Control What You Include!
Server-Side Protection against Third Party Web Tracking

Dolière Francis Somé ✉, Nataliia Bielova, and Tamara Rezk

Université Côte d'Azur, Inria, France
{doliere.some,tamara.rezk,nataliia.bielova}@inria.fr

Abstract. Third party tracking is the practice by which third parties recognize users accross different websites as they browse the web. Recent studies show that more than 90% of Alexa top 500 websites [38] contain third party content that is tracking its users across the web. Website developers often need to include third party content in order to provide basic functionality. However, when a developer includes a third party content, she cannot know whether the third party contains tracking mechanisms. If a website developer wants to protect her users from being tracked, the only solution is to exclude any third-party content, thus trading functionality for privacy. We describe and implement a privacy-preserving web architecture that gives website developers a control over third party tracking: developers are able to include functionally useful third party content, the same time ensuring that the end users are not tracked by the third parties.

Keywords: Program rewriting techniques for security ·Security by design

1 Introduction

Third party tracking is the practice by which third parties recognize users accross different websites as they browse the web. In recent years, tracking technologies have been extensively studied and measured [28, 29, 31, 34, 36, 38] – researchers have found that third parties embedded in websites use numerous technologies, such as third-party cookies, HTML5 local storage, browser cache and device fingerprinting that allow the third party to recognize users across websites [39] and build browsing history profiles. Researchers found that more than 90% of Alexa top 500 websites [38] contain third party web tracking content, while some sites include as much as 34 distinct third party content [33].

But why do website developers include so many third party content (that may track their users)? Though some third party content, such as images and CSS [2] files can be copied to the main (first-party) site, such an approach has a number of disadvantages for other kinds of content. *Advertisement* is the base of the economic model in the web – without advertisements many website providers will not be able to financially support their website maintenance. *Third party JavaScript libraries* offer extra functionality: though copies of such libraries can be stored on the main first party site, this solution will sacrifice maintenance of these libraries when new versions are released. The developer would need to manually check the new versions. *Web mashups*, as for example applications that use hotel searching together with maps, are actually based on reusing third-party content, as well as maps, and would not be able to provide their basic functionality without including the third-party content. Including JavaScript libraries, content for mashups or advertisements means that the web developers cannot provide to the users the guarantee of non-tracking.

Except for ethical decision not to track users, from May 2018 the website owners will have a legal obligation as well. The ePrivacy directive (also know as 'cookie law') will be updated to the regulation, and will make website owners liable for third party tracking that takes place in their websites. This regulation will be applied to all the services that are delivered to the natural persons located in the European Union. This regulation will apply high penalties for any violation. Hence, privacy compliance will be of high interest to all website owners and developers, and today there is no automatic tool that can help to control third party tracking.To keep a promise of non-tracking, the only solution today is to exclude any third-party content[1], thus trading functionality for privacy.

In this paper, we present a new web application architecture that allows web developers to gain control over certain types of third party content. Our solution is based on the automatic rewriting of the web application in such a way that the third party requests are redirected to a trusted web server, with a different domain than the main site. This trusted web server may be either controlled by a trusted party, or by a main site owner – it is enough that the trusted web server has a different domain. A trusted server is needed so that the user's browser will treat all redirected requests as third party requests, like in the original web application. The trusted server automatically eliminates third-party tracking cookies and other technologies.

In summary our contributions are:

- A classification of third party content that can and cannot be controlled by the website developer.
- An analysis of third party tracking capabilities – we analyze two mechanisms: recognition of a web user, and identification of the website she is visiting [2].
- A new architecture that allows to include third party content in web applications and eliminate stateful tracking.
- An implementation of our architecture, demonstrating its effectiveness at preventing stateful third party tracking in several websites.

2 Background and Motivation

Third party web tracking is the ability of a third party to re-identify users as they browse the web and record their browsing history [34]. Tracking is often

[1] For example, see `https://duckduckgo.com/`. [2] Tracking is often defined as the ability of a third party to recognize a user through different websites. However, being able to identify the websites a user is interacting with is equally crucial for the effectiveness of tracking.

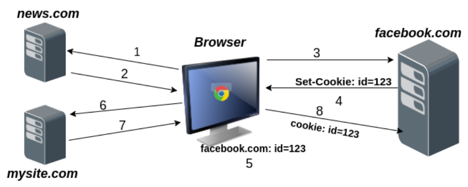


Fig. 1. Third Party Tracking

done with the purpose of web analytics, targeted advertisement, or other forms of personalization. The more a third party is prevalent among the websites a user interacts with, the more precise is the browsing history collected by the tracker. Tracking has often been conceived as the ability of a third party to recognize the web user. However, for successful tracking, each user request should contain two components:

User recognition is the information that allows tracker to recognize the user;
Website identification is the website which the user is visiting.

For example, when a user visits `news.com`, the browser may make additional requests to `facebook.com`. As a result, Facebook learns about the user's visit to `news.com`. Figure 1 shows a hypothetical example of such tracking where `facebook.com` is the third party.

Consider that a third party server, such as `facebook.com` hosts different content, and some of them are useful for the website developers. The web developer of another website, say `mysite.com`, would like to include such *functional* content from Facebook, such as Facebook "Like" button, an image, or a useful JavaScript library, but the developer does not want its users to be tracked by Facebook. If the web developer simply includes third party Facebook content in his application, all its users are likely to be tracked by cookie-based tracking. Notice that each request to `facebook.com` also contains an HTTP Referrer header, automatically attached by the browser. This header contains the website URL that the user is visiting, which allows Facebook to build user's browsing history profile.

The example demonstrates cookie-based tracking, which is extremely common [38]. Other types of third party tracking, that use client-side storage mechanisms, such as HTML5 LocalStorage, or cache, and device fingerprinting that do not require any storage capabilities, are also becoming more and more popular [29].

Web Developer Perspective. A web developer may include third party content in her webpages, either because this content *intentionally* tracks users (for example, for targeted advertising), or because this content is important for the functioning of the web application. We therefore distinguish two kinds of third party content from a web developer perspective: *tracking* and *functional*. *Tracking* content is *intentionally* embedded by website owner for tracking purposes. *Functional* content is embedded in a webpage for other purposes than tracking: for example, JavaScript libraries that provide additional functionality, such as jQuery, or other components, such as maps. In this work, we focus on *functional* content and investigate the following questions:

– What kind of third party content can be controlled from a server-side (web developer) perspective?
– How to eliminate the two components of tracking (user recognition and website identification) from functional third party content that websites embed?

2.1 Browsing Context

Browsers implement different specifications to securely fetch and aggregate third party content. One widely used approach is the *Same Origin Policy (SOP)* [15], a security mechanism designed for developers to isolate legacy content from potentially untrusted third party content. An origin is defined as scheme, host and port number, of the URL [21] of the third party content.

When a browser renders a webpage delivered by a first party, the page is placed within a *browsing context* [1]. A browsing context represents an instance of the browser in which a document such as a webpage is displayed to a user, for instance browser tabs, and popup windows. Each browsing context contains 1) a copy of the browser properties (such as browser name, version, device screen etc), stored in a specific object; 2) other objects that depend on the origin of the document according to SOP. For instance, the object `document.cookie` gives the cookies related to the domain and path of the current context.

In-Context and Cross-Context Content. Certain types of content embedded in a webpage, such as images, links, and scripts, are associated with the context of the webpage, and we call them *in-context* content. Other types of content, such as `<iframe>`, `<embed>`, and `<object>` tags are associated with their own browsing context, and we call them *cross-context* content. Usually, cross-context content, such as `<iframe>` elements, cannot be visually distinguished from the webpage in which they are embedded, however they are as autonomous as other browsing contexts, such as tabs or windows. Table 1 shows different third party content and their execution contexts.

Table 1. Third Party Content and Execution Context.

	HTML Tags	Third Party Content
in-context	`<link>`	Stylesheets
	`<img>`	Images
	`<audio>`	Audios
	`<video>`	Videos
	`<form>`	Forms
	`<script>`	Scripts
cross-context	`<(i)frame>, <frameset>, <a>`	Web pages
	`<object>, <embed>, <applet>`	Plugins and Web pages

The Same Origin Policy manages interactions between different browsing contexts. In particular, it prevents in-context scripts from interacting with cross-context iframes in case their origins are different. To communicate, they may use inter-frame communication APIs such as `postMessage` [12].

2.2 Third Party Tracking

In this work, we consider only stateful tracking technologies – they require an identifier to be stored client-side. The most common storage mechanism is cookies, but others, such as HTML5 LocalStorage and browser cache can also be used for stateful tracking. Figure 2 presents the well-known stateful tracking mechanisms. We distinguish two components necessary for successful tracking: user recognition and website identification. For each component, we describe the capabilities of in-context and cross-context. We also distinguish *passive tracking* (through HTTP headers) and *active tracking* (through JavaScript or plugin script).

	User Recognition		**Website Identification**	
	Passive	Active	Passive	Active

in-context	HTTP cookies
	Cache-Control
	Etag
	Last-Modified
		-	Referer
			Origin
				`document.URL`
				`document.location`
				`window.location`

cross-context		Flash LSOs
		`document.cookie`
		`window.localStorage`
		`window.indexedDB`
			Referer	`document.referrer`

Fig. 2. Stateful tracking mechanisms

In-Context Tracking. In-context third party content is associated with the browsing context of the webpage that embeds it (see Table 1).

Passively, such content may use HTTP headers to recognize a user and identify the visited website. When a webpage is rendered, the browser sends a request to fetch all third party content embedded in that page. The responses from the third party, along with the requested content, may contain HTTP headers that are used for tracking. For example, the *Set-cookie* HTTP header tells the browser to save third party cookies, that will be later on automatically attached to every request to that third party in the *Cookie* header. *Etag* HTTP header and other cache mechanisms like *Last-Modified* and *Cache-Control* HTTP headers may also be used to store user identifiers [39] in a browser. To identify the visited website, a third party can either check the *Referer* HTTP header, automatically attached by the browser, or an *Origin* header[3].

Actively, in-context third party content cannot use browser storage mechanisms, such as cookies or HTML5 Local Storage associated to the third party because of the limitations imposed by the SOP (see Section 2.1). For example, if a third party script from *third.com* uses `document.cookie` API, it will read the cookies of the main website, but not those of *third.com.* This allows tracking

[3] Origin header is also automatically generated by the browser when the third party content is trying to access data using Cross-Origin Resource Sharing [4] mechanism.

within the main website but does not allow tracking cross-sites [38]. For website identification, third party active content, such as scripts, can use several APIs, for example `document.location`.

Cross-Context Tracking. Cross-context content, such as iframe, is associated with the browsing context of the third party that provided this content.

Passively, the browser may transmit HTTP headers used for user recognition and website identification, just like in the case of in-context content. Every third-party request for cross-context content will contain the URL of the embedding webpage in its *Referer* header.

Requests to fetch third party content further embedded inside a cross-context (such as iframe) will carry, not the URL of the embedding webpage, but that of the iframe in their *Referer* or *Origin* headers (in the case of CORS requests). This prevents them from passively identifying the embedding webpage.

Actively, cross-context third party content can use a number of APIs to store user identifiers in the browser. These APIs include cookies (`document.cookie`), HTML5 LocalStorage (`document.localStorage`), IndexedDB, and Flash Local Stored Objects (LSOs). For website identification, `document.referrer` API can be used – it returns the value of the HTTP Referrer header transmitted in the request to the cross-context third party.

Combining In-Context and Cross-Context Tracking. Imagine a third party script from `third.com` embedded in a webpage – according to the context and to the SOP, it is in-context. If the same webpage embeds a third party iframe from `third.com` (cross-context), then because of SOP, such script and iframe cannot interact directly. However, the can still communicate through inter-frame communication APIs such as `postMessage` [12].

On one hand, the in-context script can easily identify the website using APIs such as `document.location`. On the other hand, the cross-context iframe can easily recognize the user by calling `document.cookie`. Therefore, if the iframe and the script are allowed to communicate, they can exchange those partial tracking information to fully track the user.

For example, a social widget, such as Facebook "Like" button, or Google "+1" button, may be included in webpages as a script. When the social widget script is executed on the client-side, it loads additional scripts, and new browsing contexts (iframes) allowing the third party to benefit from both in-context and cross-context capabilities to track users.

3 Privacy-preserving Web Architecture

For third party tracking to be effective, two capabilities are needed: 1) the tracker should be able to identify the website in which it is embedded, and 2) recognize the user interacting with the website. Disabling only one of these two capabilities for a given third party already prevents tracking. In order to mitigate stateful tracking (see Section 2), we make the following design choices:

1. **Preventing only user recognition for in-context**. As show in Table 2, in-context content cannot perform any active user recognition. We are left

with passive user recognition and (active and passive) website identification. Preventing passive user recognition for such content (images, scripts, forms) is possible by removing HTTP headers such as *Cookie, Set-cookie, ETag* that are sent along with requests/responses to fetch those content.
Note that it is particularly difficult to prevent active website identification because trying to alter or redefine `document.location` or `window.location` APIs, will cause the main page to reload. Therefore, in-context active content (scripts) can still perform active website identification. Nonetheless, since we remove their user recognition capability, tracking is therefore prevented for in-context content.

2. **Preventing only website identification for cross-context**. We prevent passive website identification by instructing the browser not to send the HTTP *Referer* header along with requests to fetch a cross-context content. Therefore, when the cross-context content gets loaded, the tracker is unable to identify the website in which it is embedded in. Indeed, executing `document.referrer` returns an empty string instead of the URL of the embedding page.
Because of the limitations of the SOP, a website owner has no control over cross-context third party content, such as iframes. Therefore, active and passive user recognition can still happen in third party cross-context. We discuss other possibilities to block some active user recognition APIs in Section 4.1. Nonetheless, since website identification is not possible, tracking is therefore prevented for cross-context third party content.
3. **Prevent communication between in-context and cross-context content**. Our architecture proposes a way to block such communications that can be done by `postMessage` API. We discuss the limitations of this approach in Section 4.1.

To help web developers keep their promises of non-tracking and still include third-party content in their web applications, we propose a new web application architecture. This architecture allows web developers to 1) automatically rewrite the URLs of all in-context third party content embedded in a web application, 2) redirect those requests to a trusted third party server which 3) remove/disable known *stateful* tracking mechanisms (see Section 2) for such content; 4) rewrite and redirect cross-context requests to the trusted third party so as to prevent website identification and communication with in-context scripts.

Figure 3 provides an overview of our web application architecture. It introduces two new components fully controlled by the website owner.

Rewrite Server (Section 3.1) acts like a reverse proxy [14] for the original web server. It rewrites the original web pages in such a way that all the requests to fetch all the third party content that they embed are redirected through the Middle Party Server before reaching the intended third party server.

Middle Party Server (Section 3.2) is at the core of our solution since it intercepts all browser third party requests, removes tracking, then forwards them to the intended third parties. From every response from a third party, the server removes tracking information and forwards the response back to the browser. For

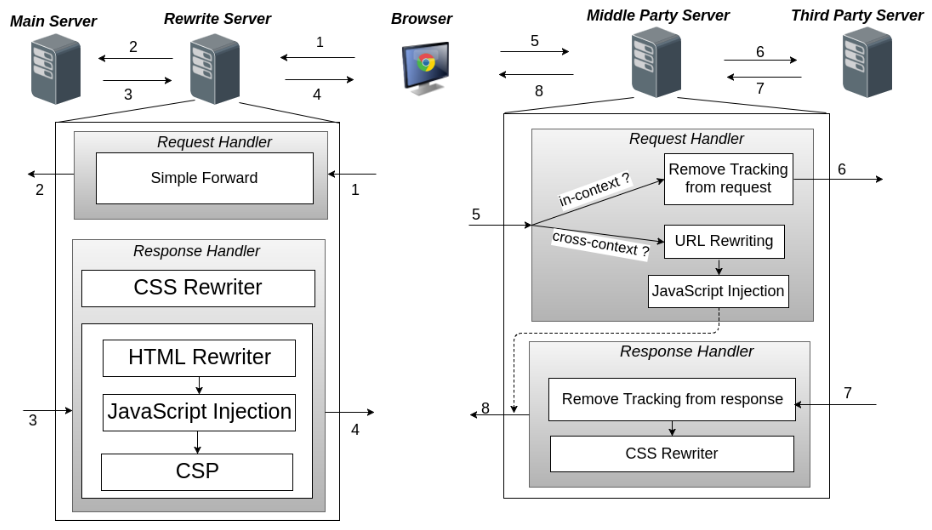


Fig. 3. Privacy-Preserving Web Architecture

in-context content such as images and scripts, the Middle Party Server prevents user recognition and website identification, while for cross-context content such as iframes, it prevents website identification and communication with other in-context scripts.

3.1 Rewrite Server

The goal of the Rewrite Server is to rewrite the original content of the requested webpages in such a way that all third party requests will be redirected to the Middle Party Server. It consists of three main components: static HTML rewriter for HTML pages, static CSS rewriter and JavaScript injection component. Into each webpage, we load a JavaScript code that insures that all dynamically generated third party content are redirected to the Middle Party Server as well.

HTML and CSS Rewriter rewrites the URLs of static third party content embedded in original web pages and CSS files in order to redirect them to the Middle Party Server. For example, the URL of a third-party script source `http://third.com/script.js` is written so that it is instead fetched through the Middle Party Server: `http://middle.com/?src=http://third.com/script.js`. The **HTML Rewriter** component is implemented using the Jsdom HTML parser [8], and **CSS Rewriter**, using the CSS parser [5] module for Node.js.

JavaScript Injection. The Rewrite Server also injects a script in all original webpages after they are rewritten. This script controls APIs used to dynamically inject content inside a webpage once the webpage is rendered in a browser. It is available at `https://webstats.inria.fr/sstp/dynamic.js`. Table 2 shows APIs that can be used to dynamically inject third party content within a webpage. They are controlled using the injected script.

A **Content Security Policy (CSP)** [44] is injected in the response header of each webpage in order to prevent third parties from bypassing the rewriting

Table 2. Injecting Dynamic Third Party Content

API	Content
`document.createElement`	inject content from Table 1
`document.write`	any content
`window.open`	Web pages(popups)
`Image`	images
`XMLHttpRequest`	any data
`Fetch, Request`	any content
`EventSource`	stream data
`WebSocket`	websocket data

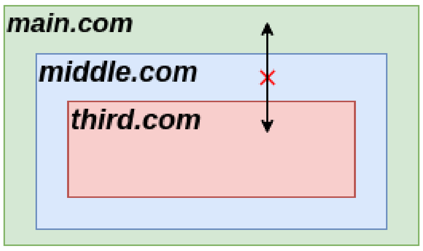


Fig. 4. Prevent Combining in-context and cross-context tracking

and redirection to the Middle Party Server. A CSP delivered with the webpage controls the resources of that page by specifying which resources are allowed to be loaded and executed. By limiting the resource origins to only those of the Middle Party Server and the website own domain, we prevent third parties from bypassing the redirection to the Middle Party Server in order to load content directly from a third party server. Such attempts will get blocked by the browser upon enforcement of the CSP of the page. The following listing gives the CSP injected in all webpages, assuming that *middle.com* is the domain of the Middle Party Server.

```
Content-Security-Policy: default-src 'self'
   middle.com; object-src 'self';
```

3.2 Middle Party

The main goal of the Middle Party is to proxy the requests and responses between browsers and third parties in order to remove tracking information exchanged between them. It functions differently for in-context and cross-context content.

In-Context content are scripts, images, etc. (see Table 1). Since a third party script from `http://third.com/script.js` is rewritten by the Rewrite Server to `http://middle.com/?src=http://third.com/script.js`, it is fetched through the Middle Party Server. This hides the third party destination from the browser, and therefore prevents it from attaching third party HTTP cookies to such requests. Because the browser will still attach some tracking information to the requests, then when the middle party receives a request URL from the browser, it takes the following steps. **Remove Tracking from request** that are set by the browser as HTTP headers. Among those headers are *Etag, If-Modified-Since, Cache-Control, Referer.* Next, it makes a request to the third party in order to get the content of the script `http://third.com/script.js`. **Remove Tracking from response** returned by the third party. The headers that the third party may send are *Set-Cookie, Etag, Last-Modified, Cache-Control.* **CSS Rewriter** rewrites the response if the content is a CSS file, in order to also redirect to the Middle Party Server any third party content that they may embed. Finally, the response is returned back to the browser.

Cross-context content are iframes, links, popups, etc. (see Table 1). The Middle Party Server prevents website identification for cross-context content and communication with in-context scripts. This is done by loading cross-context content from another cross-context controlled by the Middle Party Server as illustrated by Figure 4.

For instance, a third party iframe from `http://third.com/page.html` is rewritten to `http:// middle.com/?emb=http://third.com/page.html`. When the Middle Party Server receives such a request URL from the browser, it takes the following actions: **URL Rewriting.** Instead of fetching directly the content of `http://third.com/page.html`, the Middle Party Server generates a content in which it puts the URL of the third party content as a hyperlink `<a href = "http://third.com/page.html" rel = "noreferrer noopener"></a>`. The most important part of this content is in the `rel` attribute value. Therefore, `noreferrer noopener` instructs the browser not the send the *Referer* header when the link `http://third.com/page.html` is followed client-side. **JavaScript Injection** module adds a script to the content so that the link gets automatically followed once the content is rendered by the browser. Once the link is followed, the browser fetches the third party content directly on the third party server, without going through the Middle Party server anymore. Nonetheless, it does not include the *Referer* header for identifying the website. Therefore, the `document.referrer` API also returns an empty string inside the iframe context. This prevents it from identifying the website. The third party server response is placed within a new iframe nested within a context that belongs to the Middle Party, and not directly within the site webpage. This prevents in-context scripts and the cross-context content from exchanging tracking information as illustrated by Figure 4.

HTTPS Content. We recommend deploying the Middle Party Server as an HTTPS server. Therefore, third party content originally served over HTTPS (before rewriting) still get served over HTTPS even in the presence of the Middle Party Server . Moreover, third party content originally served over HTTP would get blocked by current browsers according to the Mixed Content policy [43]. With an HTTPS Middle Party, HTTP third party requests will not be prevented from loading since they are fetched over HTTPS through the Middle Party.

Multiple Redirections. A third party may attempt to circumvent our solution by performing multiple redirections. This is commonly used in advertisements (though ads are not in scope of this paper).

When a (third party) web server wants to perform a redirection to another server, it usually does so by including in the response, a special HTTP *Location* that indicates the server to which the next request will be sent. The Middle Party Server prevents such circumvention by rewriting the *Location* header so that the browser sends the next redirection request to the Middle Party Server again. As a result, all the redirections pass via the Middle Party.

4 Implementation

We have implemented both the Rewrite Server and the Middle Party Server as full Node.js [10] web servers supporting *HTTP(S)* protocols and web sockets. Implementation details are available at `https://webstats.inria.fr/sstp/`.

Rewrite Server. In our implementation, we deploy the Rewrite Server on the same physical machine as the original web application server. In order to do so, we moved the original server on a different port number, and the Rewrite Server on the initial port of the original server. Therefore, requests that are sent by browsers reach first the Rewrite Server. It then simply forwards them to the original server, which handles the request as usual and return a response to the Rewrite Server. Then, HTML webpages, and CSS files are rewritten using the **HTML Rewriter** and **CSS Rewriter** components respectively. To handle dynamic third party content, we inject a script. And in order to prevent malicious third parties from bypassing the redirection, we inject a CSP (See Section 3.1).

Middle Party. All requests to load third party contents embedded in a website deploying our architecture will go through the Middle Party Server. In-Context and cross-context contents are handled differently.

In-Context Contents are simply stripped off tracking information that they carry from the browser to the third parties and vice versa. See Section 3 for the list of tracking information that are removed from third party requests and responses. In particular, third party CSS responses are rewritten, using the **CSS Rewriter** component, to redirect to the Middle Party Server any third party content that they may further embed. As in the case of the Rewrite Server, this component is implemented using a CSS parser [5] for Node.js

Cross-Context contents are handled in a way that the original website identity is not leaked to them. They are also prevented from communicating with any in-context third party content to exchange tracking information. If the cross-context URL was `http://third.com/page.html`, instead of making a request to `third.com`, the Middle Party Server returns to the browser a response consisting of rewriting the URL to

```
<a href="http://third.com/page.html" rel="noreferrer
   noopener"></a>.
```

and injecting the following script:

```
var third_party = document.getElementsByTagName("a")[0];
if(window.top == window.self){
  third_party.target = "_blank";
  third_party.click();
  window.close();
}else{
  var iframe = document.createElement("iframe");
      iframe.name = "iframetarget";
  document.body.appendChild(iframe);
  third_party.target = "iframetarget";
  third_party.click();
}
```

Overall, when this response is rendered, the browser will not the *Referer* header to the third party, and the third party is prevented from communicating with in-context content, as explained in Section 3.2.

4.1 Discussion and Limitations

Our approach suffers from the following limitations. First, our implementation prevents cross-context and in-context contents from communicating with each other using `postMessage` API. However, in-context third party script can identify the website a user visits via `document.location.href` API. Then the script can include the website URL, say `http://main.com`, as a parameter of the URL of a third party iframe, for example `http://third.com/page.html?ref=http://main.com` and dynamically embed it in the webpage. In our architecture, this URL is rewritten and routed to the Middle Party. Since, the Middle Party Server does not inspect URL parameters, this information will reach the third party even though the *Referer* is not sent with cross-context requests.

Another limitation is that of dynamic CSS changes. For instance, changing the background image via the *style* object of an element in the webpage is not captured by the dynamic rewriting script injected in webpages. Therefore, if the image was a third party image, the CSP will prevent it from loading.

Performance Overhead. There is a performance cost associated with the Rewrite Server, which can be evaluated as the cost of introducing any reverse proxy to a web application architecture (See Section 3.1). Rewriting contents server-side and browser-side is also expensive in terms of performance. We believe that server-side caching mechanisms, in particular for static webpages, may help speed up the responsiveness of the Rewrite Server.

The Middle Party Server may also lead to performance overhead especially for webpages with numerous third party contents. Therefore, it can be provided as a service by a trusted external party, as it is the case for Content Distribution Networks (CDNs) serving contents for many websites.

Extension to Stateless Tracking. Even though this work did not address stateless tracking, such as device fingerprinting, our architecture already hides several fingerprintable device properties and can be extended to several others: 1) The redirection to the Middle Party anonymizes the real IP addresses of users;

2) Some stateless tracking APIs such as `window.navigator`, `window.screen`, and `HTMLCanvasElement` can be easily removed or randomized from the context of the webpage to mitigate in-context fingerprinting.

Possibility to Blocking Active User Recognition in Cross-Context. With the prevalence of third party tracking on the web, we have shown the challenges that a developer will face towards mitigating that. The sandbox attribute for iframes help prevent access to security-sensitive APIs. As tracking has become a hot concern, we suggest that similar mechanisms can help first party websites tackle third party tracking. The sandbox attribute can for instance be extended with specific values to tackle tracking. Nonetheless, the sandbox attribute can be used to prevent cross-context from some stateful tracking mechanisms [9].

5 Evaluation and Case Study

Demo website. We have set up a demo website that embeds a collection of third party content, both in-context and cross-context. In-context content include images, HTML5 audio and video, and a Google Map which further loads dynamic content such as images, fonts, scripts, and CSS files. A Youtube video is embedded as cross-context content in an iframe. The demo website is deployed at `https://sstp-rewriteproxy.inria.fr`. With the deployment of our solution, there is no change from a user perspective on how the demo website is accessed. Indeed, it is still accessible at `https://sstp-rewriteproxy.inria.fr`. However from the server-side, it is the Rewrite Server which is now running at `https://sstp-rewriteproxy.inria.fr` instead of the original server. It then intercepts user requests and forwards them to the original server which has been moved on port 8080 (http://sstp-rewriteproxy.inria.fr:8080), hidden from users and the outside.

The Middle Party Server runs at `https://sstp-middleparty.inria.fr`. With our architecture deployed, all requests to fetch third party content embedded in the demo website are redirected to the Middle Party Server. For in-context content, its removes any tracking information in the requests sent by the browser. Then it forwards the requests to the third parties. Any tracking information set by the third parties in the responses are also removed before being forwarded to the browser. For the cross-context content (Youtube Video in our demo), it is not directly loaded as an iframe inside the demo page. Instead, an iframe from the Middle Party Server is created and embedded inside the demo webpage. Then the Youtube video is automatically loaded in another iframe inside this first iframe which context is that of Middle Party Server. During this process, the *Referer* header is not leaked to Youtube (Section 3.2), preventing it from identifying the demo website in which it is included. In the Appendix, we show a screenshot of redirection requests to the Middle Party Server.

Real websites. Since we did not have access to real websites, we could not install the Rewrite Server and evaluate our solution on them. We therefore implemented a browser proxy based on a Node.js proxy [11], and included all the logic of the Rewrite Server within the proxy. The proxy was deployed at `https://sstp-rewriteproxy.inria.fr:5555` and acts like the Rewrite Server for real websites intercepting and forwarding requests to them, and rewriting the responses in order to redirect them to our Middle Party Server deployed at `https://sstp-middleparty.inria.fr`.

We then evaluated our solution on different kinds of websites: a news website `http://www.bbc.com`, an entertainment website `http://www.imdb.com`, and a shopping website `http://verbaudet.fr`. All three websites load content from various third party domains. Visually, we did not notice any change in the behaviors of the websites. We also interacted with them in a standard way (clicking on links on a news website, choosing products and putting them in the basket on the shopping website) and the main functionalities of the websites were preserved.

Overall, these evaluation scenarios have helped us improve the solution, especially rewriting dynamically injected third party content. We believe that this implementation will get even mature in the future when we will be able to convince some website owners to deploy it.

Limitations of the evaluation on real websites.

The evaluation on the real websites may break some features or introduce performances issues. Here, we discuss such problems and how to prevent them.

Third party identity (OpenID) providers such as Facebook or Google need to use third party cookies in order to be able to authenticate users to websites embedding them. Therefore, stripping off cookies can prevent users from successfully logging in to the related websites. In a deployment scenario, we make it possible for the developer to instruct the Rewrite Server not to rewrite such third party identity provider content so that users can still log in.

Furthermore, it is common for websites to rely on Content Distribution Networks (CDNs), from which they load content for performance purposes. Therefore, rewriting and redirecting CDNs requests to the Middle Party Server can introduce performance issues. In this case also, a developer can declare a list of CDNs which requests should not be rewritten by the Rewrite Server.

Finally, as one may have noticed, the real websites we have considered in our evaluation scenario are all HTTP websites. We could not evaluate our solution on real HTTPS websites because HTTPS requests and responses that arrive at the browser proxy are encrypted. Therefore, we could not be able to rewrite third party content that are embedded in such websites.

6 Related Work

A number of studies have demonstrated that third party tracking is very prevalent on the web today and analyzed the underlying tracking technologies [29,31, 34,38]. Lerner et al. [33] analyzed how third party tracking evolved for a period of twenty years. Trackers have been categorized according to either their business relationships with websites [34], their prominence [29,31] or the user browsing profile that they can build [38]. Mayer and Mitchell [34] grouped tracking mechanisms in two categories called stateful (cookie-based and super-cookies) and stateless (fingerprinting). It is rather intuitive to convince ourselves about the effectiveness of a stateful tracking, since it is based on unique identifiers that are set in users browsers. Nonetheless, the efficacy of stateless mechanisms has been extensively demonstrated. Since the pioneer work of Eckersley [28], new fingerprinting methods have been revealed in the literature [22–24,26,27,29,36,40,41]. A classification of fingerprinting techniques is provided in [42]. Those studies have contributed to raising public awareness of tracking privacy threats. Mayer and Mitchell [34] have shown that users are very sensitive to their online privacy, thus hostile to third party tracking. Englehardt et al. [30] have demonstrated that tracking can be used for surveillance purposes. The success of anti-tracking defenses is yet another illustration that users are concerned about tracking [35].

There are a number of defenses that try to protect users against third party tracking. First, major browser vendors do natively provide mechanisms for users to block third party cookies or browse in private/incognito mode for instance. More and more browsers even take a step further, considering privacy as a design principle: Tor Browser [17], TrackingFree [37], Blink [32], CLIQZ [3].

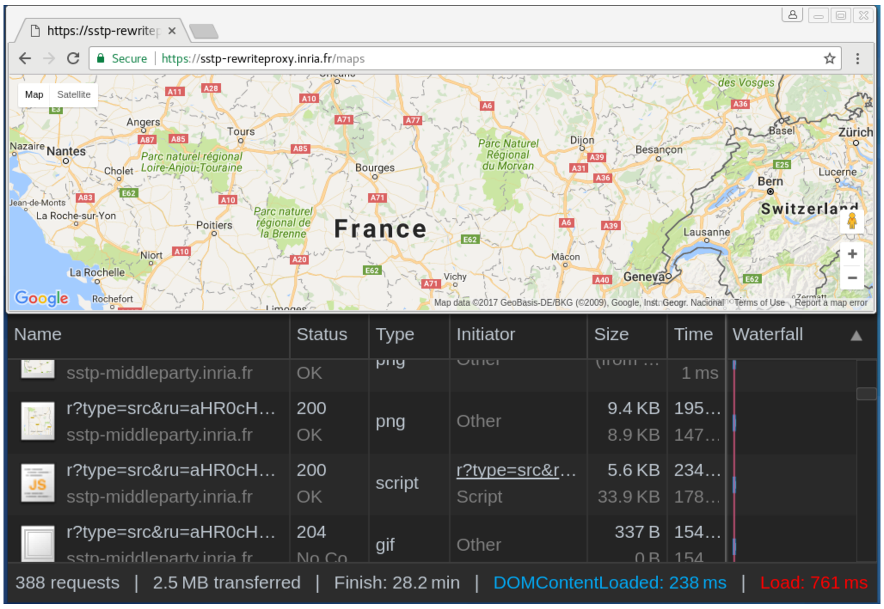
But the most popular defenses are browser extensions. Being tightly integrated into browsers, they provide additional privacy features that are not natively implemented in browsers. Well known privacy extensions are Disconnect [6], Ghostery [7], ShareMeNot [38], which is now part of PrivacyBadger [13], uBlock Origin [20] and a relatively new MyTrackingChoices [25]. Merzdovnik et al. [35] provide a large-scale evaluation of these anti-tracking defenses. Well known trackers such as advertisers, which businesses heavily depend on tracking, have also been taking steps towards limiting their own tracking capabilities [34]. The W3C is pushing forward the Do Not Track standard [18, 19] for users to easily express their tracking preferences so that trackers may comply with them. To the best of our knowledge, we are the first to investigate how a website owner can embed third party content while preventing them from accidentally tracking users. The idea of proxying requests within a webpage is inspired by web service workers API [16], though this API is still a working draft which is currently being tested in Mozilla Firefox and Google Chrome.

7 Conclusions

Most of the previous research analyzed third party tracking mechanisms, and how to block tracking from a user perspective. In this work, we classified third party tracking capabilities from a website developer perspective. We proposed a new architecture for website developers that allows to embed third party content while preserving users privacy. We implemented our solution, and evaluated it on real websites to mitigate stateful tracking.

Appendix

Screenshots of the demo website map console.



Name	Status	Type	Initiator	Size	Time	Waterfall
sstp-middleparty.inria.fr	OK	png	Other	(from ...	1 ms	
r?type=src&ru=aHR0cH... sstp-middleparty.inria.fr	200 OK	png	Other	9.4 KB 8.9 KB	195... 147...	
r?type=src&ru=aHR0cH... sstp-middleparty.inria.fr	200 OK	script	r?type=src&r... Script	5.6 KB 33.9 KB	234... 178...	
r?type=src&ru=aHR0cH... sstp-middleparty.inria.fr	204 No Co	gif	Other	337 B 0 B	154... 154	

388 requests | 2.5 MB transferred | Finish: 28.2 min | DOMContentLoaded: 238 ms | Load: 761 ms

Fig. 5. A demo page displaying a Google Maps

References

1. Browsing Contexts, https://www.w3.org/TR/html51/browsers.html
2. Cascading Style Sheets, https://www.w3.org/Style/CSS/
3. CLIQZ, https://cliqz.com
4. Cross-origin-resource sharing, https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS
5. CSS Parser for Node.js, https://github.com/reworkcss/css
6. Disconnect, https://disconnect.me/
7. Ghostery, https://www.ghostery.com/
8. HTML Parser for Node.js, https://github.com/tmpvar/jsdom
9. Iframe Sandbox Attribute, https://www.w3.org/TR/2011/WD-html5-20110525/the-iframe-element.html#attr-iframe-sandbox
10. Node.js, https://nodejs.org/en/
11. Node.js Proxy, https://newspaint.wordpress.com/2012/11/05/node-js-http-and-https-proxy
12. PostMessage - Cross-Origin Iframe Secure Communication, https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage
13. Privacy Badger, https://www.eff.org/fr/privacybadger
14. Reverse Proxy, https://en.wikipedia.org/wiki/Reverse_proxy
15. Same Origin Policy, https://www.w3.org/Security/wiki/Same_Origin_Policy
16. Service Worker API, https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
17. Tor Browser, https://www.torproject.org/projects/torbrowser/design/
18. Tracking Compliance and Scope - https://www.w3.org/TR/tracking-compliance/
19. Tracking Preference Expression, https://www.w3.org/TR/tracking-dnt/
20. uBlock Origin, https://www.ublock.org/

21. URL, https://www.w3.org/TR/url
22. Abgrall, E., Traon, Y.L., Monperrus, M., Gombault, S., Heiderich, M., Ribault, A.: XSS-FP: browser fingerprinting using HTML parser quirks. CoRR (2012)
23. Acar, G., Eubank, C., Englehardt, S., Juárez, M., Narayanan, A., Díaz, C.: The web never forgets: Persistent tracking mechanisms in the wild. In: Proc. of CCS 2014
24. Acar, G., Juárez, M., Nikiforakis, N., Díaz, C., Gürses, S.F., Piessens, F., Preneel, B.: FPDetective: dusting the web for fingerprinters. In: Proc. of CCS 2013
25. Achara, J.P., Parra-Arnau, J., Castelluccia, C.: Mytrackingchoices: Pacifying the ad-block war by enforcing user privacy preferences. CoRR (2016)
26. Boda, K., Földes, Á.M., Gulyás, G.G., Imre, S.: User tracking on the web via cross-browser fingerprinting. In: Proc. of the 16th NordSec. pp. 31–46 (2011)
27. Cao, Y., Li, S., Wijmans, E.: (cross-)browser fingerprinting via os and hardware level features. In: Proc. of the 24th NDSS (2017)
28. Eckersley, P.: How Unique Is Your Web Browser? In: Proc. of the 2010 PETS
29. Englehardt, S., Narayanan, A.: Online tracking: A 1-million-site measurement and analysis. In: Proc. of the 2016 CCS. pp. 1388–1401 (2016)
30. Englehardt, S., Reisman, D., Eubank, C., Zimmerman, P., Mayer, J., Narayanan, A., Felten, E.W.: Cookies that give you away: The surveillance implications of web tracking. In: Proc. of the 24th WWW. pp. 289–299 (2015)
31. Krishnamurthy, B., Wills, C.E.: Privacy diffusion on the web: a longitudinal perspective. In: Proc. of the 18th WWW. pp. 541–550 (2009)
32. Laperdrix, P., Rudametkin, W., Baudry, B.: Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In: Proc. of IEEE SP 2016
33. Lerner, A., Simpson, A.K., Kohno, T., Roesner, F.: Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In: Proc. of the 25th USENIX Security. Austin, TX (2016)
34. Mayer, J.R., Mitchell, J.C.: Third-party web tracking: Policy and technology. In: Proc. of the 2012 IEEE SP. pp. 413–427 (2012)
35. Merzdovnik, G., Huber, M., Buhov, D., Nikiforakis, N., Neuner, S., Schmiedecker, M., Weippl, E.: Block me if you can: A large-scale study of tracker-blocking tools. In: Proc. of the 2nd EuroSP. Paris, France (2017)
36. Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In: Proc. of the 2013 IEEE SP. pp. 541–555 (2013)
37. Pan, X., Cao, Y., Chen, Y.: I do not know what you visited last summer: Protecting users from stateful third-party web tracking with trackingfree browser. In: Proc. of the 22nd NDSS (2015)
38. Roesner, F., Kohno, T., Wetherall, D.: Detecting and defending against third-party tracking on the web. In: Proc. of the 9th NSDI. pp. 155–168 (2012)
39. Soltani, A., Canty, S., Mayo, Q., Thomas, L., Hoofnagle, C.J.: Flash Cookies and Privacy. In: AAAI spring symposium: intelligent information privacy management. pp. 158–163 (2010)
40. Starov, O., Nikiforakis, N.: Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In: Proc. of the 2017 WWW
41. Takei, N., Saito, T., Takasu, K., Yamada, T.: Web browser fingerprinting using only cascading style sheets. In: Proc. of the 10th BWCCA. pp. 57–63 (2015)
42. Upathilake, R., Li, Y., Matrawy, A.: A classification of web browser fingerprinting techniques. In: Proc. of the 7th NTMS. pp. 1–5 (2015)
43. West, M.: Mixed Content (2016), https://www.w3.org/TR/mixed-content/
44. West, M., Barth, A., Veditz, D.: Content Security Policy Level 2 (2015)