is dropped from a sub-execution at level l', where l' is the head of L_{q1}^P, but recall that L_{q1}^P does not contain any level which is bounded above by l, since we know that $L_{q1|l}^P = []$; hence, also in this case we know that $l' \not\leq l$. Thus, the original reactive system state $R_1^P(l')$ at $l' \not\leq l$ may change, but any state of a sub-execution at or below l will remain the same, which allows us to conclude that $R_1^Q \approx_l R_1^C$ from the hypothesis $R_1^P \approx_l R_1^C$.

- when any of the rules [OUT-P] or [DROP-P] is used, the list of upper integrity levels does not change − that is $L_{q1}^Q = L_{q1}^P$ implies $L_{q1|l}^Q = L_{q1|l}^P = []$. In the other case, when the output is released/dropped using rule [OUT-C] or [DROP-C] from a copy $R_1^P(l')$ such that $l' \not\leq l$, then the level l' is removed from the list of levels L_{q1}^P. However, as $l' \not\leq l$, the list of levels bounded above by l will not change − that is $L_{q1|l}^Q = L_{q1|l}^P = []$.

By combining all the information above, we get $(L_1^Q, R_1^Q, L_{q1}^Q) \approx_l (L_1^C, R_1^C, [])$.

e) We want to prove that, if $(L_1^P, R_1^P, L_{q1}^P) \approx_l (L_2^P, R_2^P, L_{q2}^P)$, then either

1) $(L_1^P, R_1^P, L_{q1}^P) \xrightarrow{o_1} (L_1^Q, R_1^Q, L_{q1}^Q)$ and $(L_2^P, R_2^P, L_{q2}^P) \xrightarrow{o_2} (L_2^Q, R_2^Q, L_{q2}^Q)$ implies $o_1 = o_2$ and $(L_1^Q, R_1^Q, L_{q1}^Q) \approx_l (L_2^Q, R_2^Q, L_{q2}^Q)$, or else

2) $(L_1^P, R_1^P, L_{q1}^P) \xrightarrow{o} (L_1^Q, R_1^Q, L_{q1}^Q)$ implies $lbl_{L_1^P}(o) \not\leq l$ and $(L_1^Q, R_1^Q, L_1^Q) \approx_l (L_2^P, R_2^P, L_{q2}^P)$, or else

3) $(L_2^P, R_2^P, L_{q2}^P) \xrightarrow{o} (L_2^Q, R_2^Q, L_{q2}^Q)$ implies $lbl_{L_2^P}(o) \not\leq l$ and $(L_1^P, R_1^P, L_{q1}^P) \approx_l (L_2^Q, R_2^Q, L_{q2}^Q)$.

Since $(L_1^P, R_1^P, L_{q1}^P) \approx_l (L_2^P, R_2^P, L_{q2}^P)$, we have $L_{1|l}^P = L_{2|l}^P$, $\forall\ l'\colon\ l' \leq l$, $R_1^P(l') = R_2^P(l')$ and $L_{q1|l}^P = L_{q2|l}^P$. Notice that $L_{q1}^P \neq []$ and $L_{q2}^P \neq []$ because (L_1^P, R_1^P, L_{q1}^P) and (L_2^P, R_2^P, L_{q2}^P) are producer states.

CASE e).1. We match this case whenever $L_{q1}^P = l_1 :: L_1$ and $L_{q2}^P = l_2 :: L_2$ with $l_1 \leq l$ and $l_2 \leq l$. Given that $L_{q1|l}^P = L_{q2|l}^P$, we know that $l_1 = l_2 \leq l$. We observe that one of the rules [OUT-P], [OUT-C], [DROP-P] or [DROP-C] applies to the first system. Assume for instance that [OUT-P] is used, then the output o is released by a sub-execution at l_1, that is $R_1^P(l_1)$, such that $l_1 \leq l$ and $lbl_{L_1^P}(o) = l_1$. Since $l_1 \leq l$, we know that $R_1^P(l_1) = R_2^P(l_1)$, hence also $R_2^P(l_1)$ can fire the same output o, which is actually the only available output, being the underlying reactive system deterministic. Observe now that $lbl_{L_2^P}(o) = lbl_{L_1^P}(o)$ by Lemma 5, since $lbl_{L_1^P}(o) \leq l$ and $L_{1|l}^P = L_{2|l}^P$. Hence, rule [OUT-P] applies also to the second system and we emit the same output event. The cases where either [OUT-C], [DROP-P] or [DROP-C] is applied are analogous.

To show that $(L_1^Q, R_1^Q, L_{q1}^Q) \approx_l (L_2^Q, R_2^Q, L_{q2}^Q)$, we first observe as before that the same output rule must be used in both systems. We then notice that: (1) none of the output rules changes the login history, (2) the two sub-executions at $l_1 \leq l$ evolve in the same way, since the underlying reactive system is deterministic, and (3) either the upper integrity lists do not change or they are changed in the same way by removing $l_1 \leq l$.

To ease the proof of cases e).2 and e).3, we will use parameters i, j: substitution $i = 1$, $j = 2$ proves e).2 while substitution $i = 2$, $j = 1$ proves e).3.

CASE e).2, e).3. We match this case whenever $L_{qi}^P = l_1' :: L_i$ with $l_1' \not\leq l$. Since $l_1' \not\leq l$, we know that $l \neq \bot$. We observe that one of the rules [OUT-P], [OUT-C], [DROP-P] or [DROP-C] applies to the system i to produce an output o_i. If rule [DROP-P] or [DROP-C] applies, the output is $\bullet$ and hence $lbl_{L_i^P}(o_i) = \bot \not\leq l$, since $l \neq \bot$. If rule [OUT-P] or [OUT-C] applies, then $lbl_{L_i^P}(o_i) = l_1' \not\leq l$.

To show that $(L_i^Q, R_i^Q, L_i^Q) \approx_l (L_j^P, R_j^P, L_{qj}^P)$, we notice that: (1) none of the output rules changes the login history, (2) the evolving sub-execution is at level $l_1' \not\preceq l$, hence it cannot break l-similarity, and (3) either the upper integrity list does not change or it is changed by removing $l_1' \not\preceq l$. □

Lemma 6. *For any state* (L, R, L_q)*, we have* $(L, R, L_q) \approx_l (L, R, L_q)$*.*

Proof. Immediate by Definition 6. □

Theorem 4 (Security). *All the initial states of the wrapper are LHDNI.*

Proof. Let $(\{\bot, \top\}, R, [])$ be an initial state of the wrapper. By Lemma 6, we have $(\{\bot, \top\}, R, []) \approx_l (\{\bot, \top\}, R, [])$ for any l. Since $\approx_l$ is an ID-bisimulation by Lemma 1, the conclusion follows by Theorem 5. □

Bibliography

[1] Mircosoft: Mitigating cross-site scripting with HTTP-only cookies.

[2] Damn vulnerable web application. `http://www.dvwa.co.uk/`, 2013.

[3] Owsap consortium. `http://www.owasp.org/`, 2013.

[4] The android project (source code and SDK). `http://source.android.com/`, 2014.

[5] Malicious HTML tags embedded in client web requests. CERT Advisory CA–2000–02, February 2000.

[6] Ben Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the 17th international conference on World Wide Web*, pages 517–524. ACM, 2008.

[7] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitechell, and Dawn Song. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 290–304. IEEE Computer Society, 2010.

[8] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 247–262. IEEE, 2012.

[9] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[10] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[11] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser, 2008.

[12] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[13] Giampaolo Bella and Lawrence C Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In *Computer Security—ESORICS 98*, pages 361–375. Springer, 1998.

[14] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[15] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *Proc. of the International Conference on Network and System Security*, pages 97–104, 2011.

[16] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for the browser: extended version. *CW Reports*, 2011.

[17] Aaron Bohannon. *Foundations of webscript security.* PhD thesis, University of Pennsylvania, 2012.

[18] 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, Berkeley, CA, USA, 2010. USENIX Association.

[19] 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.

[20] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin cookies: Session integrity for web applications. *Web 2.0 Security and Privacy (W2SP)*, 2011.

[21] 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*, pages 161–178. Springer, 2014.

[22] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan, and Mauro Tempesta. Provably sound browser-based enforcement of web session integrity (full version). `http://www.dais.unive.it/~calzavara/csf14-full.pdf`.

[23] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan, and Mauro Tempesta. Provably sound browser-based enforcement of web session integrity. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 366–380. IEEE, 2014.

[24] Michael Burrows, Martin Abadi, and Roger M Needham. A logic of authentication. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 426(1871):233–271, 1989.

[25] Stefano Calzavara, Gabriele Tolomei, Michele Bugliesi, and Salvatore Orlando. Quite a mess in my cookie jar!: leveraging machine learning to protect web authentication. In *Proceedings of the 23rd international conference on*

World wide web, pages 189–200. International World Wide Web Conferences Steering Committee, 2014.

[26] Roberto Capizzi, Antonio Longo, VN Venkatakrishnan, and A Prasad Sistla. Preventing information leaks through shadow executions. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 322–331. IEEE, 2008.

[27] Avik Chaudhuri. Language-based security on android. In *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security*, pages 1–7. ACM, 2009.

[28] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 227–238. ACM, 2011.

[29] David D Clark and David R Wilson. A comparison of commercial and military computer security policies. In *2012 IEEE Symposium on Security and Privacy*, pages 184–184. IEEE Computer Society, 1987.

[30] World Wide Web Consortium et al. Document Object Model (DOM) level 3 core specification. 2004.

[31] World Wide Web Consortium et al. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. 2011.

[32] The Coq Development Team. *The Coq Reference Manual, version 8.3*, December 2011. Available electronically at `http://coq.inria.fr/distrib/V8.3pl5/files/Reference-Manual.pdf`.

[33] Francisco Corella and K Lewison. Security analysis of double redirection protocols. Technical report, Pomcor Technical Report, 2011.

[34] Silvia Crafa, Matteo Mio, Marino Miculan, Carla Piazza, and Sabina Rossi. PicNIc-pi-calculus non-interference checker. In *Application of Concurrency to System Design, 2008. ACSD 2008. 8th International Conference on*, pages 33–38. IEEE, 2008.

[35] 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 (TOIT)*, 12(1):1, 2012.

[36] Anupam Datta, Ante Derek, John C Mitchell, and Arnab Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007.

[37] Anupam Datta, Ante Derek, John C Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *Automata, Languages and Programming*, pages 16–29. Springer, 2005.

[38] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 748–759. ACM, 2012.

[39] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 2014.

[40] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Engineering Secure Software and Systems*, pages 18–34. Springer, 2010.

[41] 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*, pages 59–72. Springer, 2012.

[42] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[43] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[44] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 109–124. IEEE, 2010.

[45] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176.

[46] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[47] Tim Dierks. The transport layer security (TLS) protocol version 1.2. 2008.

[48] ECMA Ecma. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[49] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.

[50] Jordan Elks. Man in the Middle Attack: Focus on SSLStrip. 2011.

[51] David Endler. The evolution of cross site scripting attacks. Technical report, 2002.

[52] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[53] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2002.

[54] M. Foundation. Public suffix list: Learn more about the public suffix list. `http://publicsuffix.org/learn/`, 2013.

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

[56] Ben SY Fung and Patrick PC Lee. A privacy-preserving defense mechanism against request forgery attacks. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 45–52. IEEE, 2011.

[57] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[58] David Gourley and Brian Totty. *HTTP: the definitive guide.* " O'Reilly Media, Inc.", 2002.

[59] Chris Grier, Shuo Tang, and Samuel T King. Secure web browsing with the OP web browser. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 402–416. IEEE, 2008.

[60] Jeremiah Grossman. *XSS Attacks: Cross-site scripting exploits and defense.* Syngress, 2007.

[61] Elliotte Rusty Harold. Privacy tip# 3: Block referer headers in Firefox, october 2006. `http://cafe.elharo.com/privacy/privacy-tip-3-block-referer-headers-in-firefox/`.

[62] Collin Jackson and Adam Barth. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the 17th international conference on World Wide Web*, pages 525–534. ACM, 2008.

[63] Collin Jackson, Andrew Bortz, Dan Boneh, and John C Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, pages 737–744. ACM, 2006.

[64] Jeffrey C Jackson. *Web Technologies: a computer science perspective.* Prentice-Hall, Inc., 2006.

[65] Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.

[66] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the*

21st USENIX conference on Security symposium, pages 8–8. USENIX Association, 2012.

[67] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 601–610. ACM, 2007.

[68] Martin Johns. On JavaScript Malware and Related Threats - Web Page Based Attacks Revisited. *Journal in Computer Virology*, 4(3):161 – 178, August 2008.

[69] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1531–1537. ACM, 2011.

[70] Martin Johns, Sebastian Lekies, Bastian Braun, and Benjamin Flesch. BetterAuth: web authentication revisited. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 169–178. ACM, 2012.

[71] Martin Johns and Justus Winter. RequestRodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[72] Bill Kennedy and Chuck Musciano. *HTML & XHTML: The Definitive Guide*. O'Reilly, 2002.

[73] 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 Systems Security*.

[74] Wilayat Khan, Stefano Calzavara, Michele Bugliesi, Willem De Groef, and Frank Piessens. Client side web session integrity as a non-interference prop-

erty: Extended version with proofs. Available at `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW674.abs.html`.

[75] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM, 2006.

[76] Mitja Kolšek. Session fixation vulnerability in web-based applications. *Acros Security*, page 7, 2002.

[77] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109 (Historic), February 1997. Obsoleted by RFC 2965.

[78] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965 (Historic), October 2000. Obsoleted by RFC 6265.

[79] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *In Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST*. Citeseer, 2003.

[80] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. OAuth 2.0 threat model and security considerations. 2013.

[81] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with i/o. *Journal of Symbolic Computation*, 46(2):95–118, 2011.

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

[83] John McLean. Security models and information flow. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 180–187. IEEE, 1990.

[84] Eric A Meyer. *CSS: The Definitive Guide: The Definitive Guide.* " O'Reilly Media, Inc.", 2006.

[85] Anthony D Miyazaki. Online privacy and the disclosure of cookie use: Effects on consumer trust and anticipated patronage. *Journal of Public Policy & Marketing*, 27(1):19–33, 2008.

[86] A. C. Myers. JFlow : Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999.

[87] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[88] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. software release, 2001.

[89] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747. ACM, 2012.

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

[91] Nick Nikiforakis, Yves Younan, and Wouter Joosen. Hproxy: Client-side detection of ssl stripping attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 200–218. Springer, 2010.

[92] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale*, volume 152, page 1. Citeseer, 2006.

[93] Steven Pemberton. XHTML™ 1.0 the extensible hypertext markup language (second edition). W3C recommendation, W3C, August 2002. http://www.w3.org/TR/xhtml1/.

[94] Tor Project and the Electronic Frontier Foundation. HTTPS Everywhere. `https://www.eff.org/https-everywhere`, 2014.

[95] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security*, volume 3, 2003.

[96] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 33–48. IEEE, 2013.

[97] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. HTML 4.01 specification. *W3C recommendation*, 24, 1999.

[98] Vijay Raghvendra. Session tracking on the web. Internetworking, 2000.

[99] Jesse Ruderman. The same origin policy, 2001. *URL: http://www.mozilla.org/projects/security/components/same-origin.html*, 5(2):7–2.

[100] 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)*, 2011.

[101] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[102] J. Samuel. Requestpolicy 0.5.20. `https://www.requestpolicy.com/`, 2011.

[103] Kapil Singh, Alexander Moshchuk, Helen J Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 463–478. IEEE, 2010.

[104] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307. Springer, 2007.

[105] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930. ACM, 2010.

[106] 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.

[107] Mike Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 331–346. IEEE, 2009.

[108] Anne Van Kesteren et al. Cross-origin resource sharing. *W3C Working Draft WD-cors-20100727*, 2010.

[109] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *CSF*, 2014.

[110] Helen J Wang, Chris Grier, Alexander Moshchuk, Samuel T King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the gazelle web browser. In *USENIX Security Symposium*, volume 28, 2009.

[111] Joel Weinberger, Adam Barth, and Dawn Song. Towards Client-side HTML Security Policies. In *HotSec*, 2011.

[112] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[113] Sachiko Yoshihama, Takaaki Tateishi, Naoshi Tabuchi, and Tsutomu Matsumoto. Information-flow-based access control for web browsers. *IEICE transactions on information and systems*, 92(5):836–850, 2009.

[114] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *ACM SIGPLAN Notices*, volume 42, pages 237–249. ACM, 2007.

[115] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.