Universität des Saarlandes
MI Fakultät für Mathematik und Informatik
Department of Computer Science

Master Thesis

Escaping the Cookie Prison: An In-Depth Analysis of Storage Access API Usage on the Web

submitted by

Philipp Baus

on January 3, 2025

Reviewers

Dr. Ben Stock

Dr. Giancarlo Pellegrino

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, January 3, 2025,

(Philipp Baus)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, January 3, 2025,

(Philipp Baus)

Abstract

Web browsers have implemented State Partitioning to prevent the misuse of cookies for cross-site tracking and other cross-site leak techniques. Although effective in enhancing user privacy, this approach disrupts the functionality of websites that rely on third-party cookies for seamless operation across different sites. In response, the Storage Access API was introduced, allowing framed documents to bypass State Partitioning and regain unpartitioned access to cookies that were set in a first-party context. While this provides framed third-party content with means to recover their original functionality, it also reintroduces privacy concerns by enabling the potential for cross-site tracking once again.

In our empirical study, we crawled the web to assess the adoption and usage of the SAA by collecting real-world data across various sites. To evaluate the privacy risks, we also developed a new metric, the Tracking Risk Score, using cookie classifications to quantify the likelihood of a site tracking its users across different websites. Our findings reveal that while the API is moderately used across the web, its adoption is significantly driven by a small number of heavily reused scripts. We also observed notable differences in usage patterns across regions, site popularity, and the primary content of crawled sites, alongside a modest growth in API adoption over the duration of our study. Regarding privacy risks, our analysis indicates that the overall risk remains low, with only one framed site exhibiting a significant likelihood of employing cross-site tracking mechanisms through the API. Nevertheless, even in this case, the site's ability to collect extensive tracking data is limited, as it is framed on only a small fraction of the crawled sites.

Acknowledgements

I would like to express my deepest appreciation to my advisor, Jannis Rautenstrauch, who continuously supported me throughout the writing of this thesis. I am especially grateful for the weekly meetings, during which he patiently addressed my questions and provided invaluable guidance.

Next, I wish to extend my heartfelt gratitude to my parents, Vera Baus and Volker Baus, as well as my late godfather, Werner Recktenwald, for their unwavering support and encouragement. Their guidance and belief in me, especially during my studies, have been instrumental in paving the way for this thesis.

I also want to sincerely thank the reviewers, Dr. Ben Stock and Dr. Giancarlo Pellegrino, for dedicating their time to reading and evaluating this thesis.

Furthermore, I would like to acknowledge the contribution of my proofreaders, Elisa Kalms, Maike Kalms, Thomas Helbrecht, Ole Heydt and Mika Meyer whose efforts greatly helped in improving the clarity and quality of this work.

Finally, I want to note that the tools ChatGPT and Grammarly were used to enhance the language of this thesis. ChatGPT was also employed to assist in generating code for creating visually appealing figures.

Contents

Chapter 1

Introduction

In the current digital environment, where online activities are seamlessly integrated across different platforms and websites, the importance of user privacy and data security has increased significantly. Among the most widespread threats to user privacy belongs cross-site tracking [1–6], which takes advantage of data or metadata that is accessible across different websites to identify users, leak information about them, and consequently undermine their privacy. Back in 2015, research by Li et al. already indicated that approximately 46% of the websites listed in the Alexa top 10,000 included at least one third-party tracker, with Google emerging as the leading tracking party on 25% of these sites [7]. There are various techniques that trackers can use to follow users around the web [1, 8], ranging from storage-based techniques to fingerprinting. However, despite the great amount of capabilities available to trackers, they usually settle for popular storage-based approaches such as cookies, as shown by previous research [4, 8–12].

To address this issue, modern web browsers have implemented State Partitioning techniques aimed at improving user privacy and mitigating the risks associated with user tracking [13–16]. By partitioning storage between different top-level sites through multiple partitioning keys, third-party cookies as well as other data in the browser will only be accessible on the site that the user visited when the data was stored. This approach effectively limits the scope of tracking activities to the currently visited website and prevents unauthorized access to sensitive information across different sites.

Although there have been notable advances in safeguarding user privacy through State Partitioning, it is important to acknowledge that privacy-preserving mechanisms often come with concerns in terms of compatibility [2, 9, 17–20]. Storage partitioning could potentially disrupt benign websites that depend on accessing cookies across different sites, such as single sign-on mechanisms, third-party payments, or cross-domain analytics. To address these problems, the Storage Access API [21–23] (SAA), a new mechanism to

bypass the storage partitioning of cookies, was designed and introduced in most major browsers [23–25]. The API enables framed websites to request the disabling of cookie partitioning for the framed document and subsequently regain unpartitioned access to all cookies that were stored for the framed site when it was loaded in a first-party context [21–23]. Although the API enables legitimate use cases, if not thoroughly considered and implemented, it could once again give trackers access to third-party cookies and expose users to the potential risks of cross-site tracking. This is also one of the reasons why the Brave browser opted to completely disable the Storage Access API, stating that 'this approach is not the right way to protect privacy on the web' [26].

Due to security and privacy concerns of the API, Google also performed a security audit on the Storage Access API in September 2022 that revealed flaws in the design of the API [27], which led to improvements in the official work item [21]. However, despite these efforts to improve the API's security and privacy, a significant gap remains: no independent measurements or investigations have been carried out to analyze the real-world usage and impact of the Storage Access API across various websites. This lack of empirical research also means that potential misuse of the API remains largely unexamined. Nevertheless, understanding the compatibility-privacy trade-off is essential as the adoption and usage of the Storage Access API across the web have significant implications for both user privacy and web functionality.

In order to reveal the usage of the Storage Access API on the web and evaluate the potential risks associated with it, we performed an empirical study. For this purpose, we collected real-world data on a large scale by employing a fully automated web crawler on a randomly sampled set of websites. This crawler systematically gathered data on the usage of the Storage Access API across the web, allowing us to analyze its prevalence and associated privacy implications. Through this approach, we aim to provide a comprehensive overview of how the Storage Access API is used and to identify potential risks related to data privacy and user tracking.

1.1 Research Questions

While the security audit of the Storage Access API mentioned in the previous section offered a theoretical analysis for enhancing the security and privacy of the SAA, the primary goal of this thesis is to assess the real-world privacy implications of the API. To achieve this, we aim to investigate the usage of the Storage Access API and its associated privacy concerns as of the time this thesis by performing an empirical study. We try to answer the following research questions:

(RQ1) What is the prevalence of Storage Access API usage on the web?

(RQ2) Are there any differences in usage between different popularities, regions and primary content of the crawled sites?

(RQ3) What are the privacy risks associated with Storage Access API usage? Is there a high risk of third-parties tracking users once again?

(RQ4) How does the Storage Access API landscape evolve over time?

1.2 Contribution

This thesis contributes to a better transparency of Storage Access API usage on the web by:

- Implementing a fully automated web crawler that collects data about the usage of the SAA on a predefined set of sites.
- Introducing the Tracking Risk Score (TRS), a formula to measure the tracking risk of sites that include the SAA to request unpartitioned cookie access.
- To the best of our knowledge, performing the first large-scale measurement of SAA usage on the web including its privacy risks.

1.3 Outline

First of all, we present the background needed to fully understand the concept of the SAA and its possible privacy implications in Chapter 2. Following, in Chapter 3 we also mention related work in connected research areas. Next, in Chapter 4 we explain the methodology of our study, explaining the functionalities of our web crawler, as well as introducing the Tracking Risk Score formula that we use to evaluate the tracking risk of a site utilizing the SAA to request unpartitioned cookie access. In the next two chapters, Chapter 5 and Chapter 6 we further discuss our pre-study evaluation of different crawling techniques together with the scope of our study, providing an overview of the data collection methods and the data that we later analyzed to uncover the SAA usage and its privacy implications. The results of this analysis are subsequently presented in Chapter 7, Chapter 8, Chapter 9 and Chapter 10. At the end, we also discuss our findings in Chapter 11 and conclude the results of our work in Chapter 12.

Chapter 2

Background

This chapter provides the background information necessary to understand the reasons behind the introduction of the Storage Access API, its functionality, and the privacy risks it entails. First, we will look at how cookies work, how they can be used to track users online across different sites, and how browsers took action against these tracking attempts, as it is an important part of the adoption of the Storage Access API. Next, we will continue by digging into the functionality of the API and its integration's into other browser APIs. Finally, in order to get a feeling of the privacy risks that the API entails, we will also dive into the privacy and security considerations, changes to the standard over time, and the implementation differences in the different browsers. Along the way, we will also mention extensions to the API standard, as well as alternatives that sites can use to receive unpartitioned cookie access.

For the rest of the thesis, we use the term "site" to refer to a resource's registrable domain or eTLD+1. Thus, if two resources are hosted under the same registrable domain, they are considered same-site. Additionally, when discussing the origin of an object, we refer, unless otherwise specified, to the combination of protocol (or scheme), domain, and port. If all these values match, the websites are considered same-origin.

2.1 Cookies

In the early days of the web, Internet browsers consisted only of stateless Hypertext Markup Language (HTML) pages, since there were no means to persist user state on the client side. However, this changed dramatically when Netscape implemented and deployed cookies in 1994 as part of the Mosaic browser [28, 29] which introduced the possibility to store small data pieces on the client-side. A few years later, the *Network*

Attribute	Description	Type
Max-Age	Set the time in seconds until the cookie expires.	Option
Expires	Set the expiry date of the cookie.	Option
Domain	Set the domain for the cookie. The cookie will also be sent to all subdomains.	Option
Path	Restricts the cookie access to a specific path and its sub-paths.	Option
SameSite	Specifies whether a cookie can be sent in a cross-site request. Possible values are *Strict*, *Lax*, and *None*.	Option
Secure	The cookie will only be sent in HTTPS requests.	Flag
HttpOnly	The cookie will not be accessible via JavaScript.	Flag

Table 2.1: List of common cookie attributes

Working Group adopted Hypertext Transfer Protocol (HTTP) cookies as an Internet standard with the specification "HTTP State Management Mechanism", also known as *RFC 2109* [30]. After its introduction in 1997, the document was revised twice, in 2000 [31] and 2011 [32], with the current version published as *RFC 6265*. However, since 2017, the *Internet Engineering Task Force* is working on a draft revision called *RFC 6265bis* which seeks to improve the handling of cookies, particularly with respect to security and privacy [33].

The specifications introduced two HTTP headers, *Cookie* and *Set-Cookie*, which allow one to store and retrieve small pieces of data, consisting of a name and a value, on the client side. When a web server responds to a user's request, it can use the *Set-Cookie* header to instruct the browser to store the cookie on the client side. In addition to the cookie name and value, the *Set-Cookie* header can also contain other attributes separated by a semicolon. These attributes can either be flags consisting only of the attribute name or options that have an additional value to their name. The purpose of these attributes is to allow a site to manage the access restrictions for a cookie by either loosening the policies, for example, to make it accessible on subdomains, or hardening it in order to prevent malicious parties from intercepting it through vulnerabilities such as Cross-Site Scripting (XSS). Table 2.1 shows common cookie attributes and their meaning. Once the cookie is stored by the browser, subsequent requests to the same site will include all matching cookies stored for that site using the *Cookie* header. The cookie attributes are omitted, only the name and value of the cookie are present.

Using these two headers, websites can manage state within their pages by linking an identifier to a particular state and storing it on the client side. Once the user visits the website again, the site will retrieve the cookie and its value again and can dynamically generate the website content based on the state linked to the identifier stored in the

```
Set-Cookie: session=1234; Domain=site.example; Secure; HttpOnly;
Set-Cookie: apiKey=5678; Domain=api.site.example; Secure;
            HttpOnly;
```

Listing 2.1: Example *Set-Cookie* header

```
Cookie: session=1234; apiKey=5678
```

Listing 2.2: Example *Cookie* header

cookie. This facilitates a wide variety of use cases, including managing user sessions, storing site preferences, or showing personalized ads.

Example Suppose that a user visits and logs into *site.example*. After the authentication succeeds, the website returns two cookies as shown in Listing 2.1, one for the user session on the main site and another for API access under *api.site.example*. Both cookies are security sensitive, as they authenticate the user and thus have the *Secure* and *HttpOnly* flags to protect them from being accessed by malicious parties. Subsequently, the user now visits *api.site.example*. Listing 2.2 shows the respective *Cookie* header that is attached to the request. Both cookies specified the Domain attribute when they were stored, allowing them to be sent to all subdomains of their respective values. As a result, both cookies match the *api.site.example* subdomain and are included in the HTTP request header.

2.1.1 First-/Third-Party Cookies

Cookies can be classified into two distinct categories:

(1) First-Party Cookies

(2) Third-Party Cookies / Cross-Site Cookies

The classification of cookies into first-party and third-party is determined by the browsing context. A cookie is considered a first-party cookie **(1)** when it is stored or retrieved by the website currently visited by the user [34], which we also refer to as the top-level context in this thesis. Conversely, third-party cookies **(2)** are set or retrieved by a site different from the one the user is actively visiting [34, 35]. For instance, websites may include cross-site resources, such as images, scripts, or iframes, which can store new cookies through their responses. In this case, these cookies are associated with the cross-site origin and are thus categorized as third-party cookies. However, if a user later visits the third-party site directly on the top-level in a first-party context, those same cookies will be considered first-party cookies. Conversely, if a user initially visits the third-party site directly, any cookies set during that session would be first-party cookies.

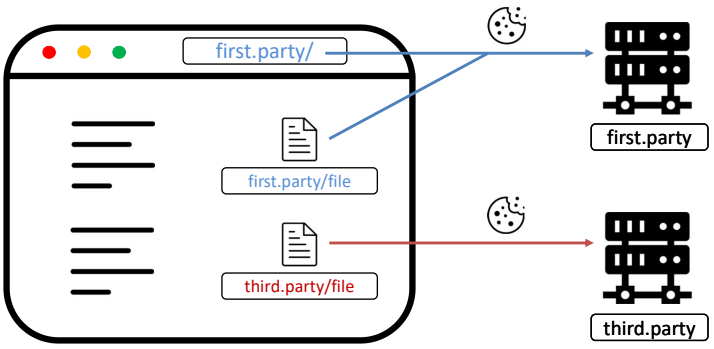


Figure 2.1: Distinction between first- and third-party cookies

When the user later accesses the same site as a third-party resource from another website, those cookies would then be considered third-party. While the classification does not directly affect cookie functionality, some browsers restrict access to third-party cookies through storage partitioning. More details on this are given in Section 2.3.

2.1.2 Functional Classification of Cookies

The classification of cookies plays an important role in helping organizations meet the EU General Data Protection Regulation (GDPR) [36], as well as the California Consumer Privacy Act (CCPA) [37]. Under these legislation, organizations are required to provide clear and sufficient information about their use of cookies, enabling users to make informed choices about which cookies they accept or reject. While they do not require a specific method for presenting this information, the use of a standardized classification system aids in conveying it.

One widely recognized classification system, developed by the UK International Chamber of Commerce (ICC), categorizes cookies into four distinct types based on their functions and purposes [38]:

1. **Strictly Necessary Cookies:** These cookies enable core services such as page navigation or access to secure areas and are essential for basic website functionality. These cookies generally do not require user consent under the GDPR, as they are necessary for the operation of the website.

2. **Performance Cookies:** Used to collect data on how visitors interact with a website, these cookies inform improvements to site performance and user experience. The GDPR requires that users be informed about these cookies and that consent is obtained before using them.

3. **Functionality Cookies:** Designed to remember user preferences and settings, these cookies enhance personalized website experiences. Since they are not essential to basic functionality, the GDPR mandates clear information disclosure and user consent.

4. **Targeting / Advertising Cookies:** These cookies track users' online behavior across websites to display targeted advertising. Because they involve personal data and tracking, the GDPR imposes the strictest consent requirements for these cookies.

Although the ICC classification system is not legally binding, it provides a framework that promotes a shared, non-technical language for explaining cookies to users. Furthermore, research can leverage these cookie categories to classify websites based on their cookie usage patterns. By analyzing cookie practices using this framework, researchers can assess the extent to which websites prioritize user privacy or are reliant on intensive data collection. For example, sites that primarily use necessary cookies may be seen as prioritizing user privacy, while those with a heavy reliance on targeting cookies may have a greater risk for user tracking and data sharing.

2.2 Cross-Site Tracking

Although cookies revolutionized the web by enabling the creation of stateful websites, unfortunately, this approach also has its downsides. The mechanism, especially due to third-party cookies, can be used to track users and observe their browsing behavior across different websites, posing a huge threat to user privacy [3, 5]. If websites include a third-party resource from a tracking party, this resource can store a unique identifier in a third-party cookie that is linked to the user visiting the website. Therefore, when the same user visits another website that also includes the same tracking party, the identifier cookie will be sent to the tracking party, allowing them to record the access to both websites and connect it to the associated user (see Figure 2.2). This technique, for example, allows for the construction of an interest profile of the user and is therefore often used to show targeted ads that fit the interests of the user according to the visited websites.

When multiple tracking parties work together, they can even share the unique user identifiers between each other, allowing them to link and match their data on a specific user across various websites or platforms [10, 39, 40]. Through the exchange of user identifiers with other trackers, they can extend their tracking capabilities, allowing them to monitor user behavior across multiple, previously unconnected website categories and

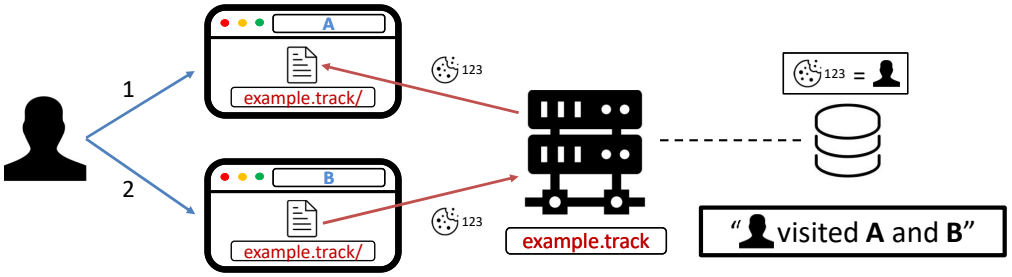


Figure 2.2: Cross-Site Tracking scenario

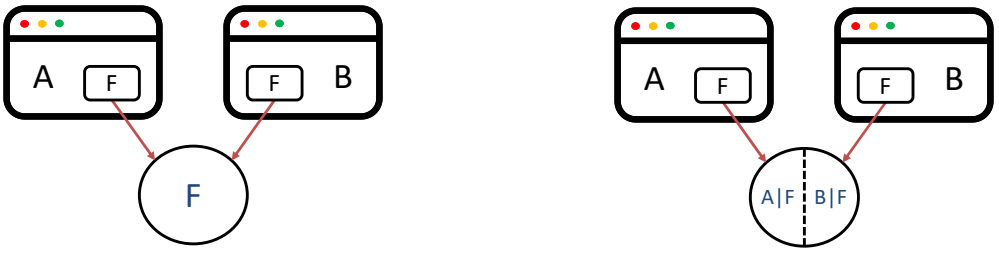
learn more about the interests of a user. However, for this synchronization to occur, the different trackers must have at least one common website or interaction point where they can track the same user. This shared tracking point allows the trackers to exchange and match their identifiers, making it possible to align their data. Without such a common site, trackers would be unable to connect their unique user identifiers, limiting the scope of cross-site tracking.

2.3 State Partitioning

Originally, browser states, such as cookies, LocalStorage, and caches, were isolated solely by a single key, the resource identifier [41]. With respect to cookies, this identifier is set to the cookie's origin, which only includes the scheme and the domain of the website that stored it [14, 42]. As a result, requests originally always contained all matching cookies stored for the requested origin, even if the origin was not loaded at the top-level and, therefore, considered as a third party. Although this may not initially seem like an issue, it laid the foundation for cross-site tracking attempts by making cookies accessible across different websites. Furthermore, this behavior also introduced other vulnerabilities known as Cross-Site Leaks [43, 44], where sensitive information could inadvertently be exposed to untrusted sites through various side channels. To protect users against these malicious practices, browser vendors implemented a novel partitioning approach called State Partitioning [13–16]. State Partitioning differs from the original approach by introducing additional partitioning keys to isolate states. Only if all partitioning keys match, access to the specific partition is granted.

2.3.1 Types of State Partitioning

The State Partitioning approaches can be differentiated by the number of keys that are used to partition the browser state. The common approaches at the time of this thesis are double-keying, triple-keying, and 2.5-keying [45]. While double-keying is often in



(a) Resource origin **(b)** Resource origin and TL site

Figure 2.3: Storage access with original partitioning (a) and double-key partitioning (b)

place to protect the user from cross-site tracking, a third key can additionally increase the security by preventing information leakage that could occur among documents that still share a partition in the double-keyed approach [41]. 2.5-keying functions similarly to triple-keying, with the distinction that the third key holds a boolean flag instead of a value, such as whether the frame is cross-site to the top-level site. Although this method does not offer the same security level as triple-keying, as there is more potential for information leakage between documents, it enhances the performance [45]. Figure 2.3 displays the original single-keyed partitioning approach utilizing the resource origin, as well as a double-keyed approach which additionally uses the top-level site as a second key.

2.3.2 Browser Differences

While most browsers have adopted State Partitioning at the time of the thesis, there are differences in the specific keys they use for partitioning. Additionally, the number of keys employed can also vary depending on which particular browser state is being partitioned. In the following, we will list the implementation differences between all browsers for which sufficient information is available on the web, including Firefox, Chrome, Safari, and Brave. However, we will mostly focus on the partitioning of cookies as it is closely related to the Storage Access API.

2.3.2.1 Firefox

In Firefox, storages are partitioned by default [14]. Cookie partitioning was introduced in 2021 with *Total Cookie Protection* as part of Firefox 86 [46]. The first roll-out was part of the Enhanced Tracking Protection (ETP) mechanism that was already established in Firefox at this point. However, the mechanism was only active when the strict mode of ETP was used as it was still being tested. After the small testing period, the mechanism was also introduced in private windows in the middle of 2021 with version 90 [47], as

well as in Firefox Focus in 2022 [48] until it was automatically activated for all Firefox users worldwide in April 2023 [49]. As of now, Firefox version 133.0 is the most recent version and uses a double-keyed partitioning approach for most browser states, including cookies. The origin of the loaded resource is used as the first key, while the schemeful top-level site serves as the second key [50].

2.3.2.2 Chrome

While Chrome also partitions most of the storage and communication APIs by default since version 115 [13], it uses a different approach when it comes to cookies. Chrome does not automatically partition cookies by default, but instead introduced a new mechanism called "Cookies Having Independent Partitioned State" (CHIPS) in version 114 [42]. It adds the ability for websites to use an additional *Partitioned* attribute when setting a cookie. Only when this attribute is set the cookie will be partitioned. The partitioning also uses a double-keyed approach with the origin of the cookie as the first key and the schemeful site as the second one. In order to encourage developers to enforce good security practices, all partitioned cookies need to have the *Secure* attribute set. Furthermore, if the cookie starts with the "__Host-" prefix it is bound to the hostname instead of the site. So, while Firefox partitions cookies by default, Google decided to use an opt-in approach instead. According to Google, the reason for this decision was that automatic partitioning could lead to unexpected bugs if web developers are unaware of cookie partitioning [42]. However, to prevent cross-site tracking using this approach, unpartitioned third-party cookies must be disallowed, so that only first-party and partitioned cookies can be stored in the browser. Earlier, Google already planned for Chrome to phase out all third-party cookies. But although the plan was set, it faced several postponements with the latest timeframe extending to early 2025 [51, 52]. In the beginning of 2024, Chrome already started testing this approach by restricting third-party cookies for 1% of all Chrome users [53]. However, in July 2024 they reverted the phaseout plans and instead of completely removing unpartitioned third-party cookies by default they will leave the decision to the user [54].

2.3.2.3 Brave

According to Brave themselves, they use the most aggressive and protective partitioning approach of any popular browser [16] through a state partitioning system that they call "Ephemeral third-party site storage" [55]. The approach not only places cookies in multiple partitions by using the top-level site as a second key, but also deletes all partitioned data when the user leaves a website or restarts the browser. In addition, no

cookies will be sent in third-party sub-resource requests, neither in a framed first-party context nor in a third-party context [55].

2.3.2.4 Safari

Regarding storage partitioning, Safari partitions third-party storage and service workers by the origin of the loaded resource and the first-party site [15]. Furthermore, Safari's storage also uses an ephemeral approach similar to Brave, which means that the storage does not persist to disk, but instead is cleared when the user quits or restarts the browser [15]. Regarding cookies, Safari previously did not allow third-party sites to set cookies as long as there were no cookies already present for this party. This means that the site would first need to be visited in a first-party context and set a first-party cookie to be able to set third-party cookies later [15]. From March 2020 on Safari decided to completely block third-party cookies [56] as the previous stateful approach could be used again as a tracking vector [57]. Following this decision, the Storage Access API was introduced to account for the blocking of third-party cookies and grant access to cookies that were previously set when the site was loaded at the top-level context [24].

2.4 Storage Access API

While State Partitioning improves the privacy across the web, it also introduces compatibility problems for sites that rely on shared data across different top-level sites. Introduced by Safari in early 2018 [24], the **Storage Access API** was developed to provide a solution to this problem, allowing framed third-party sites that depend on shared cross-site data to function correctly. The API allows framed documents to request unpartitioned access to cookies if needed. However, it is important to note that this does not grant access to all cookies associated with the origin. If unpartitioned access is granted, the document or frame can access only the cookies that were stored during previous visits to the site in a first-party context. For the rest of this thesis, we will refer to these cookies as "unpartitioned cookies".

2.4.1 API Functions

To accomplish the desired behavior, the API defines two new functions to request and check access to unpartitioned cookies: [21, 22]

```
async function doClick() {
  if (!hasAccess) {
    try {
      await document.requestStorageAccess();
      hasAccess = true;
    } catch (err) {
      // Access was not granted.
      return;
    }
  }
  if (hasAccess) {
    // Use the cookies
  }
}
document.querySelector('#my-button').addEventListener('click',
  ↪ doClick);
```

Listing 2.3: SAA example implementation to request unpartitioned cookie access [23]

```
partial interface Document {
  Promise<boolean> hasStorageAccess();
  Promise<undefined> requestStorageAccess();
};
```

Listing 2.4: SAA changes to *Document* [21]

- ***hasStorageAccess()***: Allows to check if unpartitioned third-party cookies access is granted to the framed document on which the function is called. The function returns a promise that, once it is fulfilled, holds a boolean value that indicates the access status.
- ***requestStorageAccess()***: A framed document can call this function to request unpartitioned access to its cookies that were set when the user visited the site as a first-party. If access is denied, an error is thrown. After the function finished, it simply returns an undefined promise. Under some conditions, the user might also need to grant the permission to the site

Both functions are defined on the *Document* interface as seen in Listing 2.4. The reason for this decision is that the API controls access restrictions to *document.cookie*, which is also part of the *Document* interface. And secondly, the access grant is also limited to the specific framed document that requested access, as the API currently operates on a per-frame model [22].

2.4.2 Privacy Considerations

According to the W3C[1] the API has the explicit goal "to not undermine the gains of cross-site cookie deprecation" [21] and should "not degrade privacy properties when compared to pre-removal of cross-site cookies" [21]. Otherwise, any site would be able to request storage access to perform cross-site tracking once again, which would destroy the privacy gains that State Partitioning introduced. So, in order to achieve the goal, there are different privacy considerations that the API should enforce according to the specification [21]:

1. The user needs to **interact** with the document before it can get storage access.
2. The framed document is **permitted by the embedder**.
3. The framed document must be in a **secure context**.
4. The **user permits** the access for the framed document.
5. The user does **not receive an overwhelming number of requests for permission**, preventing their explicit approval from being weakened by decision fatigue.

However, according to the specification, only the first three points are required to be implemented by browsers as it permits "implementation-defined behavior on when to grant or deny requests for unpartitioned data without requiring user choice" [21]. Regarding the last two points, the specification states that browsers should minimize implementation differences and standardize grants without user permissions, as well as denies to ensure that the developer experience does not suffer [21]. However, while all browsers that implement the API prompt the user for permission at some point, the timing varies significantly. We examine these differences in Section 2.4.6.

2.4.3 Security Considerations

Third-party cookie partitioning or deprecation did not only benefit in preventing user tracking, but instead also had a positive effect on the mitigation of attacks such as Cross-Site Request Forgery (CSRF) [21]. With the introduction of the Storage Access API, websites are now back at risk of falling victim to these attacks, especially if website operators are not aware of this risk. To protect websites from this scenario, the storage access is only valid for the frame that requested the access. Furthermore, if the frame navigates to another origin, storage access will also be revoked [21].

[1] `https://www.w3.org/`

2.4.4 Integration into other APIs

Section 2.4.2 mentions the different privacy considerations that browsers should follow when implementing the Storage Access API to ensure the privacy of their users. Two of the considerations included are that both the user and the embedder should grant permission for storage access. Rather than implementing this functionality from scratch, the Storage Access API is integrated into existing browser APIs, allowing documents to query the current user permission state and control which embedded content can access the API.

2.4.4.1 Permissions Integration

The Permissions API allows developers to query user permissions for web platform features such as geolocation, notifications, and camera access [58]. The Storage Access API adds a powerful feature called "storage-access" to the Permissions API that allows framed documents to query the permission status. The returned status can have one of the following values:

- *granted*: The permission is **granted** and access will be given after it is requested.
- *prompt*: The user will be **prompted** to deny or grant the permission when access is requested.
- *deny*: The permission is **denied** and the feature cannot be used. This state is **never returned** to avoid revealing the user's decision to developers.

The scope of the *storage-access* permission is set to the site of the top-level page and the framed document [21]. As a result, frames that share the same site also share the permission to use the Storage Access API as long as they are included on top-level pages that share the same site.

In order to query the permission status, a frame can use the code listed in Listing 2.5. What should be noted is that even when permission is granted, websites still need to request access via *requestStorageAccess* as the permission only represents the decision of the user to allow storage access and does not indicate that storage access is already granted. To verify whether unpartitioned storage is given to the frame, the *hasStorageAccess* function must be used. This is also the only option to get information about the current Storage Access API status for a framed document in Safari as it does not support the Permissions API.

```
try {
  permission = await navigator.permissions.query(
    {name: 'storage-access'}
  );
} catch (error) {
  // storage-access permission not supported
}
```

Listing 2.5: SAA permission query [23]

```
Permissions-Policy: storage-access=(self "https://site.example")
```

Listing 2.6: Example Permissions Policy header

2.4.4.2 Permissions Policy Integration

In some cases, website operators might want to disable some browser features for different reasons, and this is where the Permissions Policy comes in. The mechanism allows websites to declare whether a specific functionality can be used in a Permissions Policy header [59]. The Storage Access API also defines a policy-controlled feature called *storage-access* with the goal of disabling calls to *requestStorageAccess* for any embedded content in the document that the site operator does not trust.

If a response contains no Permissions Policy header, all embedded content is allowed to call *requestStorageAccess*. If the *Permissions Policy* header is set in the response, only the content mentioned in the header is allowed to request access [21]. Furthermore, if a frame is not allowed to request access, all of its children will also lose the permission, even if they are mentioned in the Policy. This is also the reason why "self" needs to be included if the Permissions Policy header is present, as otherwise, the visited site itself would lack the required permission, preventing all of its child frames from invoking *requestStorageAccess*. Similarly, if a frame should be granted permission to use the API function while having other parent frames between itself and the top-level site, these parents would also need to be mentioned in the Permissions Policy header. Listing 2.6 shows an example header for a website that embeds *https://site.example* directly at the top-level and wants to give it permission to request storage access.

Instead of using the Permissions Policy to prevent specific frames from requesting access, untrusted iframes can also be sandboxed. If websites sandbox an iframe but still want to keep the functionality of the API, they need to add the activation flag *allow-storage-access-by-user-activation* [21, 24] in addition to the *allow-scripts* and *allow-same-origin* that are needed to execute JavaScript code and to access same-origin cookies.

2.4.5 Changes over Time

Since the introduction of the API by Safari in the beginning of 2018 [24] there have been multiple changes to the API. Originally, storage access was granted per frame, meaning that other same-site frames also needed to request storage access even on the same top-level site. This, however, changed in 2021 when Safari updated the Storage Access API to operate on a per-page basis [60]. From this moment on, if a frame was given unpartitioned storage access, all other documents loaded on the top-level page, including the top-level document, could also make credentialed requests to the frame's origin as uncovered in a security audit commissioned by Google [27, 61]. The scope of storage access was also of concern as it could be used for endpoints unrelated to the functionality of the API for which Storage Access was requested [27]. To mitigate these issues, three solutions were proposed in the audit report [27]:

1. Prevent cross-origin frames from using storage access that was granted to another frame. Other frames that are same-site to the one that requested storage access should only get access if the frame has no cross-site parents. The API should also be integrated into the Permissions Policy to restrict the capability to request access from the top-level document to its children frames.
2. Require Cross-Origin Resource Sharing (CORS) for credentialed requests from the embedder to the embeddee.
3. Reduce the scope of storage access by limiting credentialed requests to the origin instead of the site and only sending *sameSite=None* cookies.

However, one month after the audit results were published, the view on the proposed solutions changed. Due to security considerations that would need to be in place to ensure a secure and privacy-oriented use of per-page access, the utility of the API would be reduced [62]. If per-frame access were to be used instead, all of these restrictions could be dropped without reintroducing the problems that were revealed in the audit. As a result, the Storage Access API switched back to a per-frame access in the beginning of 2023 [63].

2.4.6 Browser Differences

As mentioned in Section 2.4.2, the Storage Access API specification does not require that browsers implement all privacy considerations and leaves space for browsers to decide when to allow unpartitioned access [64]. For this reason, the implementation differs for most browsers. In the following, we will list the implementation differences between Firefox, Chrome, Brave, and Safari.

2.4.6.1 Firefox

At the time of writing this thesis, Firefox (version 131.0.3) uses a unique approach when it comes to prompting the user for storage access. By default, Firefox allows websites to temporarily gain storage access on five different top-level sites without requiring the user to grant permission via a prompt [64]. However, this only applies if the user has recently interacted with the third party. If no interaction between the user and the third party occurred beforehand, the prompt will still be required if the user has not yet granted permission [64]. Regardless, users still have the option to completely disable the auto-grant behavior or limit the number of sites where it is used by changing the corresponding configuration flags. Regarding user interactions, Firefox only requires one if access permission has not been granted yet. Nevertheless, Mozilla wrote in their blog in 2022 that every storage access request must be preceded by user interaction in order to request access even when the page is reloaded [64]. However, this is incorrect and not the case in version 131.0.3, which is why we created a bug report[2]. Once access is granted to a framed document and this document sets a cookie, Firefox will store this cookie in the first-party partition, meaning that the second partitioning key will be set to the schemeful site of the document as if the cookie has been stored when visiting the site on the top-level. Therefore, this cookie will also be accessible if a same-site frame uses the API to get cookie access on another top-level site.

2.4.6.2 Chrome

Unlike Firefox, Chrome (version 133.0.6905.0) does not grant temporary storage access for a specific number of different sites. Instead, Chrome provides another option for framed documents to skip the user prompt upon calling *requestStorageAccess*. Chrome offers website operators a method to declare relationships among sites via a feature known as *Related Website Sets* [65]. Operators can submit their domains with their corresponding use cases to the set via GitHub [66]. Once Google approves the set, the user prompt is automatically skipped if the top-level and framed sites are both present in the same set [23]. Furthermore, once a user has granted permission through the prompt, it will not appear again for 30 days. Interacting with the iframe will extend the grant each time by 30 more days [23]. User interactions will also not be required until the grant expires. To check if the grant is still active, the Permissions API can be queried. In addition to the requirements that are in place to get storage access, there are also restrictions for cookies to which access is granted. In Chrome, cookies are only accessible after calling the Storage Access API if they are set with *sameSite=None* and *secure* [23].

[2] `https://bugzilla.mozilla.org/show_bug.cgi?id=1928873`

Although other browsers set the *sameSite* attribute to *None* by default, if no value is given, Chrome instead sets it to *lax* [67]. If a framed document was granted access and it would like to set a cookie, these conditions also need to be met in order to set the cookie. Additionally, any new cookies must include the *Partitioned* attribute when third-party cookies are disabled, ensuring they are stored within the partition associated with the current top-level site as the secondary key. Unlike the behavior in Firefox, these newly stored cookies remain inaccessible when unpartitioned cookie access is granted to a frame on other sites.

2.4.6.3 Brave

The Storage Access API is disabled in Brave as the developers believe that "this approach is not the right way to protect privacy on the web" [26].

2.4.6.4 Safari

Although, as mentioned in Section 2.4.1, the Storage Access API specification proposes a per-frame model, Safari (version 18.2) instead uses a per-page approach. This means that once a frame was granted access to its unpartitioned cookies, it is now also granted to all other resource loads on the current page that are same-site to the frame [60]. Before frames can get access in Safari, they need to have been visited and interacted with as a first-party by the user in the previous 30 days. In addition, the site also needs to set cookies in a first-party context before the API may be used [60]. Furthermore, before *requestStorageAccess* is called, a user interaction must always precede. If this is not the case, the request is denied [24, 60]. Page reloads in the same tab are exempted from this behavior, and storage access is automatically granted after the page has been reloaded. Regarding user prompts, Safari only prompts the user for the first time if permission has not yet been granted. As long as Safari remembers the permission grant, the user is not prompted again [60].

2.4.7 Extensions

Initially, the Storage Access API was developed to allow framed documents to request access to unpartitioned cookies in order to avoid compatibility concerns that came with the introduction of State Partitioning. However, recently proposed extensions have expanded its scope to include more use cases. Although these extensions are not yet widely adopted, they should nonetheless be mentioned.

```
dictionary StorageAccessTypes {,
  boolean all = false;
  boolean cookies = false;
  boolean sessionStorage = false;
  boolean localStorage = false;
  boolean indexedDB = false;
  boolean locks = false;
  boolean caches = false;
  boolean getDirectory = false;
  boolean estimate = false;
  boolean createObjectURL = false;
  boolean revokeObjectURL = false;
  boolean BroadcastChannel = false;
  boolean SharedWorker = false;
};
```

Listing 2.7: Storage access types added in the SAA non-cookie storage extension [69]

```
const handle = await document.requestStorageAccess({
    localStorage: true,
});
```

Listing 2.8: Example code to request unpartitioned localStorage Access with the SAA Non-Cookie Storage Extension [70]

2.4.7.1 Storage Access For

In addition to the *hasStorageAccess* and *requestStorageAccess* functions, Google also proposed a third function to be added to the Storage Access API, called *requestStorageAccessFor* [23, 68]. While the initial two functions can be used by framed documents to request unpartitioned cookie access to their own cookie storage, *requestStorageAccessFor* instead is called by the top-level site to request storage access on behalf of another framed site in order to make credentialed cross-site requests. In order to prevent abuse, a top-level site must be in the same *Related Website Set* as the site for which the top-level requests access. However, at the time of writing this thesis, the extension was not included in the original Storage Access API draft [21], although it might be introduced later.

2.4.7.2 Non-Cookie Storage

In June 2024, the W3C published a second draft of the Storage Access API, which proposed an extension that would allow sites to request access to unpartitioned data storages beyond cookies, such as *localStorage*, *indexedDB* or *caches* [69]. In order to achieve this, the Storage Access API functions were changed:

- ***hasUnpartitionedCookieAccess()***: This is the old *hasStorageAccess* function, which was renamed to *hasUnpartitionedCookieAccess* to make it clear that it solely returns whether unpartitioned cookie access is given to the calling document.
- ***requestStorageAccess(type)***: To be able to request access to different storage types, the function was extended to take a type argument that reflects the storage to which the website would like to receive unpartitioned access. If access is granted, the function returns a handle that can be used to access the unpartitioned data. Listing 2.7 shows the different storage types that sites can request unpartitioned access to.

Once the extension is adopted, existing implementations that request unpartitioned access for cookies will need to be updated to account for the additional argument to the *requestStorageAccess* function. Listing 2.8 provides an example code snippet demonstrating how to use the Storage Access API to request unpartitioned *localStorage* access with the extension.

2.4.8 Alternatives

Apart from the SAA, there are also alternative mechanisms that can be used to evade the compatibility problems introduced through State Partitioning:

- **Federated Credential Management** (FedCM): FedCM [71] is a privacy-preserving solution that allows users to log in to websites via an identity provider without directly sharing personal information with either the provider or the site relying on the authentication. FedCM takes over the log-in process, allowing users to choose an identity provider while minimizing the data shared with the websites they access. This approach improves user privacy compared to traditional single sign-on by reducing cross-site tracking risks, making it a potential solution in a more privacy-focused web.
- **Storage Access Heuristics**: To maintain web compatibility and prevent site disruptions, modern browsers like Firefox, Chrome, and Safari allow websites to temporarily access unpartitioned cookies under specific conditions [14, 72, 73]. These conditions consist of common user workflows that provide confidence signals of legitimate third-party cookie usage, including scenarios such as interactions with pop-ups or redirects, which are often used for user authentication [72, 74]. However, web developers should not rely solely on this mechanism as it is only a temporary solution to ensure compatibility on the web [14, 72, 74]. In the future, the heuristics might get removed, and in this case, websites need to look for other ways to ensure compatibility.

Chapter 3

Related Work

In this chapter, we review related work closely connected to our study, focusing on key areas such as online tracking, privacy-compatibility trade-offs, the Storage Access API, and web measurements.

3.1 Online Tracking

Web tracking, especially through cookies and similar technologies, has been a focus of numerous research studies over the past decade, as it heavily violates user privacy. Mayer and Mitchell [1] was one of the first to conduct a comprehensive analysis of third-party tracking, highlighting the widespread use of cookies and fingerprints to track users across different websites. Their work laid the groundwork for later studies exploring how browser-based tracking mechanisms impact user privacy [3, 7] and how they have evolved over time [5]. In the following years, many other publications have also looked at other ways of tracking users online, such as mobile tracking [75–77], cookie syncing [10, 39, 40, 78], pixel tracking [9], and browser fingerprinting [20, 79–81].

While much of the research has focused on tracking mechanisms and their privacy implications, some studies have also explored methods to detect and defend against these practices. For example, Roesner et al. [6] developed a tool that identifies trackers by automatically classifying websites based on their observed client-side behavior. Furthermore, they also analyzed existing defenses and proposed a new defense to prevent tracking through social media widgets. A few years later, Bujlow et al. [8] expanded on this work by evaluating a broader range of defenses, including many approaches that were not yet covered in prior research, such as self-destroying file systems. While these studies primarily concentrated on defenses built on top of browsers like content

blocking and private browsing, Pan et al. [82] took a different approach by designing a new browser that inherently blocks third-party tracking while still trying to maintain web compatibility. Their evaluation demonstrated that this browser could effectively block all trackers across the top 500,000 Alexa sites, marking a significant advancement in practical tracking defenses.

Due to heavy privacy concerns regarding user tracking, the GDPR [36] was introduced to protect users' privacy in the European Union. Since its introduction, several works have tried to measure the impact of the regulation. These studies have shown that while the GDPR has brought some positive changes, such as increased transparency and user control over personal data, the effectiveness of these regulations in fully protecting users from tracking remains debatable on the web [83, 84], as well as in the mobile environment [85]. For instance, Kretschmer et al. [83] conducted an empirical analysis to assess how websites have adapted to the GDPR and quantified its impact on personal data processing practices. Their findings indicate that a substantial portion of websites remain non-compliant with GDPR requirements, with only a small fraction fully adhering to the regulation's standards.

3.2 Privacy-Compatibility Trade-Off

Privacy-preserving mechanisms on the web aim to protect users from online tracking by restricting the sharing of user information across sites. However, implementing these privacy protections often introduces compatibility challenges for services that rely on cross-site data sharing. Consequently, balancing strong privacy guarantees with seamless functionality has become a central challenge in web design, prompting recent research to examine the trade-offs between privacy and compatibility. To measure how different third-party storage policies impact the functionality of various websites, Jueckstock et al. [17] conducted an empirical study considering three different policies: blocking third-party storage, partitioning it by the top-level site, and ephemeral storage. The authors concluded that the ephemeral storage approach effectively protects user privacy while maintaining a high level of web compatibility.

Beyond storage-based strategies, browsers can also use filter lists to block unwanted trackers. However, a common drawback of filter lists is that they can disrupt website functionality by blocking necessary content, particularly if not thoroughly tested. To address this issue, Smith et al. [18] developed a fully automated framework to predict whether a filter list rule would break specific features on a website.

3.3 Storage Access API

When it comes to the Storage Access API, only little research has been conducted since its introduction by Safari in 2018 [24]. In September 2022, Google employees published a previously mentioned theoretical evaluation of the Storage Access API security model in order to improve the security implications of the mechanism [27]. The security audit showed that there were still some security concerns with the design at the time, however, these were changed in the official W3C work item later [21].

Beyond this audit, the research community has largely overlooked the API itself. However, a recent study performed by McQuistin et al. [86] has focused on Related Website Sets, a concept closely related to the Storage Access API. Their goal was to determine whether users can accurately determine if two sites within the same set are related to each other. Results indicated that over 35% of user assumptions were incorrect, with many users failing to recognize sites as related even when they belonged to the same set. This finding underscores the potential for abuse in indicating relationships that do not actually exist, which is particularly important for the privacy considerations of Chrome users regarding the Storage Access API. Since Chrome does not require user permission when both the top-level site and the framed document belong to the same set, being part of the same set can inadvertently make tracking easier.

3.4 Web Measurements

Web measurements have been an essential tool for understanding various aspects of the web. Therefore, several studies have highlighted the challenges and nuances of conducting reproducible web measurements under different experimental conditions. Demir et al. [87] highlighted that many web measurement studies lacked sufficient documentation of their experimental setups, leading to reproducibility issues. To address this, they proposed a list of best practices aimed at standardizing methodologies and improving the reproducibility of web measurement research.

Other studies have highlighted that experimental setups can significantly affect measurement outcomes. For example, Jueckstock et al. [88] demonstrated that factors such as vantage points and browser configurations influence crawl observations and, consequently, the insights derived from such measurements. Jueckstock et al. [89] further noted that using cloud-based vantage points for crawls can yield less accurate results due to potential "blind spots in third-party content coverage". Similarly, Ahmad et al. [90] found that the choice of crawling technology can also impact measurement results, emphasizing that technical differences in crawling tools can lead to variations in observed data. Finally,

several other works also uncovered that landing pages vary significantly from internal pages [91] and that studies that only cover landing pages would overlook a large number of third-party embeds and cookies [92].

As an alternative to live security measurements, which can suffer from reproducibility issues, Hantke et al. [93] recently also proposed using web archives for crawling instead, as they offer a more reproducible environment for security measurements compared to real-world data collection.

Chapter 4

Methodology

In order to answer the research questions previously mentioned in Section 1.1, we conducted an empirical large-scale web study to investigate the prevalence of the SAA on the web. To achieve this, we implemented a web crawler that collected real-world data from a list of randomly sampled websites. Later, we analyzed the collected data to reveal the usage numbers of the SAA on the web, as well as the privacy risks that the API might entail for users. In this chapter, we describe the implementation details of our web crawler, including its design, functionality, and the methods employed to detect instances of SAA usage on visited websites. Additionally, we elaborate on the analytical methods used, including our assessment of the privacy risks associated with SAA usage on individual sites.

4.1 Crawler Implementation

A web crawler is an automated software program or script that systematically navigates the Internet, browsing web pages, and collecting their content. It normally works on a set of initial URLs and then follows links from those pages to discover and process additional pages [94]. Web crawlers are primarily used by search engines to index the content of websites for easier retrieval during searches. However, nowadays, they are also widely used for web testing [94], as well as automated security testing [94–96]. Furthermore, they can also be employed in various large-scale data collection tasks to perform an empirical study on real-world web data.

4.1.1 General Functionality

As web crawlers can be quite complex, we decided to start our crawler implementation based on an existing crawling framework that was implemented by Rautenstrauch et al. [97] for their research on the pre- and post-login security landscape of websites[1]. It is based on the Python Playwright framework[2] that can be used to automate web testing or scraping by instrumenting different browsers through a single API. At the time of our first crawl, the current Playwright version was *1.46.0*, so we stuck to this version for all of our crawls to ensure consistency. In order to minimize dependency issues and to ensure a consistent environment containing all required browser executables, the crawler runs inside a Docker container which is based on the official Playwright Docker image[3]. The code of our crawler is fully uploaded to GitHub[4]. Figure 4.1 shows the overview of our crawling framework, consisting of three different components:

- **Database:** The database (*PostgreSQL*) is essential for the crawler functionality, as it stores all the tasks that the crawler should work on. In addition, it also stores persistent data that the crawler modules collect during the crawl. The database runs in another docker container based on the official postgres image[5], which is connected to the crawler container via a Docker network.

- **Crawler Loop:** The crawler loop is responsible for loading the tasks stored in the database and processing them. Once a task is loaded, it visits all URLs that are connected to this task using the Playwright framework. The crawler runs until the crawler loop has processed and completed all tasks. The crawler loop can also run multiple crawlers at the same time; however, two distinct threads can never work on the same task.

- **Crawler Modules:** The crawler modules serve as the crawler component that is dedicated to collecting data and information about the pages the crawler visits. Listing 4.1 shows the base *Module* class which contains three different functions that must be used to implement the functionality of the module. If the module is loaded, the functions are executed by the crawling loop at function-specific moments during the loop. To store the collected data persistently, the modules can use the database. When starting the crawler, one can decide which modules to load and include during the crawl via command line options.

[1] `https://github.com/cispa/login-security-landscape`
[2] `https://github.com/microsoft/playwright`
[3] `https://hub.docker.com/r/microsoft/playwright`
[4] `https://github.com/BausPhi/Storage-Access-Crawler`
[5] `https://hub.docker.com/_/postgres`

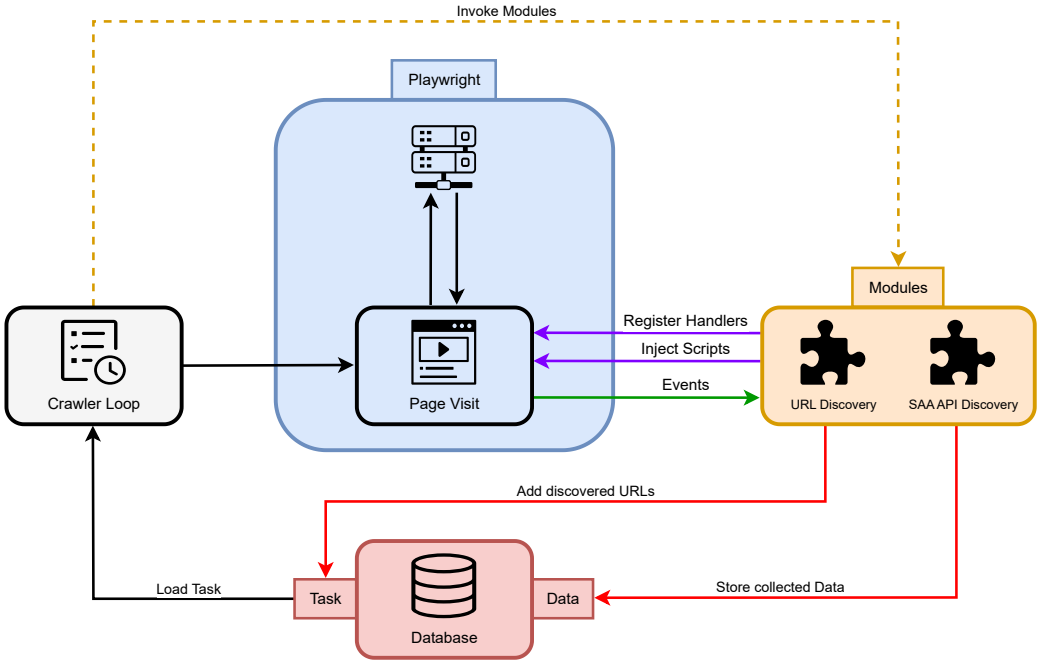


Figure 4.1: Overview of the crawler

```python
class Module:

    def __init__(self, crawler) -> None:
        from crawler import Crawler
        self.crawler: Crawler = crawler

    @staticmethod
    def register_job(log) -> None:
        # Initialize crawl preparations.
        # Called before the crawl starts.
        # Use case: Create database tables
        pass

    def add_handlers(self, url) -> None:
        # Register event handlers for a page.
        # Called before navigating to the page.
        pass

    def receive_response(self, responses, url, final_url, start,
    ↪  repetition) -> None:
        # Receive response object from the visited page.
        # Called after the page visit ended.
        pass
```

Listing 4.1: Crawler module interface [97]

Before the crawler can be started, all tasks must first be stored in the database. After this initial process is finished and all tasks are contained in the database, the crawler then starts to pick tasks from the database and visits the URLs associated with it. However, before a page is loaded, the crawler will invoke the *add_handlers* function of every crawler module currently loaded. The function can then register all of the event handlers that the module requires to collect the relevant data. Furthermore, scripts that should be injected into the page when it is initialized can also be registered inside this function. Once this step is completed, the page is finally loaded inside a headful or headless browser, depending on the crawler configuration. If an event that was previously registered occurs during the page visit, the registered handler inside the corresponding crawler module is invoked and executed. The duration of the page visit is determined by two factors: the load condition and the wait time after the load condition is fulfilled, both of which can be modified in the configuration. Once the page visit is completed, the *receive_response* function is executed, which allows the module to handle the response object of the top-level site including its content, headers, etc. After this step, the URL visit is completed and the next URL of the current task is processed. If there are no other URLs for the current task, the next task is retrieved from the database.

We use two modules in our crawler implementation: a URL discovery module that was already developed by Rautenstrauch et al. [97] and our own SAA discovery module which will be explained in detail in Section 4.1.2. The URL discovery module is responsible for parsing the content of the top-level site after each page visit to discover additional unvisited URLs and storing them in the database. Initially, tasks only contain the landing page; all other URLs will be added dynamically through the URL discovery module.

The crawler is configured through a dedicated configuration file, which allows for customization of various settings. In this file, users can adjust key parameters such as the browser to be used, whether the crawl should run in headless or headful mode, and the timezone. Additionally, it enables control over the crawler's behavior during page visits: users can specify the load condition that determines when a page is considered fully loaded, set the duration for which the crawler remains on the page after loading, and establish a timeout period that prompts the crawler to restart if an error occurs. This configuration file also supports module-specific settings. For URL discovery, users can enable or disable discovery, set a maximum number of URLs to be discovered per site, define the depth level of URL exploration, and specify whether discovered URLs should be restricted to the same-origin or same-site. Since our crawl tasks were always scoped to the site, our configuration consistently limited discovery to same-site URLs.

```
STRING_MATCHING_HAS_SAA: str = r"\bhasStorageAccess\b"
STRING_MATCHING_REQUEST_SAA: str = r"\brequestStorageAccess\b"
```

Listing 4.2: Regex pattern for string matching

4.1.2 SAA Discovery

While much of the crawler loop functionality, as well as the URL discovery module, were already included in the crawling framework, the SAA discovery module was developed in its entirety by us. The primary objective of this module is to systematically monitor and record the usage of the SAA on the websites visited by the crawler. To accomplish this, the module employs two distinct techniques: string matching, which scans for patterns indicative of API usage within the page content, and function hooking, which intercepts and logs calls to the SAA during execution.

Additionally, we made minor but essential adjustments to the crawling framework by changing the flags with which Playwright runs the browsers. Specifically, we modified Chromium to disable third-party cookies by setting the *–test-third-party-cookie-phaseout* flag. Furthermore, we also enabled state partitioning in Firefox by changing the *network.cookie.cookieBehavior* setting to "5". By default, Playwright sets this value to "4" to disable State Partitioning. These adjustments were necessary to create a crawling environment in which cookies are correctly partitioned and the SAA is required to get access to unpartitioned cookies. Without these changes, websites would not need to call the API to get unpartitioned cookie access, which would distort our results.

4.1.2.1 String Matching

In order to retrieve the content of a document or script to perform string matching on, we register a *response* handler with Playwright in the *add_handlers* function of our module. So, when the browser receives a response during a page visit, Playwright will call back to our response handler. Our handler then retrieves the content of the document or script via the Playwright *Response* object, and we can then perform string matching on it. For the string matching, we use a simple regex to detect internal or external scripts embedded in a web page that include SAA functions. Listing 4.2 shows the regex as defined in the crawler configuration. The regex we used was intentionally kept simple and triggered whenever a document contained "hasStorageAccess" or "requestStorageAccess" with a word boundary left and right. The use of word boundaries around the function names ensures that only exact matches of these terms are detected, rather than partial matches. For example, without word boundaries, the regex could match the SAA extension *requestStorageAccessFor* that was mentioned in Section 2.4.7.1 and classify it as

```
def inject_js(page):
    page.add_init_script(path="./resources/hook_api_calls.js")

# Register load event handler to inject function hooking script
self.crawler.page.on("load", inject_js(self.crawler.page))
```

Listing 4.3: Injection of the SAA hooking script

requestStorageAccess instead, which would be incorrect. Using word boundaries helps to avoid these false positives by ensuring that only whole words are detected. Additionally, the pattern also attempts to avoid overly strict patterns as it could miss variations due to different developer code styles or formatting. Minified scripts might, for example, bind the *document* object to a different variable that contains fewer characters and call the SAA functions later on this variable. In this case a stricter regex would miss the inclusion. However, the regex pattern also has a downside in that it will also match a string comprising the names of the API functions if it is enclosed by a word boundary. Nevertheless, we see the scenario in which a website includes the SAA functions in a string but does not use the functions themselves, as very unlikely.

String matching enables us to identify instances where SAA functions are present in both external and internal scripts embedded in a web page. This approach helps us detect if these functions are being included, but also gives us some additional contextual information, like the URL of the document or script where the function was included, as well as the parents of the document that embedded the script. However, while string matching is effective for flagging the presence of these functions, it does not provide deeper insights into whether these functions are actually being called during the script's execution.

4.1.2.2 Function Hooking

In order to gather more detailed information about the usage of the SAA, we also monitored the actual calls to the API functions with the help of function hooking. Function hooking is a technique used in programming to intercept calls to a particular function, which can be used to monitor and collect information whenever the function is called. It allows developers to insert custom code that can run before, after, or even instead of the original function's execution. Function hooking in JavaScript is straightforward, as scripts with access to a specific object can overwrite any of its properties or methods.

In order to inject our own function hooking script, we simply use Playwright's *add_init_script* function to inject our script into every frame on the current page that is attached or

```
const originalHasStorageAccess =
    document.hasStorageAccess;
const originalRequestStorageAccess =
    document.requestStorageAccess;

document.hasStorageAccess = async function() {
    const result = await originalHasStorageAccess.apply(
        this, arguments
    );
    window.sa_call_handler(
        "hasStorageAccess",
        window.location.href
    );
    return result;
};

document.requestStorageAccess = function() {
    const result = originalRequestStorageAccess.apply(
        this, arguments
    );
    window.sa_call_handler(
        "requestStorageAccess",
        window.location.href
    );
    return result;
};
```

Listing 4.4: SAA hooking script

```
# Expose the SAA call handler to the window
# Function is invoked with a call to "window.saa_call_handler"
self.crawler.page.expose_function(
    "sa_call_handler",
    handle_storage_access_api_called
)
```

Listing 4.5: Exposing of the SAA call handler to the window

navigated to a different URL [98]. Our function hooking script then proceeds to overwrite the original SAA functions on the *document* object in each frame. Listing 4.3 and Listing 4.4 show the injection of the hooking script with the *add_init_script* function, as well as the function hooking script which is injected into every frame. In order to retrieve and collect the call data from within the automated browser, we can expose a module function to the browser window using Playwright's *expose_function*. Listing 4.5 shows the module code that exposes the API call handler function to the page. If at some point during the page visit one of the SAA functions is invoked, the browser will instead call our function as we overwrote the original function. While our hooking function still calls the original API function at the beginning to maintain the functionality, it also calls the *saa_call_handler* module function that we exposed to the window. The call will contain

the API function that was called, as well as the document in which the function was called. The call handler will then proceed to store this data in the database.

In order to improve the effectiveness of this approach, we also assign the *storage-access* permission to the browser context. This means that every frame on each page that we are visiting already has the permission to request access to unpartitioned cookies and does not require a user prompt. Furthermore, Chromium also does not require user interaction if permission is already granted [23], so websites might be more likely to call SAA functions without any user interaction when crawling with Chromium, which should improve our results. Prior to our crawls, Playwright had not yet implemented a way to grant the *storage-access* permission to websites. Therefore, we created a feature request on the official Playwright GitHub page[6] which led to the implementation of the feature in version 1.46, right in time for our crawls.

4.1.3 Data Collection

All of the data that we collect during our crawl is stored persistently in the database. Figure 4.2 shows the database structure that is used to store all the data that is collected. The data can be separated into three distinct categories: SAA inclusions, SAA calls, and cookies.

4.1.3.1 SAA Inclusions

The SAA inclusion data contains all the information collected during the crawling phase using string matching. The collected data is divided into four distinct tables: *Script*, *Document*, *ScriptInclusion*, and *DocumentInclusion*. Each of these tables plays a specific role in organizing the data. The *Script* and *Document* tables represent unique scripts and documents that are embedded on a website and are identified by their respective *sha1* hashes. The inclusion tables, *ScriptInclusion* and *DocumentInclusion*, instead store information about the inclusion of a script or document when it is embedded on a page.

The *Script* and *Document* tables store metadata about unique JavaScript files and HTML documents encountered during the crawl, respectively. Each resource is uniquely identified using its *sha1* hash, ensuring that identical scripts or documents encountered on different pages or at different times are recognized as the same resource and are not added twice to the database. As the same script or document will also always include the same SAA functions, the information about the presence of the API functions that were found through string matching is also stored in these tables.

[6] `https://github.com/microsoft/playwright/issues/31227`

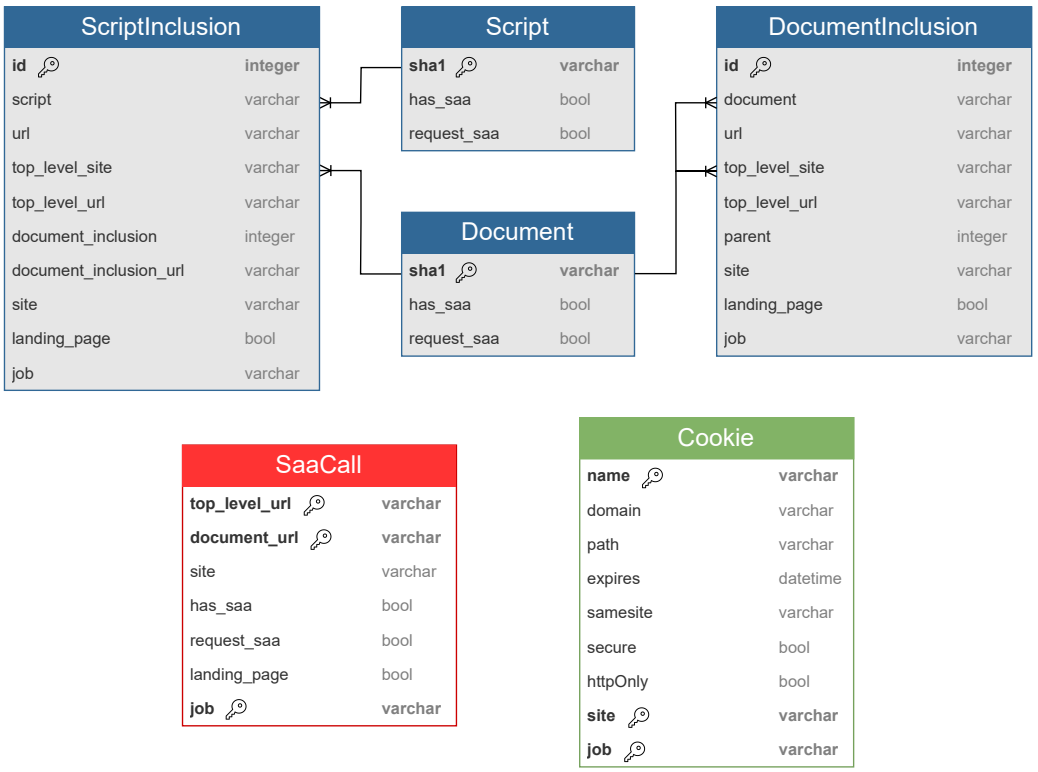


Figure 4.2: Database structure to store collected Data

The *ScriptInclusion* and *DocumentInclusion* tables track the actual inclusions and link them back to the respective resource in the *Script* or *Document* tables using a foreign key field that contains the resource's row id. Additionally, the tables also contain other information about the inclusion context, such as the resource URL, the top-level URL, the site that was visited when the resource was included, whether the visited page was the landing page, and a foreign key field to the document of the top-level site in the *Document* table. Furthermore, the *ScriptInclusion* table includes two additional columns: one that contains the ID of the corresponding entry in the *DocumentInclusion* table, indicating in which frame the script was included, as well as its url. Conversely, the *DocumentInclusion* table features a reference to the parent frame that performed the inclusion of the framed document, which provides us with means to recover the hierarchical structure of the embedded frames later. With the collected data, we are able to conduct a comprehensive analysis of various aspects related to the SAA usage and provide valuable insights into the distribution, frequency, and embedding patterns of SAA inclusions across different sites.

4.1.3.2 SAA Calls

The second type of data collected during our crawls pertains to actual calls to the SAA functions, which we captured using the function hooking approach described in Section 4.1.2.2. All collected data are stored in the *SaaCall* table, including the specific function that was invoked. However, due to the limitations of this data collection method, we were unable to gather the same granularity of data as with the inclusions identified through string matching. Specifically, we were unable to capture details about parent frames for calls. This limitation primarily stems from browsers restricting access to information about cross-origin parent frames via JavaScript. Additionally, we could not identify the exact script responsible for calling the SAA function as we could not think of a way to retrieve this information without patching the browser.

A row in the *SaaCall* table is uniquely identified by the top-level URL, the document URL in which the function was called and the crawl job. The top-level URL and the document URL were both chosen as a primary key to separate calls that occur in the same document but on different top-level pages. The job was used as a third key in order to also store calls across multiple crawls. Using our collected data on calls to the SAA, we can perform similar analyses to those conducted for inclusions. This enables us to assess patterns in API usage, such as the frequency and distribution of calls across different sites and frames. Furthermore, by comparing actual API calls to the recorded inclusions, we can measure how often the functions are actively invoked when present.

4.1.3.3 Cookies

Lastly, we also collected cookies from websites that use the SAA to request unpartitioned cookie storage, as they are used for our privacy risk analysis, which we will explain in detail in Section 4.3. While we collect inclusion and call data during the initial crawl, we handle it differently for cookies. In order to only collect cookies from websites that are using the *requestStorageAccess* function, our crawler performs a second cookie collection crawl after the initial crawl. During this second stage, the crawler will visit all domains that included *requestStorageAccess* at one point during the initial crawl as a first party to collect all cookies relevant for SAA usage. To retrieve the cookies, our crawler module uses Playwright's *cookie* function, which returns all cookies stored for the current browser context. After retrieving the cookies, they will be stored in the *Cookie* table together with all attributes, such as *sameSite* and *secure*. During the cookie collection stage, we do not collect any further SAA data, as we might visit sites that are outside our dataset and scope, which could potentially distort our results.

4.2 Evaluation Crawls

Evaluation crawls are preliminary web crawls designed to test and analyze the effectiveness of different crawling techniques and configurations. By running a series of these smaller crawls, one can gather data on various parameters, such as crawl speed, data accuracy, error rates, and resource consumption. This information helps us identify the optimal settings and strategies to apply in the primary, large-scale crawl and allows us to measure how each setting impacts performance and data quality under different conditions.

4.2.1 Multi-Browser Evaluation

First of all, we decided to evaluate the impact of crawling with **multiple different browsers**. This gave us the opportunity to examine the differences in SAA usage across different browsers, if there are any. Depending on the results, we could then decide whether we use different browsers for our main crawl or decide on one browser. By choosing only one browser, we could increase the number of sites we crawl, as it enables us to cover more pages within the same timeframe. For the evaluation, we defined that browsers are equal in terms of SAA usage for a specific site if all scripts loaded on that site that are including the API are equal. We verify the equality of scripts by calculating the *sha1* hashes and comparing them between the different browsers. Only if the list of script hashes is equal for every browser do we consider the usage equal. Due to the complexity of extracting all inline scripts, including event handlers, from HTML documents, we opted to calculate the hash over the entire HTML document instead of hashing all inline scripts separately. This was fine as we anticipated the need for manual verification after the crawl due to possible crawling inconsistencies between the browsers. During manual verification, if differences were detected between HTML documents using the SAA, we extracted only the inline scripts to compare their hashes, omitting the rest of the document from the analysis.

To deepen our analysis, we decided to also categorize the crawled sites into groups based on their SAA usage: specifically, (1) distinguishing between the use of *hasStorageAccess* and *requestStorageAccess*, and (2) identifying whether the API is utilized in external scripts or inline scripts. This categorization results in four distinct groups, each analyzed separately, allowing us to pinpoint where browser-specific inconsistencies might occur. For instance, differences between browsers may manifest primarily in inline scripts, while external scripts that use the API may generally remain consistent across browsers. It is worth noting that some websites may include resources that fall into more than one category, leading to their inclusion in multiple groups. For each group, we consider a site's SAA usage to be consistent across browsers if all resources that belong to this

group and that employ the API are identical on that site. Any variations in resources on the same site that belong to other groups are excluded from the analysis of the regarded group.

During the evaluation, we did not look manually into the scripts themselves to check if they contained conditions which would call the API only when loaded in specific browsers. This would have been too much of an overhead for an evaluation crawl. Additionally, we excluded actual calls to the API from our analysis due to limitations in granting the *storage-access* permission in WebKit and Firefox browsers. This permission is currently supported only by Chromium in Playwright version 1.46, which could lead to incomplete data if certain *requestStorageAccess* calls are missed in Firefox and WebKit. Including these calls could bias our results as the call differences between Chromium and the other browsers could simply be related to the missing permission which might discourage sites from calling the API in Firefox and WebKit.

4.2.2 User Interaction Evaluation

Secondly, we also evaluated the impact of **user interactions** on the crawling results as prior research indicated that user interactions impact the number of nodes that are loaded on a website [99], which could also have an impact on SAA inclusions and subsequent calls to it. Furthermore, all browsers require user interaction in the framed document before unpartitioned cookie access can be granted. Therefore, websites might hide the *requestStorageAccess* call behind a user interaction which we could not trigger without any simulated actions, which might reduce our capabilities to detect inclusions and reveal calls to the API. However, as our interactions are performed randomly, it is not guaranteed to improve the results as calls might be hidden behind more complex interactions that we are unable to perform with this approach. To measure the difference, we ran one instance of the crawler without any interactions, while a second instance instead employed different user interactions. To simulate interactions like scrolling and button-clicking in a way that closely mimics real user behavior, we needed more than simple JavaScript injection, as code-generated interactions are not recognized as genuine user actions. Instead, we used the Playwright API which provides us with functionality to simulate such user interactions on the visited page and making them appear to the browser as though performed by an actual user. The *mouse.click* function is used to simulate clicks on elements on the current page, while the *mouse.wheel* function mimics mouse wheel movements to perform scrolling.

4.3 Privacy Risk Analysis

To assess the tracking risks associated with the SAA, we propose a formula to measure the *Tracking Risk Score* (TRS) for sites using the API. This score captures the likelihood that the API is being used to track users, combining two critical factors:

- **Percentage of Advertising Cookies (A):** The proportion of cookies classified as targeting or advertising cookies. These cookies are directly associated with tracking and user profiling, making them a key indicator of tracking risk.
- **Tranco Rank (R):** The rank of the website on the Tranco list, representing its popularity and influence. Higher-ranked websites (lower rank numbers) typically attract larger audiences and have more sophisticated tracking systems, increasing privacy risks.

The TRS formula integrates these attributes into a normalized score ranging from 0 to 100, calculated as follows:

$$TRS = \left(w_A \cdot \frac{A}{100} + w_R \cdot \left(\frac{log_{10}(R_{max}) - log_{10}(R) + 1}{log_{10}(R_{max})} \right) \right) \cdot 100$$

where:

$\mathbf{A}$ = Percentage of advertising cookies on the website

$\mathbf{w_A}$ = Weight factor for the percentage of advertising cookies

$\mathbf{R_{max}}$ = Maximum Tranco rank

$\mathbf{R}$ = Tranco rank of the website

$\mathbf{w_R}$ = Weight factor for the Tranco rank

In order to obtain a value between 0 and 100, the weights (w_A and w_R) must add up to 1. We decided to assign the following values to the weights:

$$\mathbf{w_A} = 0.7 \text{ and } \mathbf{w_R} = 0.3$$

This weighting decision reflects the relative importance of advertising cookies and Tranco rank in assessing privacy risks. Advertising cookies are directly tied to user tracking and profiling, and they are the primary mechanism for cross-site tracking. Because of this, we assigned a weight of 0.7 to the advertising cookies component. This emphasizes the central role of advertising cookies in user privacy risks. In contrast, the Tranco rank,

which reflects the website's popularity, plays a secondary but still important role in privacy risk. Higher-ranked websites tend to have a larger audience and may use more sophisticated tracking systems, but their role in privacy risks is more indirect. Therefore, we assigned a weight of 0.3 to the Tranco rank, recognizing its relevance while prioritizing the direct impact of advertising cookies on user privacy. If a site was not part of the Tranco Rank, we set the rank to the last listed rank in the Tranco list which is equal to R_{max}.

To calculate the percentage of advertising cookies employed on a site, we collect unpartitioned cookies from all sites that were found to include the *requestStorageAccess* function in one of their embedded scripts during the initial part of our crawl. We refrained from using only sites that actually called the function as we could miss sites that hide the call behind some user interaction, which we would not be able to trigger with our crawling setup. Furthermore, we exclude session cookies from our analysis, as session cookies are deleted after each browsing session and are unlikely to be used for tracking purposes, given that tracking would only be possible within the same browser session. Additionally, only *sameSite=None* cookies are considered in our analysis, as these are the only cookies that can be sent in cross-site requests even if storage access was granted.

In order to calculate the score for the Tranco rank of the website, we use a logarithmic scale to ensure that the difference between ranks at the top of the Tranco list has a more substantial impact on the score than differences among lower-ranked sites. For instance, we assume that the distinction between the 1st and 100th rank is far more significant in terms of potential tracking impact than the difference between the 10,000th and 20,000th rank.

Importantly, tracking risk does not automatically translate into high privacy risks. If a site is embedded on only a small number of top-level sites, the tracking party may be unable to collect a large amount of information through cross-site tracking, limiting its capacity to compromise user privacy significantly. So, After evaluating the risk of a site tracking users through the SAA, we also look at the privacy implications by examining the total number of distinct top-level sites on which the sites are framed. This metric is crucial for assessing the potential scope of information that might be collected through cross-site tracking. The more top-level sites frame a site, the greater its reach across the web, enabling it to aggregate data from diverse sources. This extended reach allows the site to build more precise and comprehensive user profiles, which has great implications on user cross-site tracking and subsequently also privacy.

Chapter 5

Evaluation of Crawling Strategies

In this chapter, we will present the results of our crawl evaluation regarding the use of multiple browsers, as well as the use of user interactions. We analyze how the browser choice impacts the detection of Storage Access API usage across different websites by examining whether certain browsers reveal more instances of the API than others. Additionally, we assess how simulated user interactions, such as clicking and scrolling, influence the detection of the API. This analysis provides insights into the effectiveness of the different crawling techniques and helps us in adopting the optimal crawling scope for our subsequent crawls to capture the best possible results on the Storage Access API usage on the web.

5.1 Multiple Browsers

In the browser evaluation, our goal was to determine whether the Storage Access API inclusions vary significantly across different browsers. Based on the results of this preliminary evaluation, we plan to adapt the crawling scope for the main crawl, potentially by using multiple browsers. We decided to evaluate this crawling strategy with Chromium, Firefox, and WebKit as they are available by default in the Playwright Docker container. We used Playwright version 1.46, which resulted in the following browser versions:

- Chromium: **128.0.6613.18**
- Firefox: **128.0**
- WebKit: **18.0**

5.1.1 Experiment Scope

In our browser evaluation crawl, we selected a random sample of 5,000 websites from the Tranco ranking [100], generated on 25 June 2024[1]. To ensure diversity in site popularity, we selected the sample dataset from different rankings: specifically, we sampled 2,500 websites from the Top 10,000, 1,500 websites from ranks 10,000 to 100,000, and 1,000 websites from ranks beyond 100,000. The number of sites chosen was deliberately kept low, as we anticipated a need for manual verification due to inconsistent crawling behavior between the browsers, such as inconsistent load times of files, resulting in false positive differences. Additionally, to be able to expand the number of sites that we crawled, we limited our visits to the landing page of each site. This also ensured that the visited pages were equal for every browser as the landing page was fixed for all browsers and we did not visit any dynamically discovered URLs. Furthermore, to minimize the possible changes that are applied to the site in between the visits of different browsers, we crawled a site with all three browsers one after the other before moving to the next site. To manually verify all differences discovered during the crawl, we visited the respective sites in each browser and made sure that all the scripts that included API functions were equal and included in each browser.

5.1.2 Result

Table 5.1 shows the results of our multi-browser evaluation, detailing the total number of top-level sites using the Storage Access API, along with the number and percentage of these sites with consistent Storage Access API usage across browsers. Table 5.1a represents the values retrieved during the crawl before our manual verification, while Table 5.1b already contains the corrections from the verification process. In general, 449 out of 517 sites, equivalent to 86.85%, were found to have consistent Storage Access API usage during our initial crawl. After completing manual verification, this number increased to 514 sites, bringing the equality rate up to 99.42%. The inconsistencies that we rectified in our manual analysis mainly arose from differences in bot detection mechanisms like CAPTCHA that sometimes triggered in one of the browsers, resource load errors occurring in WebKit, and encoding differences across the different browsers. The results indicate that the majority of the embedded resources utilizing the Storage Access API exhibit consistent inclusions across all different browsers, with only minimal variations observed. In addition, we also proceeded with a detailed analysis of variations within different script groups, as outlined in Section 4.2.1. Our analysis revealed that external scripts using *hasStorageAccess* and *requestStorageAccess*, as well as inline scripts

[1] `https://tranco-list.eu/list/4Q39X/1000000`

		hasStorageAccess		requestStorageAccess	
	Total	Inline	External	Inline	External
Equal Sites	449	10	437	9	140
Total Sites	517	16	498	10	175
Equality Rate	86.85%	62.5%	87.75%	90%	80%

(a) Prior to manual verification

		hasStorageAccess		requestStorageAccess	
	Total	Inline	External	Inline	External
Equal Sites	514	15	495	10	174
Total Sites	517	16	498	10	175
Equality Rate	99.42%	93.75%	99.40%	100%	99.43%

(b) After manual verification

Table 5.1: Equality of SAA script inclusions across all evaluated browsers

using *requestStorageAccess*, demonstrated consistent API from 99.40 up to 100% of the visited sites. Notably, sites with inline scripts using *hasStorageAccess* displayed a lower consistency rate, with only a 93.75% equality rate. However, it is important to note that this particular group comprised only 16 sites, with just one site exhibiting differing behavior, which might limit the generalizability of the observed discrepancy.

In summary, our evaluation crawl indicates that Storage Access API inclusions are consistent across different browsers, with no significant discrepancies observed. Based on these findings, we adapted our crawling strategy to use a single browser, enabling us to increase site coverage within the same timeframe compared to a multi-browser strategy. Given Chromium's stability and its unique support for the Playwright *storage-access* permission, which should facilitate the detection of additional API calls via function hooking, we selected Chromium as our primary browser for our further crawls.

5.2 User Interactions

In our second evaluation crawl, we aimed to quantify the impact of simulated user interactions on both the frequency of Storage Access API inclusions and the actual calls of the API. This assessment seeks to determine whether the use of user interactions yields statistically significant variations in API calls compared to non-interactive crawls. Should our findings indicate a measurable increase in Storage Access API engagement when user interactions are implemented, we will adjust the crawling scope for our main crawl to incorporate these interactions, thereby enhancing the effectiveness of our data collection approach.

Type	Interaction	# Discovered sites across five crawls					
		Unique	Mean	Maximum	StD	≥80%	≥100%
Calls	✓	311	292	295	3.54	286	263
	×	305	295	298	2.42	295	278
Inclusions	✓	2,822	2,700	2,716	12.70	2,692	2,538
	×	2,798	2,707	2,711	3.93	2,713	2,606

Table 5.2: Discovered sites with at least one SAA inclusion or call during interaction evaluation

5.2.1 Experiment Scope

In contrast to the multi-browser evaluation, we did not see the need to verify any collected data manually in this evaluation. Therefore, we decided to sample a bigger dataset of 20,000 sites from the Tranco ranking[2] [100] to crawl for in the evaluation. Specifically, we sampled 10,000 sites from the Top 10,000, 6,000 sites from ranks 10,000 to 100,000, and 4,000 sites from ranks beyond 100,000. Furthermore, as mentioned in the previous section, we only performed the crawl with Chromium, as the browser evaluation did not show any significant differences between the tested browsers. While we conducted only a single crawl in the browser evaluation, we will perform this evaluation five times each, both with and without user interactions, to strengthen our findings. This repeated approach increases the reliability of our results, helping to confirm whether observed differences are statistically meaningful rather than due to crawling inconsistencies.

5.2.2 Result

In our interaction evaluation, we collected a total of 46,837 Storage Access API inclusions across all five crawls, with 23,378 occurring with interaction and 23,459 without interaction. Additionally, we identified a total of 3,117 API calls, of which 1,552 were made with interaction and 1,565 without interaction. This indicates that crawling without user interaction did not negatively impact the total number of inclusions and calls discovered during the crawl; in fact, it slightly improved the results.

Additionally, we also measured the amount of different sites that were discovered by the two approaches. Table 5.2 presents a detailed breakdown of the statistics regarding the discovered sites utilizing the SAA. While the crawls utilizing user interactions found slightly more unique sites for both API calls and inclusions among the five crawls, the average number of discovered sites slightly increased when no interactions were employed. By looking at the standard deviation, as well as the sites that were discovered in more than 80% or 100% of the crawls, we can also see that using no interactions at all performed

[2] `https://tranco-list.eu/list/4Q39X/1000000`

way better in terms of consistency. Using no interaction found more sites that were present in 80% or 100% of the crawls and additionally had a smaller standard deviation, especially for API inclusions.

These insights suggest that while crawling with user interactions may uncover a slightly higher number of unique sites across multiple crawls, it performs less consistently compared to crawling without interactions. The number of discovered sites fluctuates more with interactions, and fewer sites are consistently identified across multiple crawls. Given that one of our research questions focuses on tracking changes in the API landscape over time through repeated crawls, consistency is a crucial factor. High variation between crawls could negatively impact our findings related to this research question. For this reason, we decided to exclude user interactions in future crawls and instead proceed without them to maintain consistency in our data collection.

Chapter 6

Study Scope

In this chapter, we will present the scope of our large-scale study that we conducted using our web crawler to collect real-world data and address the research questions outlined in Section 1.1. We will detail the creation of the crawling dataset, describe the crawling strategy employed, and explain the crawler configuration and timeframe of our crawls. Additionally, we will provide an overview of the sites and pages successfully crawled without errors, offering a comprehensive view of the dataset that we use in our analysis.

6.1 Website Dataset

Our study involved crawling a dataset of 100,000 unique sites sourced from the Tranco list [100] generated on 25 June 2024[1]. We visited up to ten pages on each site, resulting in a potential maximum of 1,000,000 crawled pages. To build a representative sample for diverse popularities, we selected four ranking ranges and sampled 25,000 sites from each:

- Top 25,000: Represents the most prevalent and frequently visited sites on the web, which are essential to include for their influence and widespread use.
- 25,001 – 100,000: Capturing a mix of popular sites that are still prominent but slightly less visited.
- 100,001 – 500,000: Representing a broader range of moderately popular sites, which may have more specialized content.
- 500,001 - 4,395,999: Including less popular, potentially unique sites with lower traffic from the rest of the ranking, adding valuable diversity for a comprehensive dataset.

[1] `https://tranco-list.eu/list/4Q39X/full`

This approach ensured a diverse representation of web domains across different popularity levels, which is essential for achieving generalizability in our findings. The crawl was conducted five times in total to build a comprehensive dataset that would allow us to also compare the temporal changes in the Storage Access API landscape over the given time period.

6.2 Crawling Strategy

As stated in Chapter 5, based on the results of our crawling strategy evaluation, we chose to conduct our main crawl using a single browser without any user interactions. We selected Chromium for this crawl, as it is the most stable browser supported by Playwright and currently the only one that supports the *storage-access* permission. This feature is critical for increasing the likelihood that websites will invoke Storage Access API functions during the crawl, as described in Section 4.1.2.2.

6.3 Crawler Configuration

For each page visited during the crawl, we waited five seconds after the page had fully loaded before moving to the next page. If a page took longer than 30 seconds to load, the crawler timed out and proceeded to the following page. Additionally, the crawler was set to automatically restart if it remained inactive for 600 seconds. URL discovery was restricted to URLs that were same-site to the currently crawled site. We set the discovery depth to 2, meaning new URLs were identified on the landing page (depth 0) and on any pages directly linked from it (depth 1).

6.4 Timeframe

Our five crawls took place from **October 16, 2024, to November 17, 2024**. Figure 6.1 illustrates the complete timeline of the five crawls conducted for this study. We used our dataset consisting of 100,000 sites for all five crawls to ensure consistency and to eliminate variations that could arise from visiting different sites across the crawls. While crawls one, two, three, and four were performed in quick succession, there were a total of eleven days between the fourth and the final crawl. Initially, we started the fifth crawl three days after the fourth crawl. After analyzing the crawl results, it became evident that an issue occurred during this crawl. The number of sites that timed out increased by over 2 000 compared to previous crawls, and the collected data significantly deviated

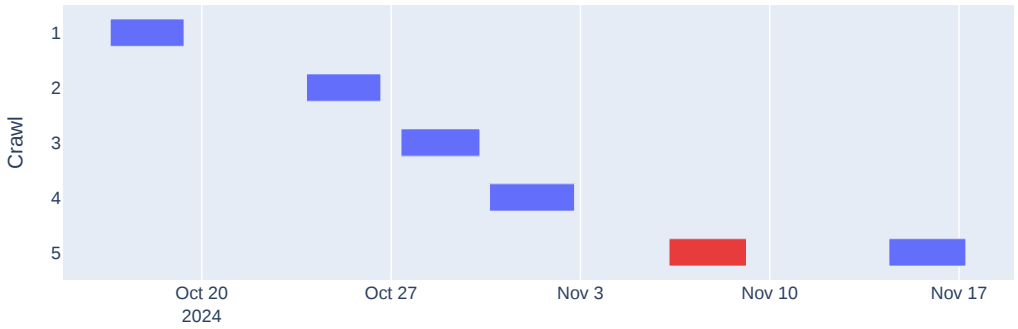


Figure 6.1: Crawling timeline

	№ of Crawl					
	1	2	3	4	5	Mean
Sites	68,864	68,665	68,476	68,344	68,482	68,566
Pages	511,504	511,182	507,877	508,448	505,201	508,842

Table 6.1: Successfully crawled sites and pages

from the patterns observed in earlier iterations. These discrepancies strongly indicated a malfunction during the crawl. Consequently, we decided to repeat the fifth crawl on November 14, 2024, more than two weeks after completing the fourth crawl. Both the failed (red) and the successful (blue) iterations of the fifth crawl are highlighted in the timeline for clarity.

6.5 Successfully crawled Sites and Pages

Although we created a dataset containing 100 000 sites to visit a total of 1 000 000 pages, the total number of successfully crawled sites and pages was lower. This discrepancy is due to several factors. Some pages encountered loading issues caused by errors such as certificate validation failures, timeouts, or network connectivity issues. Additionally, for certain sites, the landing page was unavailable for various reasons. For example, Content Delivery Networks (CDNs) do not serve content on the landing page and only provide content at specific paths. As a result, no additional pages could be discovered or accessed for these sites, leading to no pages being crawled at all in such cases. The status code of the visited page had no influence on the determination of a successful visit; a site visit was deemed unsuccessful solely based on whether it failed to load. Table 6.1 presents the number of successfully crawled sites and pages for each crawl, along with the average across all crawls. On average, we successfully crawled 68 566 sites and 508 842 pages. However, the data shows a consistent decline in the number of successfully crawled sites and pages throughout our crawling process, with the exception of the last crawl, where

the number of successfully crawled sites experienced a marginal increase compared to the previous ones. According to the collected data, this continuous decline was primarily due to an increasing number of sites timing out during the crawl. To address **RQ1**–**RQ3**, we selected the first crawl as it successfully gathered data from the largest number of sites and pages among all the crawls. For **RQ4**, which examines changes in the API landscape over time, we will use data from all crawls combined to capture the temporal trends in API usage and their privacy risks. Throughout this thesis, if we talk about the number of crawled sites and pages, we always refer to the number of successfully visited sites and pages.

Chapter 7

Overview of SAA Usage

This chapter presents the results of our analysis of general SAA usage on the web, focusing on inclusions, API calls, and the number of sites utilizing the API. It provides a comprehensive overview of API adoption across the web, addressing **RQ1**. As detailed in Section 6.4, our findings are based on data collected during the initial crawl conducted from October 16, 2024, to October 19, 2024. This crawl proved to be the most stable among all attempts, achieving the highest number of successfully crawled sites and pages, thereby serving as a robust foundation for our analysis.

7.1 Inclusions

First of all, we analyzed the script inclusions utilizing the SAA that we uncovered during the crawl with the help of string matching on the embedded scripts. However, while these numbers give a good indication of how frequently the API is being included, they may not fully capture the diversity of scripts using the API. To complement this, we also evaluated the number of unique scripts, as this provides a better understanding of how many distinct scripts are integrating the API across the web.

7.1.1 Unique Scripts

Table 7.1 summarizes the total number of unique scripts included during the crawl, categorized by type (inline or external) and the API function they utilize. Since some scripts include both functions, they are counted separately for each function, resulting in the total number of unique scripts being lower than the sum of the scripts associated with each specific function. In our analysis, two scripts are considered distinct if their hash differs. In total, we discovered 1,181 unique scripts, of which 91.28% were external

Script Type	hasStorageAccess	requestStorageAccess	Total
External	960	824	1,078
Inline	78	75	103
Total	1,038	899	1,181

Table 7.1: Number of discovered unique scripts

scripts and 8.72% were inline. This distribution indicates that the API is predominantly utilized in external scripts, with relatively few occurrences in inline scripts. Furthermore, for external scripts, the number of scripts using the *hasStorageAccess* API function (960) exceeds those using the *requestStorageAccess* function (824), reflecting a difference of 15.25% between the two SAA functions. Due to the high prevalence of external unique scripts, this trend remains consistent when considering all unique scripts, with a similar disparity of 14.35%. Overall, we consider the amount of unique scripts integrating API functionality pretty low, as we discovered only a total of 1,181 distinct scripts on the 511,504 pages that we crawled, resulting in an average of 0.0023 unique scripts per page.

7.1.2 Total Inclusions

After analyzing the unique scripts, we also examined the total number of inclusions as scripts are often included multiple times, not just once. These repeated inclusions can significantly affect the usage statistics and distribution of included API functions, offering a different perspective compared to the analysis based solely on unique scripts. Again, the total number of inclusions might differ from the addition of inclusions for specific functions as the included scripts might contain both functions. Overall, we recorded 184,160 inclusions among the 511,504 successfully crawled pages, resulting in an average of 0.36 inclusions per page. The trend of most API functionality being included in external scripts, as seen with unique scripts, also extends to the total inclusions (see Table 7.2). Our analysis indicates that 83.80% of all inclusions occurred within external scripts, vastly outnumbering those in inline scripts. Furthermore, the data highlight another trend observed with unique scripts: inclusions utilizing *hasStorageAccess* (174,932) significantly outnumber those using *requestStorageAccess* (26,169), resulting in a difference of 147.95% in favor of *hasStorageAccess*. This difference is even more pronounced than for unique scripts, suggesting that scripts employing the *hasStorageAccess* functionality are included far more frequently than those utilizing *requestStorageAccess*.

Inclusion Context	Script Type	№ of Inclusions		
		hasStorageAccess	requestStorageAccess	Total
Top-Level	External	66,564	10,160	70,435
	Inline	1	21	21
Framed	External	78,680	14,421	83,886
	Inline	29,687	1,537	29,818
Total	External	145,244	24,581	154,321
	Inline	29,688	1,558	29,839

Table 7.2: Number of inclusions grouped by context, script type, and SAA function

7.1.3 Inclusion Context

By grouping the inclusions based on their context, we also analyzed the differences between inclusions occurring directly in the top-level context and those within a framed context. Since the SAA functionality can only be utilized inside frames, as the top-level context by default has access to the cookies that the SAA would grant access to, we anticipated that the majority of inclusions would occur inside framed documents. Surprisingly, Table 7.2 shows that 70,456 inclusions, accounting for over 38.26% of all recorded inclusions, happen directly in the top-level context. For external scripts, this percentage is even higher at 45.64%, coming close to those included within a framed context. Additionally, nearly all top-level context inclusions occur in external scripts, with only 0.03% of instances involving inline scripts.

Furthermore, we also calculated the level in the frame hierarchy in which the inclusion happened. As mentioned in Section 4.1.3.1, we stored the frame hierarchy of all websites that we visited, which allowed us to restore all parent frames of a specific embedded frame. Therefore, we were able to calculate the number of parents of a frame that included Storage Access API resources. The frame level at which the inclusion occurs is referred to as the inclusion level. Figure 7.1 presents the number of inclusions per inclusion level, displayed on a logarithmic scale. The results confirm that 38.26% of all inclusions occur at level zero, corresponding to the top-level context. However, the majority of inclusions, totaling 58.90%, are still observed at level one. This indicates that these inclusions occurred in documents directly framed by the top-level site. Beyond level one, the number of inclusions sharply decreases with each subsequent level. While the highest inclusion level recorded is six, the combined number of inclusions at levels four, five, and six is minimal, accounting for only 0.02% of all inclusions. If the incorrectly attributed inclusions at level zero were removed, first level inclusions would account for nearly 95.40% of all inclusions. This highlights that when the Storage Access API is used correctly within framed documents, inclusions almost exclusively occur at the first level.

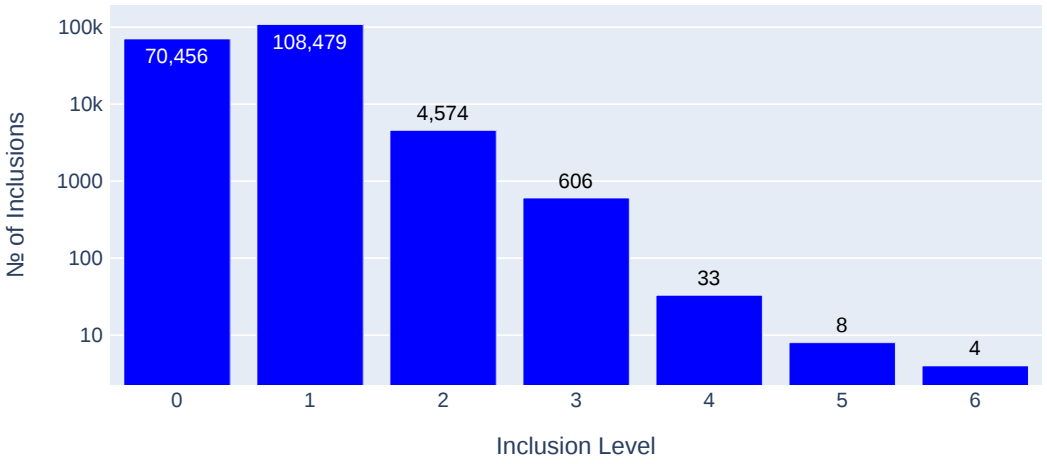


Figure 7.1: Level of SAA Inclusions

7.1.4 Inclusion Frequency of Unique Scripts

Next, we examined the distribution of unique scripts based on the number of times they were included. Figure 7.2 illustrates this distribution by categorizing scripts into different inclusion ranges. The majority of unique scripts are included only once across all crawled pages, with a significant portion included between two and ten times, as well as within the 10–99 inclusion range. Together, these three groups account for 94.41% of all unique scripts, indicating that many scripts utilizing the API are used sparingly across the web. In contrast, only 55 scripts are included between 100 and 999 times, suggesting broader adoption across multiple sites. Furthermore, only a small number of scripts have extremely high inclusion counts, with six scripts appearing between 1,000 and 9,999 times, and three scripts appearing more than 10,000 times. This trend is underscored by the median of six inclusions, which is significantly lower than the average (155.94), indicating that most scripts are included far fewer times than the average. We also found that scripts with fewer than 100 inclusions make up 94.41% of all unique scripts but only account for 5.41% of all inclusions. Conversely, scripts with more than 100 inclusions represent just 5.59% of all unique scripts but contribute to 94.59% of all inclusions. Notably, the three most included scripts alone are responsible for 65.91% of all inclusions in our crawl. This imbalance further highlights the unequal impact of a few predominantly included scripts relative to the numerous infrequently included ones.

Furthermore, we evaluated the differences in function usage of unique scripts based on the number of times they were included. Figure 7.3 shows that scripts included fewer than 99 times show a similar distribution between the two API functions, with a marginally higher number of scripts using *hasStorageAccess* compared to *requestStorageAccess*. However, this trend changes dramatically when considering scripts with more than 1,000 inclusions.

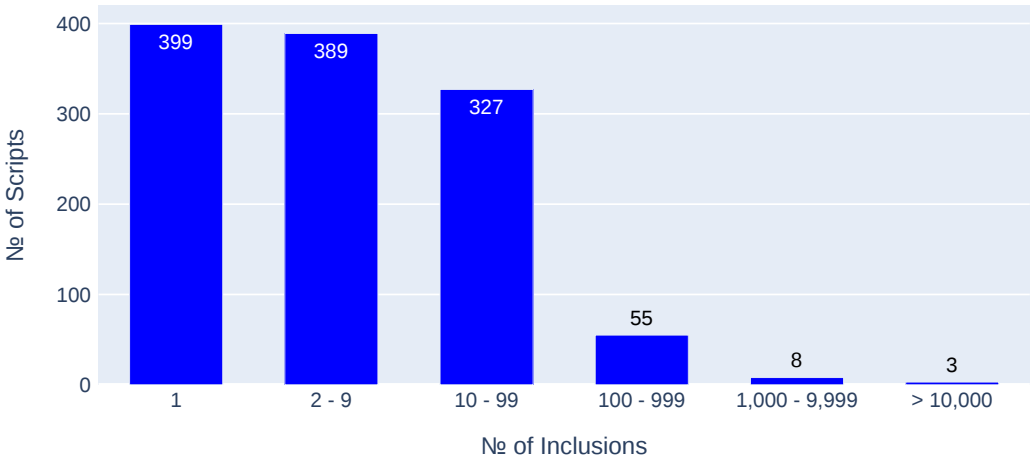


Figure 7.2: Distribution of unique scripts by number of times they are included

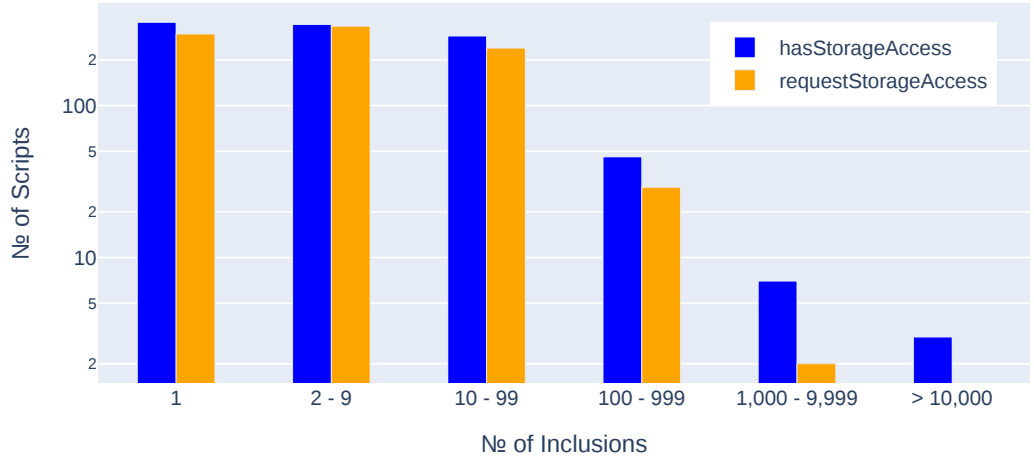


Figure 7.3: Function inclusions of unique scripts by number of times they are included

For these scripts, there is a significant increase in the use of *hasStorageAccess*, with only two out of eleven scripts (18.18%) utilizing *requestStorageAccess*. Furthermore, all six of the scripts with the highest inclusion counts exclusively rely on *hasStorageAccess*. Given that scripts with more than 1,000 inclusions are responsible for 85.14% of all inclusions, they have a substantial impact on the predominant use of *hasStorageAccess* across all inclusions.

7.2 Calls

Next, we focus on the actual API calls that we recorded during our crawl with the help of function hooking. This analysis provides valuable insight into how frequently the Storage Access API is actively used, as opposed to simple inclusions, which only indicate that a script contains the API functionality and may invoke it at some point. However,

Call Context	№ of Calls		
	hasStorageAccess	requestStorageAccess	Total
Top-Level	279	16	295
Framed	14,921	515	15,436
Total	15,200	531	15,731

Table 7.3: Number of calls grouped by context and SAA function

due to limitations in our crawling setup, we are unable to trace an API call back to the specific script responsible for it. Additionally, we cannot retrieve information about parent frames or the exact level at which the call occurred in the frame hierarchy, limiting our ability to conduct an analysis as detailed as we did for script inclusions.

7.2.1 Total Calls

In total, we recorded 15,731 API calls across 14,880 distinct documents from all the crawled sites. Of these calls, 96.62% were to *hasStorageAccess*, while only 3.38% were to *requestStorageAccess*. This distribution mirrors the trend observed in API inclusions, where *hasStorageAccess* is the predominantly utilized function; however, the difference is even more pronounced for API calls. It's worth noting that this increase in disparity compared to inclusions could also be influenced by limitations in our crawling setup, which does not simulate user interactions. As some sites may conceal calls to *requestStorageAccess* behind user interactions, despite permission having already been granted by our setup, the number of calls to *requestStorageAccess* that we discovered might be lower than for a real user visiting the same website.

7.2.2 Call Context

Although we were not able to calculate the inclusion level for API calls, our data still enabled us to identify the context (framed or top-level) in which a Storage Access API call was made, similar to how we analyzed it for inclusions. We compared the URL of the frame making the API call with the URL of the top-level page. If the URLs matched, the call most likely occurred in the top-level context; otherwise, it took place in a framed context. Although this may lead to false positives, as the top-level page might contain frames that share the same URL, we consider the scenario of a page framing itself highly unlikely and, therefore, disregard this case.

While 38.26% of all inclusions happened in the top-level context, as mentioned in Section 7.1.3, the proportion is significantly lower for API calls. Specifically, we found that only 295 calls were made in the top-level context, representing just 1.88% of all API

calls. This demonstrates that while Storage Access API functions are frequently included in the top-level context, actual calls to the API rarely occur there. Instead, the majority of API calls, with 15,436 and 98.12% of all calls, take place within framed documents. Additionally, the API calls occurring in the top-level context almost exclusively involve the *hasStorageAccess* function, with only 5.42% of calls affecting *requestStorageAccess*. For calls in the framed context, the share of *requestStorageAccess* is even lower at 3.34%.

7.3 Sites

Lastly, we analyze the number of sites utilizing or including the Storage Access API. For this analysis, we distinguish between two different types of contexts in which the SAA might be used:

1. **Top-level sites** that **do not use the SAA itself**, but rather include **framed documents** that utilize the API for their services. In the rest of the thesis, we will refer to these sites as "top-level SAA sites". Additionally, any mention of these sites including or calling the API refers specifically to occurrences within a framed context.

2. **Sites that are framed as part of another page** and **implement the API itself** to either request access or query the current access status. These sites may not be part of our initial website dataset as they might have only been discovered in a framed context. In the rest of the thesis, we will refer to these sites as "framed SAA sites".

For this part of the analysis, we also only considered API inclusions and calls that occurred inside framed documents to exclude top-level SAA sites that use the API incorrectly in the top-level context. In this case, these sites would already have unpartitioned access without calling the API, and therefore, the use of the API would not have any effect. Also, a site is already considered to use the API if it includes or calls the API at least once during our crawls.

7.3.1 Top-Level SAA Sites

Firstly, we examine the number of top-level SAA sites and their pages that were found to include or call the API at least once during the crawl. Although these sites do not use the API itself, they still give an indication of how many sites incorporate third-party content that relies on the SAA, providing valuable insight into the prevalence of API-related dependencies on top-level sites.

	Inclusions			Calls			Crawled
	hSA	rSA	Total	hSA	rSA	Total	
Sites	15,107	2,819	15,225	3,046	179	3,151	68,864
Pages	70,828	7,667	71,219	13,878	891	14,527	511,504

Table 7.4: Number of discovered top-level SAA sites and pages categorized by SAA functions used in the framed SAA sites

7.3.1.1 Total Sites

In total, we identified 15,225 top-level SAA sites and 71,219 of their pages that included at least one SAA resource within a framed document. This represents 22.11% of the sites and 13.92% of pages that we successfully crawled and results in an average of 4.68 pages per site including the API. When focusing only on sites where at least one of the framed SAA sites called an API function, the numbers are significantly lower, just as observed for the general inclusions and calls. We only registered 2,980 sites with at least one call among the framed SAA sites, accounting for just 4.33% of all successfully crawled sites. On these sites, 13,877 pages—or 2.71% of all crawled pages—contained a call to the API. For top-level SAA sites that exhibit calls to the API, we observe a similar average of pages per site (4.61) compared to inclusions. This suggests that while top-level SAA sites are more frequently observed in terms of inclusions, whenever a top-level SAA site makes an API call on one of its pages, nearly all of its other pages with an API inclusion also call the API. Conversely, the top-level SAA sites that include the API but do not call it often exhibit this behavior among every page. Furthermore, Table 7.4 once again presents a large disparity in function usage for both inclusions and calls, with sites highly favoring the use of *hasStorageAccess* over *requestStorageAccess*.

7.3.1.2 Landing Page Usage

In order to differentiate between inclusions and calls happening on the landing page and non-landing pages of top-level SAA sites, we stored an additional *landing_page* field for the call and inclusion tables inside the database. By using this information, we revealed that 8,793 top-level SAA sites included Storage Access API scripts on their landing pages. In comparison, 13,113 sites included API scripts on non-landing pages, meaning that the proportion of top-level SAA sites with inclusions on non-landing pages is 49.13% higher. Likewise, 1,596 sites made API calls from landing pages, compared to 2,681 from non-landing pages, representing a 67.98% higher amount of sites calling the API on non-landing pages. These findings demonstrate that more top-level SAA sites are including the API on non-landing pages both for script inclusions and function calls.

	Inclusions			Calls		
	hSA	rSA	Total	hSA	rSA	Total
Sites	585	163	629	57	13	60
Pages	59,098	6,919	59,541	15,091	128	15,174

Table 7.5: Number of discovered framed SAA sites and pages

On the other hand, the measurements reveal no significant differences between the two page types when analyzing the average number of inclusions and calls on pages that include or invoke at least one API function. Landing pages of top-level SAA sites include an average of 1.64 API inclusions, which is comparable to the average of 1.58 for non-landing pages. Similarly, function calls average 1.10 per landing page, only marginally lower than the 1.11 observed on non-landing pages. This suggests that while top-level SAA sites use the API more frequently on non-landing pages, the average number of inclusions and calls per page remains consistent across both page types, assuming they utilize the API.

7.3.2 Framed SAA Sites

Next, we analyze the usage of the SAA for sites and pages that are framed within another page and actually implement and utilize the API to check or request access to unpartitioned cookies. This analysis provides insights into how widespread the use of the API is in framed contexts across the web. Additionally, we particularly emphasize sites utilizing the *requestStorageAccess* method, as these sites are crucial for the privacy risk analysis which we will discuss in Chapter 9.

7.3.2.1 Total Sites

Overall, we discovered a total of 113,704 frames that included Storage Access API scripts directly inside the frame's context. These frames originated from 59,541 different pages across 629 distinct sites. Furthermore, we uncovered a total of 29,850 distinct framed sites across the crawl, meaning that 2.11% of all framed sites utilized the SAA. Additionally, Table 7.5 also shows the difference regarding the function usage on these sites and pages. As already expected, *hasStorageAccess* is significantly more prevalent; specifically, *hasStorageAccess* is included in 93.00% of the framed SAA sites. In contrast, *requestStorageAccess* appears in only 25.91% of these sites. When considering the actual API calls, the disparity is even more noticeable. *hasStorageAccess* was invoked on 57 framed sites, accounting for 95.00% of all framed SAA sites, while *requestStorageAccess* was called on just 13 sites, representing a mere 8.13% of framed SAA sites.

	Site	Frames
1	google.com	62,099
2	yandex.com	26,292
3	vimeo.com	9,109
4	recaptcha.net	3,414
5	livechatinc.com	2,531
6	facebook.com	2,177
7	yandex.ru	1,825
8	hcaptcha.com	1,113
9	bugherd.com	359
10	go.com	346

(a) Total

	Site	Frames
1	vimeo.com	9,109
2	facebook.com	2,177
3	google.com	1,243
4	hcaptcha.com	1,113
5	bugherd.com	359
6	zalo.me	301
7	beehiiv.com	272
8	stripecdn.com	162
9	blogger.com	119
10	shopify.com	111

(b) *requestStorageAccess*

Table 7.6: Most framed sites including SAA

7.3.2.2 Most Framed Sites

Total Frames Table 7.6 shows ten of the most framed sites including the SAA overall (Table 7.6a), as well as those specifically using *requestStorageAccess* (Table 7.6b). In the total frame count, *google.com* emerges as the most dominant, accounting for 54.61% of all frames, which is higher than the combined total of all other sites. The second-ranked site, *yandex.com*, already has less than half the number of frames, which shows a significant disparity between the first and the second ranked site. Beyond *Google*, other major technology companies are also featured in the top ten, including *Yandex* (ranks two and seven), *Vimeo* (rank three), and *Facebook* (rank six). Additionally, the top sites include two distinct providers of CAPTCHA services: *recaptcha.net* and *hcaptcha.com*. However, a notable shift occurs when focusing exclusively on frames utilizing *requestStorageAccess*. Google, which dominates the total frame count, falls to third place with a dramatic 98.00% reduction in frames. Several sites that also ranked highly when considering all API functions—such as *yandex.com*, *yandex.ru*, and *recaptcha.net*—drop out of the top ten entirely. On the other hand, *vimeo.com*, as well as *facebook.com*, *bugherd.com* and *hcaptcha.com* maintain their frame count, showing no decline when compared to their total frames, illustrating a consistent use of both *hasStorageAccess* and *requestStorageAccess* functionality. This consistency suggests that these sites might systematically request access on all pages they are framed on. Similar to Google's dominance in the total frame count, *vimeo.com* accounts for an impressive 57.08% of all frames using *requestStorageAccess*, underscoring its significant role in the usage of this specific API functionality.

Distinct Top-Level Sites In addition to analyzing the total frame counts, we also examined the number of distinct top-level sites that the framed SAA sites using *requestStorageAccess* were framed on. Figure 7.4 shows the distribution by presenting the

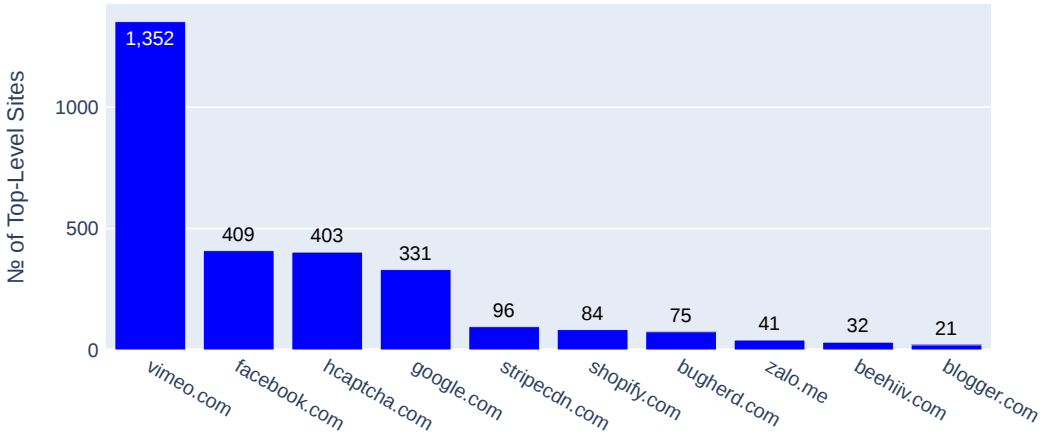


Figure 7.4: Distribution of framed SAA sites using *requestStorageAccess* by number of distinct top-level sites that embed them

ten sites with the highest count. The most frequently embedded site is *vimeo.com* with embeddings on 1,352 distinct top-level sites. No other site comes close, with the second-ranked site, *facebook.com*, already 69.75% lower at 409 top-level sites. Furthermore, the tenth-ranked site is embedded on only 21 top-level sites, highlighting the steep decline in embedding frequency across distinct top-level sites already beyond the top few domains. Most of the highly embedded sites also rank within the top 10,000 on the Tranco list, suggesting a strong correlation between high popularity and frequent embeddings as a framed SAA site. Additionally, these sites predominantly consist of global tech companies or social networks/online communities. With Facebook and Google, we also have two sites in the top four that were previously found to be involved in third-party tracking [7]. Together, they are present in 3,420 frames across 812 distinct top-level sites when it comes to *requestStorageAccess* usage.

7.3.2.3 Same-Site Frames

At the time of our analysis, most browsers employed a double-keyed partitioning approach for cookies, where the second key corresponds to the top-level site on which the cookie was stored. Under this approach, same-site frames retain access to the same cookies as their second key will also match, eliminating the need for the use of the SAA in such cases. Therefore, similar to the API usage in the top-level context, we consider this use incorrect. While we expected this number to be pretty low, overall we observed 358 framed SAA sites that were at least once included on top-level pages that were same-site. This accounts for 56.92% of all discovered framed SAA sites which further underscores the low amount of framed SAA sites being included on several distinct top-level sites.

Nevertheless, all of these framed SAA sites were embedded exclusively on same-site top-level pages, with no framed occurrences of these sites on cross-site pages. To evaluate the privacy risks in Chapter 9, however, we did not exclude these sites, as in the future they could potentially be framed on other cross-site pages, enabling them to perform cross-site tracking. On the other hand, each framed SAA site that was embedded on more than one distinct top-level site did not have any embedding of frames on same-site pages.

Chapter 8

Comparative Analysis of SAA Usage

While we analyzed the general usage of the SAA in Chapter 7, we additionally conducted a comparative analysis of the API usage, examining variations across popularity rankings, regions, and website content to address **RQ2**.

8.1 Categorization of Sites

To facilitate the comparative analysis, we categorize the sites into distinct groups based on criteria such as regional characteristics, site content, and their popularity ranking. These groupings allow us to examine variations in Storage Access API usage across the categories, enabling a more nuanced understanding of API adoption and behavior.

Popularity In order to group the crawled websites into different popularity groups, we use the *Tranco* ranking [100], which we also used to sample our website dataset for our crawls. We decided to use the Tranco ranking for this task as it provides a ranking that is hardened against manipulation, which has been possible for other lists, such as the *Alexa* list or *Cisco Umbrella* [100]. Furthermore, it is especially suitable for research, as it allows researchers to reference the exact list used for their work, greatly improving reproducibility. In our case, we used the Tranco list generated on 25 June 2024[1] as already mentioned in Section 6.1.

Regions Secondly, we need to categorize our crawled webpages by region. We investigated two approaches: using the site's TLD country code or the IP address geolocation of the server that returned the response. We ultimately chose to use the TLD country code because we found the IP address geolocation to be inaccurate in several circumstances. Assume that a major website is accessible by users from all over

[1] `https://tranco-list.eu/list/4Q39X/full`

the world. Without delivering content from numerous locations, the website's operation would be inefficient and result in higher load times. As a result, we estimate that for larger websites, we will always obtain a geolocation close to the country where we have set up our crawling infrastructure, which may significantly influence the distribution of website regions and falsify the categorization. When we use the TLD to classify these websites, we can ignore this problem, as the TLD tells us exactly for which country the domain was registered, which is often also the target audience of the site. Furthermore, we can define sites without a country TLD as global. Unfortunately, websites sometimes use country-specific TLDs as abbreviations for other terms. For example, *.ai*, originally designated as the country TLD for Anguilla, is frequently used for websites related to artificial intelligence. However, such cases are disregarded in our analysis, as their impact on the results is expected to be minimal.

Website Content Lastly, we also need to group websites into different categories depending on the primary content type or functionality they provide to their users. To perform this categorization, we use Chrome's *Topics API* which was developed as an alternative to third-party cookies by providing advertisers with a categorization of the websites that a user visited [101]. In our case, instead of using the Topics API itself, we use the Topics API classifier, which can be accessed and queried using the Chrome browser under *chrome://topics-internals/*. The current version that we use to categorize the sites in our research is *version 5*. Although the API supports multiple levels of classification granularity, we only use the top-level categories, as incorporating deeper levels would result in an unwieldy number of categories. Furthermore, it allows us to analyze trends and patterns in the SAA usage across broad content types, facilitating clearer insights into its adoption.

8.2 Popularity

Popularity is a significant factor influencing the adoption and usage patterns of various web APIs. To understand its impact on Storage Access API usage, we evaluate usage patterns for different popularity ranges. To compare the collected data, we group the sites into four different popularity ranges as defined in Section 6.1 as part of the sampling process of our website dataset: High popularity (1–25,000), moderate popularity (25,001–100,000), low popularity (100,001–500,000), and minimal popularity (>500,000).

Although we sampled 25,000 sites from each popularity range, the number of crawled sites and pages differed. Table 8.1 shows that the moderate popularity range had the highest number of successfully crawled sites, closely followed by the high popularity range. The two lowest popularity ranges had a significantly lower number of crawled

	High	Moderate	Medium	Low
Sites	18,527	18,949	15,731	15,657
Pages	143,192	147,187	117,055	104,070

Table 8.1: Successfully crawled sites and pages per popularity range

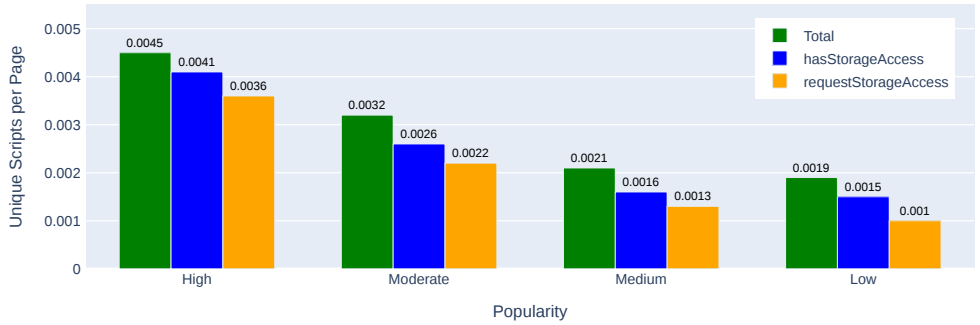


Figure 8.1: Average unique SAA scripts per page by popularity

sites, with around 3,000 fewer sites. Similarly, for pages, we observe the same order with significantly fewer pages being crawled in the medium and low popularities. For this reason, we cannot directly compare the total numbers, but instead must rely on relative values to make fair comparisons of the Storage Access API usage across the different popularity groups.

8.2.1 Script Diversity

First of all, we were able to uncover a correlation between website popularity and the SAA script diversity. Our data shows a clear downward trend: as the top-level site's popularity decreases, so does the average amount of unique SAA scripts per page. Figure 8.1 depicts that sites in the high popularity range include the most unique scripts on average across all popularity ranges, with 0.0045 unique scripts per page. From there on, the numbers drop continuously with decreasing popularity until reaching minimal popularity that averages just 0.0019 unique scripts per page. Overall, we see a decrease of 59.15% between high and minimal popularity sites. The continuous decline with decreasing popularity suggests a clear correlation between website popularity and the number of distinct API scripts included, which also holds when considering the specific API functions separately. This means that highly popular sites tend to include a wider variety of SAA scripts, likely reflecting their more complex functionalities and reliance on diverse third-party services. Conversely, sites in the minimal popularity range rely on a smaller set of unique scripts. Interestingly, while including the least amount of unique scripts, minimal popularity sites exhibit the highest average inclusions per page, which further indicates that they reuse the scripts more frequently than sites in any other popularity.

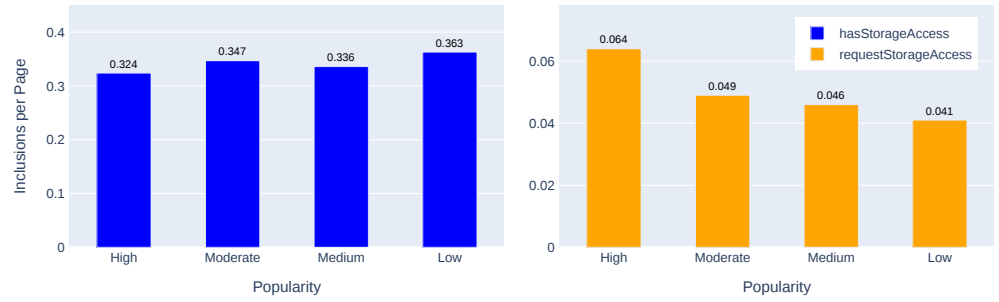


Figure 8.2: Average SAA inclusions per page by popularity

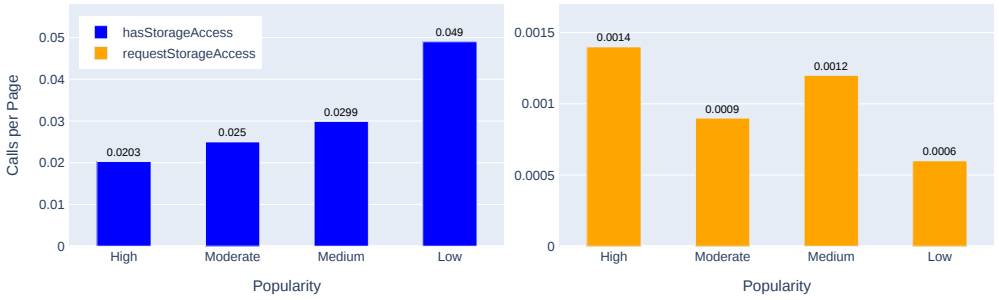


Figure 8.3: Average SAA calls per page by popularity

8.2.2 Function Usage

Next, we also observe a significant impact of a site's popularity on the usage of SAA functions, both in terms of calls and inclusions. As illustrated in Figure 8.3, minimal popularity sites exhibit a significantly higher average number of *hasStorageAccess* calls than any other popularity range. With increasing site popularity, this average steadily declines, reaching its lowest value for high-popularity sites. For *hasStorageAccess* inclusions, the decline is also observed but not consistent (see Figure 8.2). Nevertheless, minimal popularity sites again lead with the highest average inclusions per page, while high popularity sites show the lowest average. On the other hand, when examining *requestStorageAccess* inclusions and calls, we observe the exact opposite trend. In this case, high popularity sites show the highest average usage per page for both inclusions and calls, while minimal popularity sites present the lowest. These findings highlight a clear correlation between a website's popularity and its reliance on specific SAA functions. High popularity websites predominantly utilize the *requestStorageAccess* function to request access to unpartitioned cookies. Conversely, minimal popularity sites rely more heavily on the *hasStorageAccess* function instead.

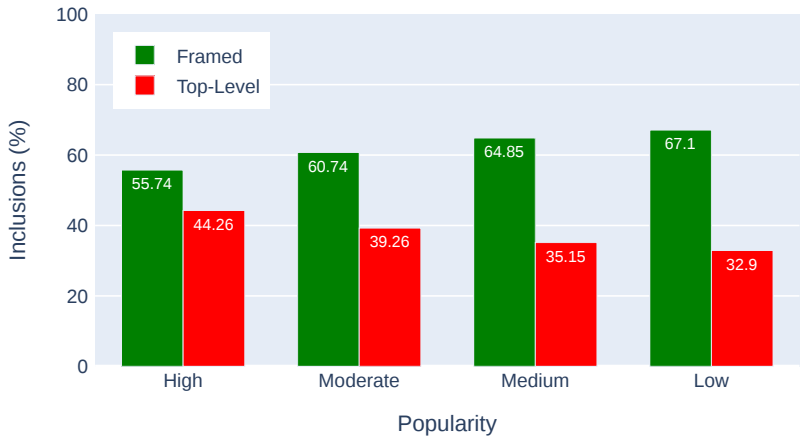


Figure 8.4: Inclusion context by popularity

8.2.3 Usage Context

Lastly, Figure 8.4 reveals a significant variation in the SAA inclusion context across different popularity ranges. High popularity sites exhibit the highest percentage of SAA inclusions in the top-level context, with 44.25% of all inclusions occurring in this context. Conversely, they also show the lowest amount of correct inclusions inside a framed context (55.74%). As site popularity decreases, the proportion of incorrect inclusions in the top-level context steadily declines. For minimal popularity sites, this percentage reaches its lowest point, with only 32.9% of inclusions occurring in the top-level context. These findings are unexpected, as we assumed that developers of high popularity sites would be more likely to adhere to best practices and not include the API directly in the top-level context. We assume that this trend may be linked to the increased number of framed SAA sites among higher popular websites. During our crawl, many of these high popularity framed SAA sites were accessed in a top-level context, as they were part of our dataset. As a result, we may have also visited pages that are typically embedded with SAA usage on other sites. If these pages do not adapt their content based on the context in which they are included, the SAA functions would still appear in the top-level context in such cases and increase the percentage of inclusions in the top-level context.

8.3 Region

Regional diversity is another key factor that might influence the adoption and usage patterns of the Storage Access API on the web. To understand its impact on Storage Access API usage, we evaluate how the origin of a website impacts the usage numbers. As mentioned in Section 8.1, we retrieve the origin of the website by the country code of their TLD. For websites that do not use a country-specific TLD, we classify them

Cluster	Crawled Sites	Crawled Pages	Country Codes
DZ	52	350	DZ, MA, EG, TN
TW	149	842	TW, HK
RU	3,270	21,278	RU, UA
BE	1,447	10,388	BE, NL, FR
PE	2,509	16,573	MX, PE, EC, PA, BO, GT, UY, CO, DO, CR, AR, CL, VE, BR
CA	2,300	16,069	US, AU, NZ, CA, UK, PH
TH	563	3,502	VN, ID, TH
JP	1,335	8,597	JP
KR	156	771	KR
NG	916	5,346	KE, IN, NG, ZA
ES	3,250	22,811	IT, PL, ES, TR, DE
Other	7,388	49,669	-
Global	45,529	355,308	-

Table 8.2: Regional clusters used for regional analysis

as *Global*. However, given the extensive number of country-specific TLDs present in our dataset, we further group countries based on similarities in their internet traffic as measured by Ruth et al. [102]. This approach enables us to also account for language regions in addition to geographical regions, as similar internet traffic patterns are often associated not only with geographical closeness but also with shared languages. To ensure completeness, country TLDs that do not belong to any cluster, as defined by Ruth et al., are categorized under *Other*.

Table 8.2 presents the regional clusters along with the corresponding country codes that belong to the cluster, as well as the successfully crawled sites and pages for each cluster. Additionally, Table A.1 provides a mapping of country codes to their corresponding country names. Although our approach does not assign every site using a country TLD to one of the clusters, 68.34% of these sites and 68.20% of their pages are assigned to one of the clusters.

8.3.1 Script Diversity

In terms of script diversity, two regions, *DZ* and *TW*, stand out by significantly exceeding the average number of unique scripts per page observed in other regions. Both regions exhibit averages exceeding 0.03 distinct scripts per page, a stark contrast to the overall average of 0.011. When these outliers are included, the standard deviation rises to 0.010, nearly equaling the overall average. This highlights the substantial variability in script diversity across regions, with the data deviating almost as much as the mean itself. Apart from *DZ* and *TW*, two additional regions, *TH* and *KR*, display slightly increased script diversity, with average unique scripts per page of 0.015 and 0.018, respectively. In

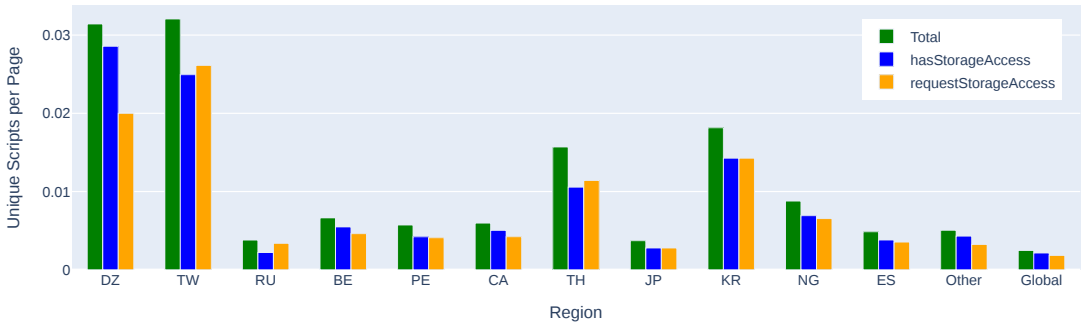


Figure 8.5: Average unique scripts per page by region

contrast, the remaining regions show relatively minimal variation, with averages ranging between 0.002 and 0.009. These regions also exhibit a much lower standard deviation of 0.001, indicating a more consistent and less diverse script usage pattern compared to the outliers.

8.3.2 Inclusions

Figure 8.6 presents the average inclusions per page across all clusters considered in this analysis, including those not part of any region (*Other*) and global sites. In total, the average inclusions vary substantially across regions, driven primarily by a significant outlier in the *RU* region, which far surpasses all others with a maximum of 1.14 and highlights a significant gap between it and the remaining regions. Although the standard deviation drops from 0.24 to just 0.09 indicating less variability when excluding *RU*, we still observe notable differences between 0.17 and 0.46 average inclusions among the remaining regions. Sites in the global cluster present 0.31 average inclusions, which is equal to the median across all regions, indicating that global sites represent a middle ground in terms of API script inclusions.

While the *RU* region dominates in overall inclusions, its performance shifts drastically when focusing exclusively on those utilizing *requestStorageAccess*, presenting the lowest average inclusions per page (0.03) and a large disparity between the total or *hasStorageAccess* inclusions and those specifically utilizing *requestStorageAccess*. While most regions exhibit only a marginally smaller number of *hasStorageAccess* inclusions compared to the total, the majority exhibit significantly lower averages for *requestStorageAccess* inclusions compared to their overall totals. Nevertheless, our results also show two regions, *TW* and *KR*, that utilize the SAA function on a more regular basis in more than 50% of their total inclusions. For *KR* the number of *requestStorageAccess* inclusions even surpasses those that utilize *hasStorageAccess*. In addition, to the differences in the total inclusions, these results further highlight significant differences in function usage across the regional

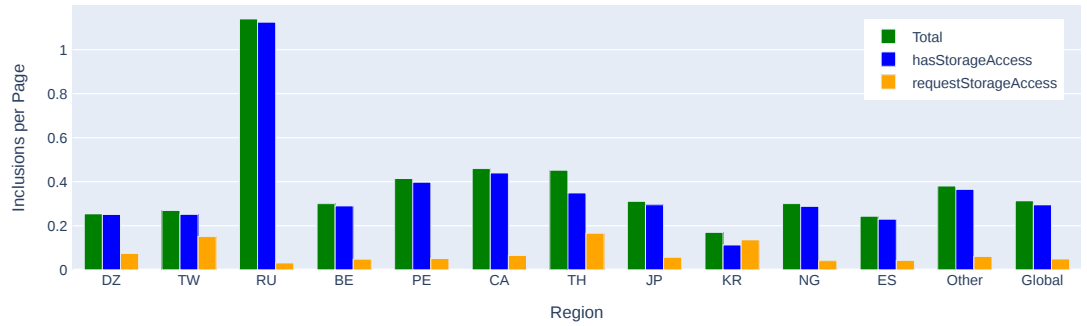


Figure 8.6: Average SAA inclusions per page by region

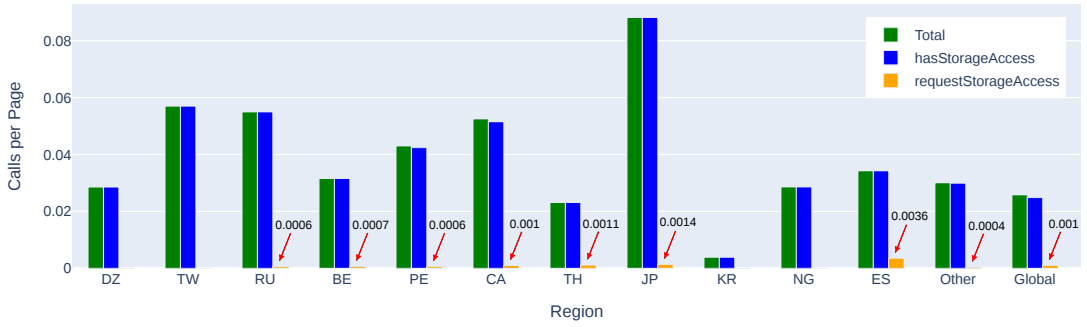


Figure 8.7: Average SAA calls per page by region

clusters. Regions such as *TH* and *KR* place greater emphasis on *requestStorageAccess*, while others, such as *RU*, almost purely rely on *hasStorageAccess*.

8.3.3 Calls

Similarly to inclusions, we also analyzed the average calls per page by the site's underlying region. Figure 8.7 illustrates the average calls per page for each region cluster. For calls, we identified one region (*JP*) that significantly outperformed others, similar to how *RU* dominated inclusions. Crawled sites within the *JP* cluster recorded an average of 0.088 calls per page, substantially exceeding the second-ranked region *TW* (0.057). *RU*, which registered by far the most average inclusions per page, ranks third with 0.055 calls per page. While each cluster exceeded 0.02 calls per page, *KR* was the only exception registering the lowest number with just 0.004 calls per page, which results in a 95.58% decrease between the maximum and minimum average call rate. Overall, the data shows a standard deviation of 0.02 which accounts for more than half of the average, indicating a significantly high level of deviation across the regions.

Additionally, our data indicates that for four out of the 13 regional clusters no calls to the *requestStorageAccess* function were recorded, making it impossible to calculate an

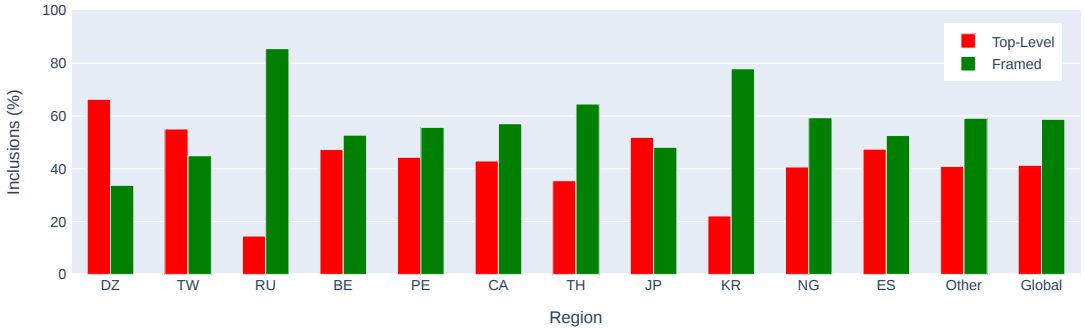


Figure 8.8: Inclusion context by region

average for these regions. Interestingly, despite being one of the regions with the highest *requestStorageAccess* inclusions, the *KR* region did not register any calls to the SAA function. However, this lack of recorded calls could be attributed to the limited number of pages sampled from these regions. If our dataset had included a larger sample of pages from these clusters, we assume that at least one call would have been observed for these regions. In addition, we also once again observe one region that outperforms the others. While the SAA function was called between 0.0006 and 0.0014 times on average per page for most of the regions, the *ES* region recorded 0.0036 inclusions per page, significantly outnumbering the second-highest call rate.

8.3.4 Usage Context

We did not only observe differences in the average number of SAA inclusions and calls but also in their usage context. Overall, the proportion of inclusions incorrectly occurring in the top-level context ranged from 14.52% to 66.29%, resulting in a substantial span of 51.77% between the minimum (*RU*) and maximum (*DZ*). However, for most regions, the proportion of top-level inclusions deviates by only 10% or less from the overall proportion calculated in our general analysis in Chapter 7. Additionally, the majority of sites demonstrate a higher prevalence of framed inclusions compared to top-level inclusions, with the notable exceptions of *DZ*, *TW*, and *JP*, where top-level inclusions surpass framed ones.

8.4 Website Content

Lastly, in addition to popularity and region, the primary content of a website may also influence the adoption and usage of the Storage Access API. Some content categories may rely more heavily on embedding third-party services that utilize the API, potentially

Category	Crawled Sites	Crawled Pages
Arts & Entertainment	17,524	79,389
Unknown	11,170	59,161
News	10,087	55,438
Business & Industrial	7,180	36,107
Internet & Telecom	6,403	25,738
Jobs & Education	5,895	33,800
Computers & Electronics	5,602	27,014
Online Communities	5,358	24,620
Shopping	4,087	24,784
Games	3,654	17,556
Finance	3,526	16,306
Travel & Transportation	2,629	15,020
Law & Government	2,580	13,343
Sports	2,204	13,725
Autos & Vehicles	1,960	11,189
Books & Literature	1,881	9,234
People & Society	1,776	9,952
Food & Drink	1,752	10,793
Hobbies & Leisure	1,340	7,277
Home & Garden	1,210	7,613
Beauty & Fitness	1,120	7,235
Real Estate	630	3,682
Pets & Animals	432	2,528

Table 8.3: Crawled sites and pages per primary content category

contributing to higher adoption rates in those categories. To explore this, we categorize the websites in our dataset based on their primary content type using the Topics API classifier, as detailed in Section 8.1. This classification allows us to assess whether certain types of websites exhibit higher API adoption compared to others. To avoid overcomplicating the analysis and to ensure meaningful comparisons, we limit the categorization to first-level granularity.

Due to the inherent imbalance among different categories on the web, the categorized sites exhibit a skewed distribution, with many categories containing only a small number of sites. In contrast, the *Arts & Entertainment* category accounts for the most sites in our website dataset with more than one-fourth of all sites. Additionally, 16.22% of sites could not be classified by the Topics API classifier and ended up in the *Unknown* cluster. Table 8.3 presents the total amount of successfully crawled sites and pages per primary content category.

8.4.1 Script Diversity

When examining the unique scripts, our data shows differences among the primary site content of the embedding top-level sites. Figure 8.9 shows a maximum value of 0.015 unique scripts per page in total, observed in the *Pets & Animals* category, which is five times higher than the minimum value of 0.003 recorded for *Unknown* sites. Most categories feature fewer than 0.01 unique scripts per page, with four outliers exceeding this threshold. The skewed distribution is highlighted by the high standard deviation of 0.003, which represents 42.73% of the overall average and indicates significant differences

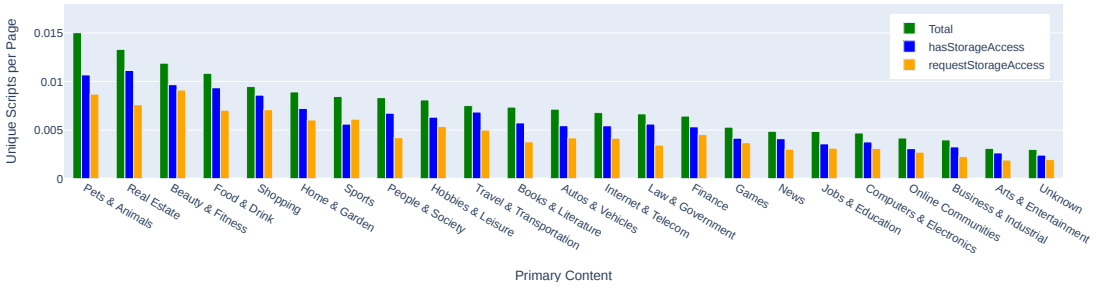


Figure 8.9: Unique scripts per page by primary site content

	Minimum	Maximum	Average	StD
hasStorageAccess	0.0209	0.0451	0.0296	0.0066
requestStorageAccess	0.0000	0.0028	0.0009	0.0007
Total	0.0220	0.0470	0.0303	0.0067

Table 8.4: Statistics of average calls per page across primary site contents

in SAA script diversity across the different content categories. The low number of unique scripts for *Unknown* sites can likely be attributed to the nature of these sites, which predominantly belong to lower-popularity tiers. Due to their limited popularity, these sites often feature simpler designs with minimal functionalities and are more likely to deliver static content without relying on third-party frames that utilize the API. In contrast, categories with higher script counts—such as *Shopping* and *Food & Drink*—tend to incorporate third-party frames for features like advertising, payment systems, and analytics tools, which are often provided by external services and may involve SAA usage in a framed context. Interestingly, the two categories with the highest average of unique scripts (*Real Estate* and *Pets & Animals*) are those for which we crawled the fewest sites, 630 and 432, respectively. If more sites from these categories were included in the dataset, the average number of unique scripts could potentially decrease due to a larger and more diverse sample.

8.4.2 Calls

Although no significant differences in SAA inclusions are observed across the various content categories, a notable variation emerges in API call rates. Table 8.4 depicts that the average SAA calls per page range from 0.0470 to 0.0220, reflecting a difference of 72.46% with an average of 0.0303. The standard deviation of 0.0067 (22.11% of average) underscores the moderate variation between the average call rates. The data also contains two outliers, *Food & Drink* and *Pet & Animals*, each exceeding 0.04 calls per page. However, it is important to note that the *Pets & Animals* category is based on a relatively small number of crawled sites. As a result, the observed value may not be

fully representative of the category, and the true average could differ. When the outliers are excluded, the standard deviation drops to 0.0049 indicating a lower variability across the remaining categories.

Calls to *hasStorageAccess* exhibit a distribution similar to the total average calls. While the minimum (0.0209) and maximum (0.0451) values are slightly lower than the total calls, the average also decreases marginally to 0.0296. Notably, the standard deviation remains almost unchanged at 0.0066, indicating that the variability in *hasStorageAccess* calls closely aligns with the overall API call pattern. On the other hand, calls to *requestStorageAccess* show significant differences to the total average calls. First of all, the standard deviation is much higher at 77.78% of the average which indicates a significantly large deviation between the average call rate to the function. Furthermore, we also observe two categories that do not record any calls. Specifically, the *Pets & Animals* and *Real Estate* categories stand out as having no calls to this API. However, it is important to acknowledge that these two categories are based on the smallest number of crawled sites, which could account for the absence of recorded calls. If a larger number of sites in these categories had been crawled, it is possible that calls to the API might have been observed.

Chapter 9

Privacy Risk Analysis

After examining the general usage of the SAA, this chapter focuses on the privacy risks associated with websites that use *requestStorageAccess* to request unpartitioned storage access when embedded within other top-level sites. This analysis will consequently address and answer **RQ3**. To assess the privacy risk, we utilize the *Privacy Risk Score* introduced in Section 4.3. The score calculates a site's tracking risk based on the percentage of first-party advertising cookies it uses and its position in the Tranco ranking.

9.1 Cookie Classification

As mentioned in Section 2.1.2, cookies can be classified into four distinct categories which provide a coarse overview regarding the functionality of cookies. These classifications include: strictly necessary, performance, functionality, and targeting/advertising. This classification system provides a framework for analyzing the cookies collected from sites utilizing *requestStorageAccess*. Specifically, it enables us to calculate the proportion of cookies used for advertising purposes on a given site, which we will use as part of our tracking risk calculation.

In order to connect the collected cookies with the corresponding category, we retrieved the classification information from cookie databases that were available on the web. After evaluating multiple cookie databases on the web such as *Cookiepedia*[1], *Cookiedatabase*[2] and *CookieSearch*[3], we stuck to *Cookiepedia*. This decision was made as the categories on Cookiepedia closely followed the classification system that we wanted to use. Although the other databases also used similar classification systems, they did not fully match

[1] `https://cookiepedia.co.uk/`
[2] `https://cookiedatabase.org/`
[3] `https://cookiesearch.org/`

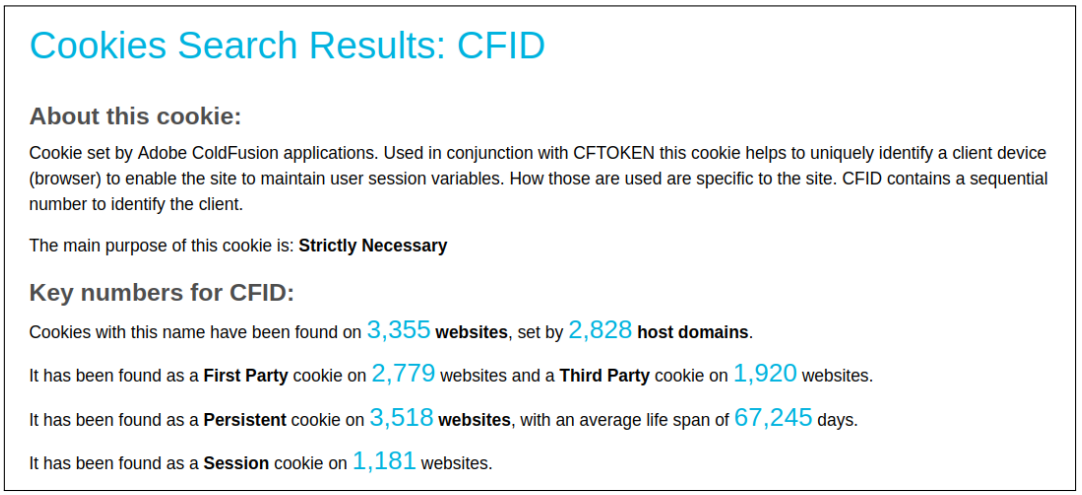


Figure 9.1: Cookie classification on *Cookiepedia*

with *Cookiepedia*, which is also why we were not able to combine multiple databases. To extract the classification data from *Cookiepedia*, we queried the site and parsed the HTML content to obtain the relevant information about each cookie. Figure 9.1, for example, shows the *Cookiepedia* result for the "CFID" cookie that we discovered during the crawl. However, some cookies lacked entries in *Cookiepedia* and were therefore categorized as "Unknown". Notably, several of these unclassified cookies appeared on numerous sites. To address this, we conducted additional manual research by consulting additional resources, such as the cookie policies provided by websites, to classify the cookies that were found on over 50 different sites. This threshold was selected because below this limit, the proportion of "Unknown" cookies rose sharply, which would have substantially extended the time required for manual analysis.

9.2 Cookie Distribution

During our cookie collection, we discovered in total 920 unique cookies across 109 sites. This corresponds to 66.87% of all the sites that we uncovered to include *requestStorageAccess* in a framed context and potentially using the API to request unpartitioned cookie access. For the remaining 54 sites, either the landing page was unavailable, or there were no cookies found. Additionally, the landing pages of some sites had no links to other pages, which did not allow us to continue crawling the site.

During the manual cookie classification process, we categorized a total of 21 cookies. Among these, 7 were classified as *Advertising* cookies, 7 as *Functionality* cookies, and 7 as a *Performance* cookie. Figure 9.2 shows the classification of the unique cookies after both the automatic and manual classification. The distribution of cookies shows significant variation across categories. *Unknown* cookies constitute the largest group,

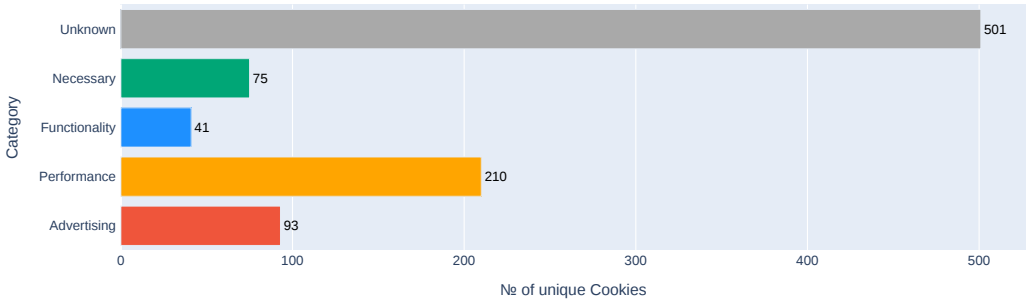


Figure 9.2: Collected cookies by category

accounting for 54.57% of all unique cookies. *Performance* cookies follow as the second most prevalent category, making up around 22.72%. In contrast, *Advertising* cookies (10.11%) and *Necessary* cookies (8.15%) are much less common, with *Advertising* slightly outnumbering *Necessary*. *Functionality* cookies, at 4.46%, represent the smallest category in the dataset. *Performance* and *Advertising*, the categories with the highest overall counts of classified cookies, also dominate the top 5 most frequently used cookies, highlighting the prevalence of these two categories in the most recurrent cookies.

However, when examining the average number of sites on which a unique cookie is used per category, we find that the averages for all categories—except *Unknown*—range between 1.97 and 2.17, with *Advertising* cookies having the highest average. In contrast, *Unknown* cookies have an average of only 1.17 sites per unique cookie. This indicates that while we found more unique *Unknown* cookies overall in our crawl, they tend to be used less often than cookies of other categories, and in many instances, just on a single site.

9.3 Classified Cookies per Site

Table 9.1 presents the statistics of cookie classification percentages across sites, both before and after the manual classification. Notably, the average percentage of classified cookies increased from 45.94% to 60.07%, reflecting an overall improvement of 14.13% following manual classification. Furthermore, the median also increased by 10.61% from 50.00% to 60.61%, indicating that after manual classification more than half of the sites feature more than 60% of classified cookies. The standard deviation also slightly increased after manual classification, once again suggesting a slightly enlarged deviation due to the increased classification percentage. However, despite these improvements through manual classification, some sites still have no classified cookies, preventing us from assessing their privacy risks. Without classified cookies, we cannot calculate the percentage of *Advertising* cookies they use, preventing a meaningful analysis of their

	Before Manual Classification	After Manual Classification
Mean	45.94%	60.07%
StD	27.20%	28.06%
Min	0%	0%
Max	100%	100%
25%	60.00%	80.00%
50%	50.00%	60.61%
75%	27.78%	42.86%

Table 9.1: Statistics of successfully classified cookies across sites

privacy implications. Still, 75% of the sites have at least 42.86% of their cookies classified after the full classification process. Furthermore, 25% of sites have classified at least 80% of their cookies, enabling a more reliable assessment of privacy risks with marginal error.

9.4 Privacy Risk

As mentioned in Section 4.3, the Tracking Risk Score formula combines key attributes into a normalized score between 0 and 100, providing a quantifiable measure of a site's risk of tracking a user utilizing the SAA. The formula integrates the percentage of advertising cookies and the Tranco rank of a website, weighted by their respective importance. For the calculation, we will assume that the proportion of advertising cookies remains unchanged even if the unclassified cookies are subsequently classified. To account for uncertainty introduced by unclassified cookies, we will also calculate the upper and lower bound error for each site. The upper bound error is determined by assuming that all unclassified cookies are also advertising cookies, thereby maximizing their contribution to the percentage of advertising cookies. Conversely, the lower bound error assumes that none of the unclassified cookies are advertising cookies, minimizing their contribution. Including this range in our results will provide a more comprehensive understanding of how unclassified cookies may impact the risk calculation, ensuring the results reflect the uncertainty caused by incomplete classification.

Furthermore, after calculating the TRS, we will also evaluate the number of top-level sites that embed the framed SAA sites exhibiting a high TRS. This metric is key to evaluating the potential scope of information collected if tracking is performed by the site. A site framed on a larger number of top-level sites gains broader reach, enabling it to aggregate data from diverse sources through cross-site tracking. This expanded data collection enhances the site's ability to construct more detailed and comprehensive user profiles, which significantly impacts user privacy.

9.4.1 Tracking Risk Score

We successfully computed the Tracking Risk Score (TRS) for 102 out of 109 sites. For the remaining 7 sites, we could not gather any classified cookies, making TRS calculation pointless. This is because the formula's *Advertising* cookie percentage component would have evaluated to zero, leaving the final score solely reliant on the Tranco rank, which, if considered alone, does not accurately reflect actual tracking behavior. Moreover, the combined error rate of the upper- and lower-bound errors would spike to 70, the maximum possible combined error, further underscoring the unreliability of the score in such cases. Among the 102 sites with calculated TRS, the average tracking risk was 24.37, with a median score of 21.23. This median indicates that over half of the sites exhibited a moderate level of tracking risk, with a significant number scoring below the average. The highest score, 81.92, was observed for *youtube.com*, reflecting a high likelihood of user tracking due to the platform's extensive use of *Advertising* cookies. In total, 6 sites had a TRS exceeding 50, and 4 of these surpassed 75, signifying a substantially high tracking risk. This suggests that these sites are likely engaging in active tracking practices. Conversely, only one site had a TRS of 0.00, as it contained no advertising cookies and was absent from the Tranco ranking, indicating no measurable tracking risk.

Table 9.2 provides detailed insights into the upper- and lower-bound errors associated with the calculated tracking risks. While both error types show a maximum error exceeding 55, their distributions differ significantly. The lower-bound error has an average rate of 5.49, with over 50% of the sites exhibiting an error of 2.22 or less. Only 5 sites (4.9% of the total sites) had an error exceeding 20, with the maximum observed at 56.00. Notably, the minimum lower-bound error of 0.00 was found in 44 sites (43.14% of the total sites), indicating that for many sites, the lower-bound error is negligible. This suggests that while a few outliers show high lower-bound errors, the vast majority of sites have minimal errors in this direction, making it unlikely that their calculated tracking risk is significantly underestimated due to unclassified cookies. In contrast, the upper-bound error shows more pronounced variability and higher error rates. The average upper error is almost four times greater than the lower error, at 19.57, with a standard deviation of 15.47 compared to 9.42 for the lower-bound error. This indicates a wider spread and a tendency toward higher errors in the upper-bound direction. Furthermore, more than half of the sites have an upper-bound error exceeding 20, highlighting the potential for underestimating the actual tracking risk for some sites due to a large proportion of unclassified cookies.

Figure 9.3 shows the top ten sites with the highest TRS based on our collected data. As mentioned earlier, *youtube.com* ranks first with the highest risk score of 81.92, highlighting a significantly increased risk of user tracking through the SAA. Additionally, the error for

	TRS	Upper Error	Lower Error
Mean	24.37	19.57	5.49
StD	17.33	15.47	9.42
Min	0.00	0.00	0.00
Max	81.92	57.65	56.00
25%	10.89	6.61	0.00
50%	21.23	19.88	2.22
75%	31.52	29.74	7.12

Table 9.2: TRS statistics with upper- and lower-bound error

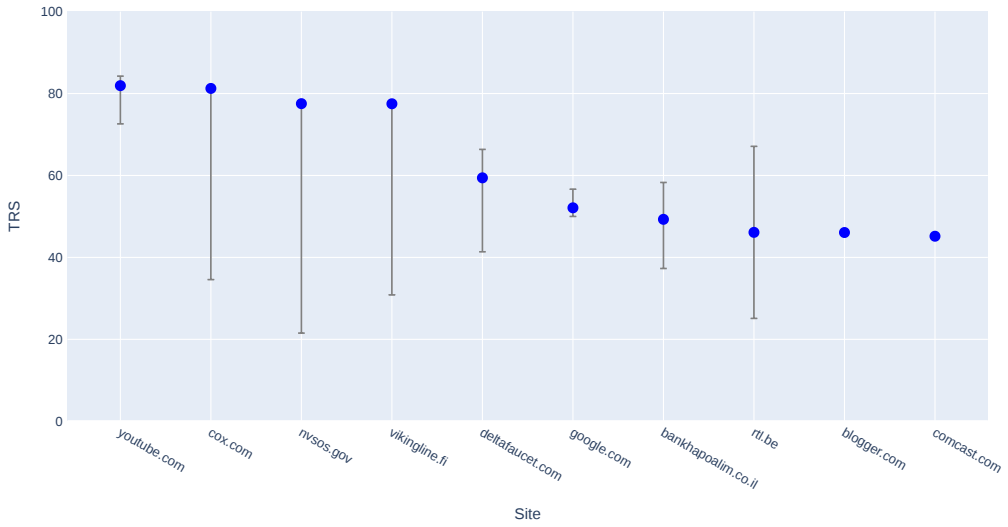


Figure 9.3: Top ten sites with highest TRS

youtube.com is relatively low, indicating that the calculated risk score is fairly accurate with minimal uncertainty in the measurement. Three other sites—*cox.com*, *nvsos.gov*, and *vikingline.fi*—also score similarly to *youtube.com*. However, these sites exhibit a much higher lower-bound error, suggesting that their actual tracking risk might be lower than the calculated scores. After these sites, the risk score drops significantly from 77.50 to 59.42 and continues to decrease to 45.17 by rank 10. Notably, *google.com*, ranked 6th, also exhibits a marginal error, similar to youtube.com. Additionally, two sites ranked ninth and tenth have zero error, indicating that all cookies on these sites were successfully classified. All four top ten sites with minimal error are also present in the Top 1,000 of the Tranco ranking, which suggests that their popularity may contribute to a higher classification of cookies in online databases.

9.4.2 Privacy Implications

To evaluate the potential privacy implications of sites using the API to request unpartitioned cookie storage, we analyzed the number of distinct top-level sites where they are

framed while utilizing *requestStorageAccess*. This metric reflects the potential effectiveness of cross-site tracking. The broader the range of top-level sites framing a given site, the greater its capacity to aggregate data from diverse sources. Such extended reach enables a site to collect more comprehensive user information, build detailed profiles, and track users more effectively, all of which carry significant privacy implications.

Among the sites with a Tracking Risk Score (TRS) exceeding 50—indicating a higher likelihood of tracking users—only *google.com* is framed on more than three top-level sites while using *requestStorageAccess*. Most other sites in this category are framed on only one top-level site, with a single exception framed on three. This suggests that while these sites may be more likely to track users, their ability to build extensive user profiles is limited due to the restricted number of distinct top-level sites where they are framed. For instance, sites such as *youtube.com*, *nvsos.gov*, *vikingline.fi*, and *deltafaucet.com*, each framed on only one top-level site, are restricted to tracking users within that context, effectively eliminating cross-site tracking potential. In contrast, *google.com*, which is framed on 331 top-level sites, has a significantly broader reach for potential cross-site tracking. However, despite this reach, *google.com* exhibits a TRS of only 52.11, which is lower than the top four ranked sites, whose scores are substantially higher.

Interestingly, other sites embedded across many distinct top-level sites tend to have moderate or low TRS values, suggesting minimal tracking risk. For example, *vimeo.com*, framed on 1,352 distinct top-level sites, has a risk score of only 21.91. Similarly, *zalo.me* and *bugherd.com*, embedded on 42 and 45 top-level sites respectively, have TRS values of just 14.18 and 18.86. The site with the highest risk score among those framed on more than 40 top-level sites is *shopify.com*, with a TRS of only 26.41.

These results suggest that, based on our TRS, sites framed on a large number of distinct top-level sites, despite having the potential for extensive cross-site tracking, exhibit only a low risk of user tracking, making tracking scenarios highly unlikely. Conversely, sites with significant risk scores are generally framed on a small number of distinct top-level sites, which greatly limits their ability to engage in comprehensive cross-site tracking. Among the analyzed sites, only *google.com* stands out as demonstrating a notable risk of user tracking while also being framed on a large number of different top-level sites while using *requestStorageAccess*. However, since *google.com* is present on only 0.48% of all successfully crawled sites, the likelihood of it building a comprehensive user profile is minimal, as it would require a user to visit multiple of these specific sites. Therefore, we assess the overall privacy risk to users posed by the SAA as minimal in the current state.

Chapter 10

Evolution of the SAA Landscape Over Time

To answer the last research question (**RQ4**) of this thesis regarding the evolution of the Storage Access API landscape over time, we analyze the changes in API usage and associated privacy risks across multiple crawls. By comparing data from all five crawls conducted between October 16, 2024, and November 17, 2024, as outlined in Chapter 6, we aim to identify trends and shifts in the SAA landscape over this brief period. Although the timeframe spans only about one month, it still provides valuable insights into how the landscape evolves over a short period and the potential impact of these changes on privacy risks. To clarify the distinction between the five crawls, we enumerate them as shown in Figure 6.1 from one to five, using this enumeration consistently throughout the text, figures, and tables.

As already mentioned in Section 6.5, the number of successfully crawled sites and pages differed across the five crawls that we conducted. More specifically, Table 6.1 illustrates that the number of crawled sites and pages consistently decreased across the crawls, with the exception of the last crawl, which showed a slight increase compared to the previous one. This decline was primarily driven by pages that were successfully crawled in previous instances but subsequently timed out during later crawls. Another contributing factor was that some sites' hostnames no longer resolved, preventing them from being crawled. Although the difference between the crawl with the lowest number of successfully crawled sites and the one with the highest was only 0.76%, we compared the collected data relative to the successfully crawled sites and pages to ensure a comparison that accurately reflects the evolution of the API landscape.

10.1 API Usage

To begin, we compared the API usage across all crawls to determine differences in API adoption among them. Specifically, similar to our general SAA analysis, we examined script inclusions, calls, and the number of sites utilizing the API. For the latter, we again further differentiated between top-level SAA sites and framed SAA sites.

Overall, the usage of the SAA remains largely consistent across all crawls that we conducted, with no significant differences observed between them. Still, we noted a marginal increase in usage over time between the crawls, especially for the usage of *requestStorageAccess*. This suggests that while the adoption of the SAA has not significantly shifted during the crawling timespan, the API landscape may be gradually and slowly evolving. Over time, more sites may adopt the SAA or frame other services that utilize the API, inducing slow but steady changes in its usage patterns. Nevertheless, it is important to consider that these differences were only marginal and could be influenced by crawling inconsistencies between the different crawls. Variations in the sites and pages crawled due to differences in URL discovery or timeouts during the crawl could impact the observed averages and subsequently explain the small fluctuations between the crawls. Due to the small differences observed in the data, we present the findings in the form of statistics across the different crawls.

10.1.1 Inclusions

Table 10.1 presents the statistical analysis of the discovered average inclusions per page across the different crawls, both in total and categorized by the included API function. The narrow standard deviation of 0.0032 (0.88% of the average) indicates an almost non-existent variability in the total inclusions. The minimum (0.3588) and maximum (0.3659) values further highlight this limited variation in the inclusion rates. Notably, while none of the first three crawls exceeded an average of 0.36 inclusions per page, the fourth and fifth crawls indicate a slight increase by surpassing this threshold, with average inclusions of 0.3659 and 0.3645, respectively. Overall, we observe an increase of 1.23% in total average inclusions from the first to the last crawl. When examining the SAA functions individually, *hasStorageAccess* shows a similar increase of 2.15% over the crawling period. In contrast, *requestStorageAccess* demonstrates a significantly higher growth, with a 15.38% increase, which may suggest a growing trend toward utilizing the API specifically to request unpartitioned cookie access and a shift in function usage. The most substantial difference for *requestStorageAccess* is observed between the fourth and last crawl, likely due to the longer time span between these two crawls.

	Minimum	Maximum	Average	StD
hasStorageAccess	0.3406	0.3480	0.3438	0.0033
requestStorageAccess	0.0510	0.0595	0.0528	0.0037
Total	0.3588	0.3659	0.3618	0.0032

Table 10.1: Statistics of average inclusions per page across all crawls

	Minimum	Maximum	Average	StD
hasStorageAccess	0.03069	0.03130	0.03098	0.00022
requestStorageAccess	0.00178	0.00191	0.00186	0.00006
Total	0.03226	0.03285	0.03250	0.00023

Table 10.2: Statistics of average calls per page across all crawls

10.1.2 Calls

We see a similar difference in the average calls per page compared to inclusions. Again, we only observe a difference of 1.51% between the first and last crawl, indicating a marginal increase in total call rates over the time in which we conducted our crawls. Table 10.2 further shows a low standard deviation of 0.00023 accounting for only 0.71% of the average calls per page, which further underscores the low variability of the data. Similar to inclusions, the small difference also persists when breaking down the average calls by the specific function that was executed. For *hasStorageAccess*, the number of average calls per page starts at 0.03088 and finishes at its maximum of 0.03130 during the final crawl, resulting in an overall increase of 1.35% from start to end. For *requestStorageAccess*, the highest average calls instead occurred during the third crawl. Nevertheless, we still observe an increase of 6.74% between the first and last crawl. While this upward trend for *requestStorageAccess* is smaller than for inclusions, it may still indicate that the amount of calls to the function is slowly increasing over time, with more sites shifting to calling the API in order to request unpartitioned cookie access.

10.1.3 Top-Level SAA Sites

Next, we analyzed the changes in the percentage of top-level SAA sites relative to all crawled sites. Top-level SAA sites with API inclusions showed a steady increase over our crawls, slightly outpacing the rise in average inclusions. Specifically, the percentage of top-level SAA sites increased from 22.11% in the first crawl to 23.06% in the final crawl, representing a modest growth of 4.3%. Moreover, we observed a notable jump in the percentage of sites including *requestStorageAccess* scripts between the fourth and final crawl, mirroring the trend observed for average inclusions. During our crawls, the percentage of top-level SAA sites using the *requestStorageAccess* rose from 4.16% to

5.32%, highlighting a significant uptick in the percentage of top-level SAA sites including this function.

10.1.4 Framed SAA Sites

To evaluate the changes in the framed SAA sites, we calculated the average number of distinct framed SAA sites per crawled page. This approach offers a consistent metric for comparing the amount of distinct framed SAA sites, even when the total number of crawled pages differs across the crawls. By normalizing the data in this way, we can more fairly assess trends in the inclusion of framed SAA sites and identify shifts in the distinct amount of sites using the SAA in a framed context. While most other metrics indicate a marginal increase in SAA usage, the average number of distinct framed SAA sites actually decreases over the course of the crawls. Overall, we observed a marginal decrease of 0.04%, from 0.12297 framed SAA sites per page to 0.12292. However, these minimal differences are likely negligible, as they are so small that they can be attributed to inconsistencies during the crawls. Nevertheless, when focusing solely on the framed SAA sites that utilize the *requestStorageAccess* function, we observe an increase of 6.21%. This suggests that, while the total average number of distinct framed sites remains stable, the number of sites using *requestStorageAccess* is gradually growing. This trend could be explained by framed SAA sites that initially used only *hasStorageAccess*, now adopting *requestStorageAccess* as well, thereby increasing the proportion of sites utilizing this function while keeping the total number of framed sites relatively unchanged.

10.2 Privacy Risks

Finally, we evaluated the impact of changes in API usage on the privacy risks by analyzing variations in the tracking risk score (TRS) across our crawls and considering the broader implications of these changes for user privacy. As discussed in Chapter 9, we collected cookies for sites that used *requestStorageAccess* within a framed context, which could indicate an intention to request unpartitioned cookie access. The cookies were collected when the site was visited at the top level to collect unpartitioned cookies to which the Storage Access API would give access if requested and permitted.

The number of sites using *requestStorageAccess* in a framed context varied slightly across crawls, ranging from a minimum of 154 to a maximum of 171. We also collected cookies multiple times from sites identified in several crawls, as their cookie usage may have changed since their initial discovery. However, we were unable to crawl certain sites where the landing page was unavailable. Additionally, some sites did not provide any

	1	2	3	4	5
Sites	102	95	102	87	95
Mean	24.37	27.56	26.85	27.13	26.82
StD	17.33	20.65	20.40	19.55	18.81
Min	0.00	4.32	0.03	0.10	0.00
Max	81.92	86.88	82.92	82.92	81.92
25%	10.89	12.47	11.28	12.85	11.72
50%	21.23	21.91	22.97	22.37	23.12
75%	31.52	32.18	34.45	32.22	34.38

Table 10.3: TRS statistics across all crawls

classified cookies, making the TRS calculation impossible for those. As a result, the number of sites for which we could calculate the tracking risk score ranged only from 87 to 102 among the crawls.

10.2.1 Tracking Risk Score

First, we again calculated the Tracking Risk Score for each site across all crawls. Table 10.3 presents the statistics of the calculated TRS for each crawl. The average tracking score varied across the different crawls. The first crawl recorded the lowest average TRS at 24.37, notably lower than the averages observed in the subsequent crawls. The other crawls all reported average scores of 26.82 or higher, with the second crawl reaching the maximum average score of 27.56. While the differences are smaller, a similar trend is observable in the median across all crawls. Similar to the average, the first crawl also recorded the lowest median score of 21.23. Interestingly, despite the second crawl having the highest average score, its median score of 21.91 was the second-lowest among all crawls. This discrepancy between the mean and median indicates that the second crawl likely included more outliers or sites with exceptionally high tracking risk scores, which skewed the average upward. Supporting this observation, the second crawl also recorded the highest maximum TRS at 86.88, exceeding the maximum value of any other crawl by 3.96. Additionally, the second crawl is also the only one where the minimum tracking score observed is higher than 0.10 with a score of 4.32. Apart from the first and second crawls, the differences between the other crawls were only marginal. Although the standard deviation differed slightly between 18.81 and 20.40 indicating a slight difference in the scattering of the data, the average tracking risk is almost equal for all of them with values between 26.82 and 27.13.

Figure 10.1 additionally shows the top 30 TRS scores for each of the crawls. From rank 10 onward, all crawls exhibit a consistent trend: the TRS values are almost equal and steadily decline from approximately 50 at rank ten to around 30 by rank 30. However, notable differences emerge between ranks 5 and 10, where each crawl experiences a

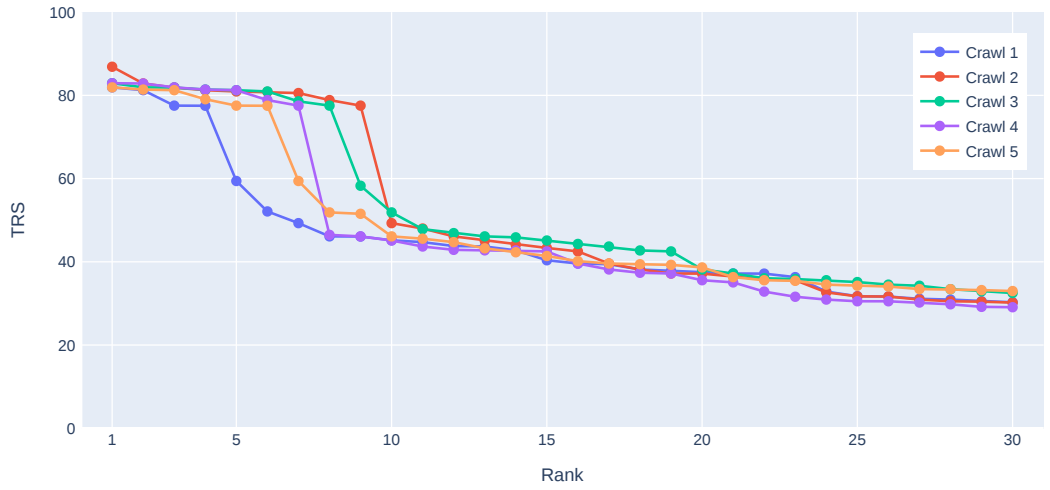


Figure 10.1: Top 30 TRS scores by crawl

significant drop in TRS at varying points. For the first crawl, the scores begin to drop earlier, starting at rank four, indicating that fewer sites in this crawl achieve very high TRS values. This early drop aligns with the first crawl's lower overall average TRS. In contrast, the second crawl stands out with the most TRS values exceeding 75, contributing to its position as the crawl with the highest average score. Notably, the second crawl also records the highest TRS at rank one (86.87), whereas all other crawls report similar top scores of approximately 82. These results suggest that over the course of the crawling period, the tracking risk scores remained relatively stable from rank 10 onward, with no significant changes observed. The notable differences, however, appear between ranks 5 and 10, where the scores vary more prominently. Interestingly, no clear trend over time emerges: the first and second crawls recorded both the highest and lowest numbers of high-risk sites in this ranking range, while subsequent crawls settled in between these extremes. One possible explanation for these differences is the variation in the discovered sites across the crawls. Since the examined sites were not identical across all crawls, the first crawl missed some high-risk sites that were captured in later crawls, contributing to the earlier drop in TRS values. Conversely, the second crawl observed the highest number of high-risk sites, leading to a later drop in scores.

10.2.2 Privacy Implications

While the Tracking Risk Score reflects the likelihood of sites tracking users via the Storage Access API, we also assess the broader privacy implications regarding cross-site tracking. A site embedded across a significant number of distinct top-level sites has the potential to facilitate more efficient cross-site tracking. By accumulating user data from a diverse

range of sources, such a site can construct a more comprehensive user profile. While this may benefit the tracking party, it constitutes a significant violation of user privacy.

Our findings in Section 9.4.2 revealed that during the first crawl, only one site, *google.com*, combined a high TRS (greater than 50) - indicating a strong likelihood of tracking users - with a notable presence across different top-level sites. Specifically, *google.com* was embedded on 331 different sites while using *requestStorageAccess*. A similar pattern emerged in the fifth crawl: once again, *google.com* exhibited a TRS of 51.54, suggesting a likelihood of user tracking. However, the number of top-level sites where *google.com* was embedded slightly decreased to 323. Importantly, in both cases, no other site combined the two critical factors, high tracking risk and widespread embedding, which would be necessary for efficient cross-site tracking. On the other hand, in crawls one through three, we did not observe any site that posed an increased risk of user tracking while also being embedded on a significant number of distinct top-level sites. Although *google.com* remained embedded across a large number of distinct sites during all three crawls, its Tracking Risk Score (TRS) never exceeded 48.00 in these crawls, falling below the threshold indicative of an increased likelihood of tracking.

These results suggest that there were no significant changes in the privacy implications over the span of our crawling period compared to our privacy risk analysis in Chapter 9. While *google.com* only indicated a privacy risk in two out of five crawls, we still assume that the site might use the API to track its users, as previous research already showed that *Google* is the most prevalent tracking party across the web [7].

Chapter 11

Discussion

In this chapter, we present and discuss the results of our empirical study outlined in the previous chapters. Additionally, we address the limitations, as well as ethical considerations of our work, and explore potential directions for future research that can build upon our findings.

11.1 Prevalence of the SAA

At the time of this thesis, the number of unique scripts incorporating SAA functionality remains relatively low, suggesting a limited diversity of such scripts across the web. Furthermore, we observe the same trend with distinct framed SAA sites. Nevertheless, we found that almost one-fourth of all top-level sites in our crawl included at least one API script, indicating a fairly widespread adoption in terms of inclusions across sites. This high usage is primarily driven by a small number of distinct scripts that are heavily included. While most unique scripts appeared only once throughout the crawl, a few were reused extensively, with more than 10,000 inclusions. Notably, these heavily reused scripts accounted for over 94% of all inclusions, suggesting that API functionality is predominantly used by a small set of scripts. Additionally, these scripts mainly rely on the *hasStorageAccess* function, which significantly influences the overall landscape of API function usage. Despite the high inclusion numbers, actual API calls were much less frequent, occurring at only one-fifth of the inclusion rate. This suggests that the API may often be included in scripts but is invoked only sporadically, limiting its active usage.

Throughout all of our crawls, API usage remained relatively stable, showing no significant fluctuations. However, over the entire crawling period, we observed a gradual upward

trend. Specifically, there was an average increase of 1.23% in inclusions per page and 6.47% in calls per page from the first to the last crawl. This growth was largely driven by an increasing number of *requestStorageAccess* inclusions and calls across the crawled pages, signaling a shift from a landscape predominantly reliant on *hasStorageAccess* toward a more balanced usage of both methods. In addition, it also suggests a gradual rise in API usage during the crawl timeframe, further supported by the growing percentage of top-level SAA sites. Nonetheless, it should be noted that these differences are relatively minor and could potentially be attributed to inconsistencies in the crawling process.

11.2 Usage Context

The SAA is designed to address situations where third-party content embedded within a framed context needs access to its unpartitioned cookies. Since top-level sites inherently have access to these cookies, the API's functionality is specifically intended for use in framed contexts. Despite this, we found that a significant portion with 38.26% of all inclusions occurs incorrectly in the top-level context. In contrast, only 2.00% of API calls are made inappropriately in the top-level context, suggesting that incorrect usage is far less common for API calls than for inclusions. We attribute the high rate of incorrect inclusions in the top-level context to one or more of the following factors:

- **Multi-purpose scripts**: Top-level sites often include versatile scripts that serve multiple functionalities, some of which may include SAA functions. As a result, the API may be included frequently in the top-level context without the intention of actually using the API functionality. In this case, the functionality would be rarely invoked, limiting the occurrence of incorrect calls.
- **Developer awareness**: Some developers may not fully understand that the SAA is specifically designed for use within framed contexts. This lack of awareness could also lead to the unintended inclusion of API functions in the top-level context.
- **Multi-context loads**: Many pages that are embedded as frames can also be accessed directly as top-level pages. If these pages do not differentiate between top-level and framed loads, the content remains identical in both contexts. As a result, SAA resources are included unnecessarily at the top level, leading to incorrect inclusions.

11.3 Comparative Analysis

In Chapter 8, we observed notable differences in SAA usage when comparing top-level sites, based on factors such as their popularity, region, and primary content type. These differences were observed across various aspects of SAA usage, including script diversity, function usage, average inclusions, average calls, and the inclusion context. Script diversity varied noticeably across all three analyses, indicating a strong dependence on a site's popularity, regional characteristics, and content type. This suggests that the diversity of SAA scripts is not consistent across the web but is influenced by the specific properties of each site. Moreover, our findings show that a site's region has a more substantial impact on average inclusions and calls compared to the content category. In the regional analysis, we observed notable outliers in both inclusions and calls that significantly outperformed other regions. In contrast, the content category analysis revealed a more balanced distribution of average calls, with fewer substantial outliers, highlighting that region plays a more prominent role in influencing SAA usage than content type. Similarly, the region also has a more significant impact on the inclusion context compared to popularity. In the regional analysis, we observed a highly skewed distribution of the inclusion context, with some regions exhibiting far higher inclusion rates in the top-level context than others. In contrast, the popularity analysis demonstrated a more consistent, marginal decline in inclusion rates across different popularity ranges. This further highlights that regional factors are also more influential in shaping inclusion contexts than the site's popularity.

11.4 Privacy Risk

While the API may provide tracking parties with possibilities to perform cross-site tracking even after the introduction of storage partitioning, we observed that this might not be the case at all in the current state of the API. During our privacy risk analysis, we calculated the TRS of every framed SAA site that included the *requestStorageAccess* function at least once during the crawl. We observed that the far majority of these sites depict a tracking risk score of less than 50 indicating that it is more likely that the site does not employ tracking mechanisms. Overall, only 5.88% of framed SAA sites actually retrieved a TRS of more than 50. Furthermore, the sites that indicated the likelihood of tracking behavior mostly were not framed on more than three different top-level sites, in most cases even only on one. Therefore, these sites often did not have sufficient capabilities to perform efficient cross-site tracking and build a comprehensive user profile. The only exception to this trend is *google.com* which was the only site that exhibited both a likelihood for tracking with a TRS score of more than 50, as well as a

significant number of top-level sites where it was embedded. Nevertheless, the overall percentage of top-level sites where *google.com* was framed on was limited to just 0.48% of the crawled sites, indicating a low likelihood that users might visit multiple of these sites and fall victim to cross-site tracking. These results also did not significantly differ throughout the crawls. While we observed a slight difference between the TRS of the sites, especially for those sites exhibiting a high TRS, we did not observe another site with a high possibility to abuse the API for cross-site tracking. Subsequently, we consider the privacy risk of the API moderately low.

11.5 Limitations

While our study provides valuable insights into the usage and implications of the SAA, it is important to acknowledge certain limitations that may have influenced our findings. First, as mentioned earlier, our crawler did not simulate user interactions when visiting sites, as initial tests showed that the interactions we performed during evaluation did not significantly increase the observed API usage. Although we granted the *storage-access* permission to all frames permitting them to use the API, sites may still hide API calls behind user interactions. This limitation may have led to an underestimation of actual API usage, particularly in scenarios where user engagement is typically required to trigger the calls, such as calls to *requestStorageAccess*. Second, our crawls were conducted exclusively from a single geographic location within the university network, which could have influenced the results in two key ways: (1) Prior research has shown that the location from which a crawl is conducted can significantly impact the outcome [88]. Therefore, crawling from additional vantage points across different regions would improve the generalizability of our findings. (2) The university network used for our crawls has also been employed in previous studies by other researchers involving web crawling. This history might have influenced how some sites responded and they potentially blocked our requests completely or hid the content behind a bot prevention mechanism such as *CAPTCHA*. This could have further restricted our ability to comprehensively analyze the API's usage across affected sites.

In addition to limitations in data collection, the study also faced limitations in the privacy risk analysis. A notable issue was the classification of cookies, with a significant portion of them labeled as *Unknown*. Although we tried to counteract it by manually classifying the ones that are highly used, it still introduced a degree of uncertainty into our TRS calculations. To account for this, we represented the uncertainty using lower and upper error bounds, ensuring transparency about the potential variability in our results. Furthermore, the purpose of a cookie is not always identifiable by its name as

sites may use cookies with the same name for different purposes. This may have also affected our results as this means that the classification might not always reflect the real-world usage.

11.6 Ethical Considerations

Conducting web crawls inherently introduces additional load to the websites being accessed. While this load is unlikely to pose significant issues for highly popular sites that routinely handle large volumes of traffic, smaller websites may experience negative impacts on their performance. This could disrupt the user experience, potentially frustrating visitors who might choose to leave the site. Such disruptions could harm the site's reputation, especially if users perceive the slow performance as a persistent issue. To mitigate these risks, we implemented measures to minimize the load imposed on all crawled websites. Our crawling approach was carefully designed to visit pages sequentially, ensuring that at no time multiple crawler threads were accessing the same site simultaneously. By maintaining this controlled approach, we aimed to reduce the likelihood of noticeable performance degradation and to uphold ethical standards in our data collection process.

11.7 Future Work

In our work, we primarily focused on the usage of the Storage Access API, examining script inclusions, calls, and the number of sites employing the API. However, future work could also investigate other components of the API ecosystem, such as the Permissions Policy header integration, which enables site administrators to restrict the use of *requestStorageAccess* for embedded frames and their nested children. Further studies could examine how frequently the Permissions Policy header is employed to limit API functionalities and the contexts in which it is most commonly used. Moreover, as mentioned in Section 2.4.7.2, the SAA recently received a draft proposing an extension to its functionality [69] by expanding its coverage to include non-cookie storage mechanisms such as *localStorage* or *IndexedDB*. Once this extension is widely adopted across the web, future research could examine the impact of the extended API functionality on usage patterns, including a detailed analysis of which storage types are most commonly accessed using the API. Additionally, comparing these findings with our current results, which focused on the API's usage when limited to cookie-based storage, could provide a comprehensive view of how the extension influences adoption and application. Such

comparative analyses would provide valuable insights into how these changes shape the role of the API in web development.

Apart from these technical investigations, researchers could also delve into the usability and comprehensibility of the privacy implications communicated through SAA permission dialogues. Studies could assess whether users fully understand the risks of granting storage access and the potential consequences of their choices. Insights from such research could guide improvements in the design of permission dialogues, ensuring they effectively communicate key privacy implications and foster informed decision-making by users.

Chapter 12

Conclusion

In our work, we investigated the prevalence and privacy implications of the Storage Access API on the web by conducting a large-scale empirical study. To gather real-world data, we developed a custom web crawler that utilized string matching and function hooking to identify API inclusions and calls. To ensure optimal data collection, we evaluated and compared two different crawling approaches, incorporating multiple browsers and simulating user interactions, to determine the most effective setup. For our primary crawl, we then deployed the crawler using the best-evaluated setup (single browser, no interactions) on 100,000 websites. On these sites, we visited up to ten pages per site for a potential total of 1,000,000 pages, providing us with a robust basis for analyzing SAA usage patterns. Additionally, we conducted the crawl five times over a period of around one month to capture temporal changes and trends in the API's adoption across the web.

Our results showed that the Storage Access API is already moderately used across the web, with API inclusions on more than one-fifth of all crawled sites. Among the observed calls and inclusions, the *hasStorageAccess* function was used far more frequently than *requestStorageAccess*, suggesting that developers predominantly rely on this function to check the storage access status rather than actively requesting access to unpartitioned cookies. Furthermore, the API landscape is highly influenced by a few unique scripts that are included repeatedly across a large number of different top-level sites, which indicates that the API is only used sparingly apart from the main use cases that these scripts provide. Our comparative analysis also revealed correlations between the usage of the API and the popularity of a top-level site, suggesting that more popular websites provide a wider variety of unique scripts and use of *requestStorageAccess*. Additionally, our comparisons regarding region and primary site content also showed significant differences in API usage, highlighting the diverse adoption patterns depending on geographic and contextual factors.

In terms of privacy risks, our analysis suggests that only a few framed SAA sites exhibit a high tracking risk. Furthermore, only one site (*google.com*) also demonstrated a high potential for cross-site tracking in using the API by exhibiting both a high tracking risk score and widespread framing across a significant number of top-level sites. Nevertheless, the moderately low percentage of top-level sites where *google.com* was framed also indicates only a marginal likelihood that a user will visit multiple of these websites and subsequently will fall victim to cross-site tracking. Furthermore, we observed no significant changes in privacy risks across all five of our crawls, suggesting consistency in the privacy risks. These findings collectively support the conclusion that, as of now, the Storage Access API does not present a significant threat to user privacy on the web.

Abbreviations

Acronym	Meaning
API	**A**pplication **P**rogramming **I**nterface
CAPTCHA	**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part
CORS	**C**ross-**O**rigin **R**esource **S**haring
CSRF	**C**ross-**S**ite **R**equest **F**orgery
eTLD+1	**e**ffective **T**op-**L**evel **D**omain plus one
ETP	**E**nhanced **T**racking **P**rotection
FedCM	**Fed**erated **C**redential **M**anagement API
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
hSA	**h**as**S**torage**A**ccess
ID	**Id**entity
IdP	**Id**entity **Provider**
IP	**I**nternet **P**rotocol
JSON	**J**ava**S**cript **O**bject **N**otation
RP	**R**elying **P**arty
RQ	**R**esearch **Q**uestion
rSA	**r**equest**S**torage**A**ccess
SAA	**S**torage **A**ccess **A**PI
StD	**St**andard **D**eviation
TL	**T**op-**L**evel
TLD	**T**op-**L**evel **D**omain
TRS	**T**racking **R**isk **S**core
URL	**U**niform **R**esource **L**ocator

Appendix A

Country Code

Country Code	Country
AR	Argentina
AU	Australia
BE	Belgium
BO	Bolivia
BR	Brazil
CA	Canada
CL	Chile
CO	Colombia
CR	Costa Rica
DE	Germany
DO	Dominican Republic
DZ	Algeria
EC	Ecuador
EG	Egypt
ES	Spain
FR	France
GT	Guatemala
HK	Hong Kong
ID	Indonesia
IN	India
IT	Italy
JP	Japan
KE	Kenya
KR	South Korea
MA	Morocco
MX	Mexico
NG	Nigeria
NL	Netherlands
NZ	New Zealand
PA	Panama
PE	Peru
PH	Philippines
PL	Poland
RU	Russia
TH	Thailand
TN	Tunisia
TR	Turkey
TW	Taiwan
UA	Ukraine
UK	United Kingdom
US	United States
UY	Uruguay
VE	Venezuela
VN	Vietnam
ZA	South Africa

Table A.1: List of country codes and corresponding countries used in regional analysis

List of Figures

List of Tables

Listings

Bibliography

[1] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 413–427, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6234427.

[2] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol, "Tracking the trackers," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 121–132, https://dl.acm.org/doi/abs/10.1145/2872427.2883028.

[3] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten, "Cookies that give you away: The surveillance implications of web tracking," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 289–299, https://dl.acm.org/doi/pdf/10.1145/2736277.2741679.

[4] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1388–1401, https://dl.acm.org/doi/pdf/10.1145/2976749.2978313.

[5] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, https://www.usenix.org/system/files/sec16_paper_lerner_1.pdf.

[6] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 155–168, https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final17.pdf.

[7] T.-C. Li, H. Hang, M. Faloutsos, and P. Efstathopoulos, "Trackadvisor: Taking back browsing privacy from third-party trackers," in *Proceedings of the 16th International Conference on Passive and Active Measurement*. Springer, 2015, pp. 277–289, https://link.springer.com/content/pdf/10.1007/978-3-319-15509-8.pdf.

[8] T. Bujlow, V. Carela-Español, J. Sole-Pareta, and P. Barlet-Ros, "A survey on web tracking: Mechanisms, implications, and defenses," in *Proceedings of the IEEE*, vol. 105, no. 8, 2017, pp. 1476–1510, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7872467.

[9] I. Fouad, N. Bielova, A. Legout, and N. Sarafijanovic-Djukic, "Missed by filter lists: Detecting unknown third-party trackers with invisible pixels," *arXiv preprint arXiv:1812.01514*, 2018, https://arxiv.org/pdf/1812.01514.

[10] P. Papadopoulos, N. Kourtellis, and E. Markatos, "Cookie synchronization: Everything you always wanted to know but were afraid to ask," in *The World Wide Web Conference*, 2019, pp. 1432–1442, https://dl.acm.org/doi/pdf/10.1145/3308558.3313542.

[11] M. Koop, E. Tews, and S. Katzenbeisser, "In-depth evaluation of redirect tracking and link usage," *Proceedings on Privacy Enhancing Technologies*, 2020, https://petsymposium.org/popets/2020/popets-2020-0079.pdf.

[12] S. Zimmeck, J. S. Li, H. Kim, S. M. Bellovin, and T. Jebara, "A privacy analysis of cross-device tracking," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1391–1408, https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-zimmeck.pdf.

[13] Google, "Storage partitioning," https://developers.google.com/privacy-sandbox/3pcd/storage-partitioning, accessed: 2024-07-10.

[14] Mozilla, "Introducing state partitioning," https://hacks.mozilla.org/2021/02/introducing-state-partitioning/, accessed: 2024-10-09.

[15] Apple, "Tracking prevention in webkit," https://webkit.org/tracking-prevention/, accessed: 2024-10-08.

[16] Brave, "Partitioning network-state for privacy," https://brave.com/privacy-updates/14-partitioning-network-state/, accessed: 2024-10-08.

[17] J. Jueckstock, P. Snyder, S. Sarker, A. Kapravelos, and B. Livshits, "Measuring the privacy vs. compatibility trade-off in preventing third-party stateful tracking," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 710–720, https://brave.com/research/files/storage-policies-www-2022.pdf.

[18] M. Smith, P. Snyder, M. Haller, B. Livshits, D. Stefan, and H. Haddadi, "Blocked or broken? automatically detecting when privacy interventions break websites," *arXiv preprint arXiv:2203.03528*, 2022, https://brave.com/research/files/PETS-2022-Blocked-or-Broken.pdf.

[19] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 179–194, https://dl.acm.org/doi/pdf/10.1145/3133956.3133966.

[20] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1143–1161, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9519502.

[21] W3C-PrivacyCG, "The storage access api," https://privacycg.github.io/storage-access/, accessed: 2024-10-07.

[22] ——, "The storage access api," https://github.com/privacycg/storage-access/, accessed: 2024-10-07.

[23] Google, "Storage access api," https://developers.google.com/privacy-sandbox/3pcd/storage-access-api, accessed: 2024-10-08.

[24] Apple, "Introducing storage access api," https://webkit.org/blog/8124/introducing-storage-access-api/, accessed: 2024-07-10.

[25] Mozilla, "Enable the storage access api on desktop," https://bugzilla.mozilla.org/show_bug.cgi?id=1513021, accessed: 2024-07-10.

[26] Brave, ""google sign-in" permission," https://brave.com/privacy-updates/24-google-sign-in-permission/, accessed: 2024-07-10.

[27] J. H. Artur Janc, "Improving the storage access api security model," https://docs.google.com/document/d/

1AsrETl-7XvnZNbG81Zy9BcZfKbqACQYBSrjM3VsIpjY/edit, accessed: 2024-10-07.

[28] M. L. Jones, "Cookies: a legacy of controversy," *Internet Histories*, vol. 4, no. 1, pp. 87–104, 2020, https://www.tandfonline.com/doi/full/10.1080/24701475.2020.1725852.

[29] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan, "An empirical study of web cookies," in *Proceedings of the 25th international conference on world wide web*, 2016, pp. 891–901, https://dl.acm.org/doi/pdf/10.1145/2872427.2882991.

[30] Network Working Group, "Http state management mechanism," *RFC 2109*, 1997, https://datatracker.ietf.org/doc/html/rfc2109.

[31] ——, "Http state management mechanism," *RFC 2965*, 2000, https://datatracker.ietf.org/doc/html/rfc2965.

[32] Internet Engineering Task Force, "Http state management mechanism," *RFC 6265*, 2011, https://datatracker.ietf.org/doc/html/rfc6265.

[33] ——, "Http state management mechanism," *RFC 6265bis*, 2017, https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-00.

[34] Google, "Delete, allow and manage cookies in chrome," https://support.google.com/chrome/answer/95647?hl=en&co=GENIE.Platform%3DDesktop, accessed: 2024-10-04.

[35] Mozilla, "Third-party cookies and firefox tracking protection," https://support.mozilla.org/en-US/kb/third-party-cookies-firefox-tracking-protection, accessed: 2024-09-30.

[36] Eurpean Union, "Regulation (eu) 2016/679 of the european parliament and of the council," https://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016, accessed: 2024-11-05.

[37] State of California, "California consumer privacy act (ccpa)," https://oag.ca.gov/privacy/ccpa, 2018, accessed: 2024-11-05.

[38] Cookiepedia, "How we classify cookies," https://cookiepedia.co.uk/classify-cookies, accessed: 2024-10-31.

[39] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 674–689, https://dl.acm.org/doi/pdf/10.1145/2660267.2660347.

[40] C. Castelluccia, L. Olejnik, and T. Minh-Dung, "Selling off privacy at auction," in *Network and Distributed System Security Symposium (NDSS)*, 2014, https://inria.hal.science/hal-00915249v1/document.

[41] A. Janc, "Partitioning designs for http cache & network state," https://arturjanc.com/partitioning-designs.pdf, accessed: 2024-10-19.

[42] Google, "Cookies having independent partitioned state (chips)," https://developers.google.com/privacy-sandbox/cookies/chips, accessed: 2024-10-08.

[43] J. Rautenstrauch, G. Pellegrino, and B. Stock, "The leaky web: Automated discovery of cross-site information leaks in browsers and the web," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2744–2760, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10179311.

[44] L. Knittel, C. Mainka, M. Niemietz, D. T. Noß, and J. Schwenk, "Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1771–1788, https://dl.acm.org/doi/pdf/10.1145/3460120.3484739.

[45] A. Janc, "Web platform improvements: Partitioning and cookies," https://arturjanc.com/xsleaks-2023-partitioning-and-cookies.pdf, accessed: 2024-10-09.

[46] Mozilla, "Firefox 86 introduces total cookie protection," https://blog.mozilla.org/security/2021/02/23/total-cookie-protection/, accessed: 2024-04-23.

[47] ——, "Firefox 89 blocks cross-site cookie tracking by default in private browsing," https://blog.mozilla.org/security/2021/06/01/total-cookie-protection-in-private-browsing/, accessed: 2024-04-23.

[48] ——, "New year, new privacy protection for firefox focus on android," https://blog.mozilla.org/en/mozilla/new-privacy-protection-for-firefox-focus-on-android/, accessed: 2024-04-23.

[49] ——, "Firefox rolls out total cookie protection by default to more users worldwide," https://blog.mozilla.org/en/mozilla/firefox-rolls-out-total-cookie-protection-by-default-to-all-users-worldwide/, accessed: 2024-04-23.

[50] ——, "Introducing state partitioning," https://hacks.mozilla.org/2021/02/introducing-state-partitioning/, accessed: 2024-09-30.

[51] Google, "An updated timeline for privacy sandbox milestones," https://blog.google/products/chrome/updated-timeline-privacy-sandbox-milestones/, accessed: 2024-10-21.

[52] ——, "Frequently asked questions related to third-party cookie deprecation in chrome," https://support.google.com/displayvideo/answer/14762010, accessed: 2024-10-08.

[53] ——, "Updates bei temporären ausnahmen für drittanbieter-cookies in chrome," https://developers.google.com/privacy-sandbox/blog/3pc-exceptions-update, accessed: 2024-10-08.

[54] UK Competition and M. Authority, "Investigation into google's 'privacy sandbox' browser changes," https://www.gov.uk/cma-cases/investigation-into-googles-privacy-sandbox-browser-changes, accessed: 2024-10-08.

[55] Brave, "Ephemeral third-party site storage," https://brave.com/privacy-updates/7-ephemeral-storage/, accessed: 2024-10-08.

[56] Apple, "Full third-party cookie blocking and more," https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/, accessed: 2024-05-15.

[57] ——, "Preventing tracking prevention tracking," https://webkit.org/blog/9661/preventing-tracking-prevention-tracking/, accessed: 2024-05-15.

[58] W3C, "Permissions," https://www.w3.org/TR/permissions/, accessed: 2024-10-09.

[59] ——, "Permissions policy," https://www.w3.org/TR/permissions-policy/, accessed: 2024-10-09.

[60] Apple, "Updates to the storage access api," https://webkit.org/blog/11545/updates-to-the-storage-access-api/, accessed: 2024-05-29.

[61] A. Janc, “Improving the storage access api security model,” https://github.com/privacycg/storage-access/issues/113, accessed: 2024-10-07.

[62] J. Hofmann, “Consider restricting storage access scope back to per-frame,” https://github.com/privacycg/storage-access/issues/122, accessed: 2024-10-07.

[63] C. Fredrickson, “Modify storage access to use a "per-frame" model,” https://github.com/privacycg/storage-access/pull/141, accessed: 2024-10-07.

[64] Mozilla, “Improving the storage access api in firefox,” https://hacks.mozilla.org/2022/02/improving-the-storage-access-api-in-firefox/, accessed: 2024-10-10.

[65] Google, “Related website sets,” https://developers.google.com/privacy-sandbox/cookies/related-website-sets, accessed: 2024-10-10.

[66] ——, “Related website sets,” https://github.com/GoogleChrome/related-website-sets, accessed: 2024-10-10.

[67] ——, “Get ready for new samesite=none; secure cookie settings,” https://developers.google.com/search/blog/2020/01/get-ready-for-new-samesitenone-secure, accessed: 2024-10-10.

[68] W3C-PrivacyCG, “requeststorageaccessfor explainer,” https://github.com/privacycg/requestStorageAccessFor, accessed: 2024-10-21.

[69] ——, “Extending storage access api (saa) to non-cookie storage,” https://privacycg.github.io/saa-non-cookie-storage/, accessed: 2024-10-22.

[70] Mozilla Developer Network, “Using the storage access api,” https://developer.mozilla.org/en-US/docs/Web/API/Storage_Access_API/Using, accessed: 2024-10-22.

[71] Google, “Federated credential management api overview,” https://developers.google.com/privacy-sandbox/cookies/fedcm, accessed: 2024-10-22.

[72] ——, “Temporary third-party cookie access using heuristics based exceptions,” https://developers.google.com/privacy-sandbox/cookies/temporary-exceptions/heuristics-based-exceptions, accessed: 2024-10-09.

[73] Apple, “Intelligent tracking prevention 2.0,” https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/, accessed: 2024-10-09.

[74] A. Maliev, B. Kelly, and H. Cho, "Cookie access heuristics explainer," https://github.com/amaliev/3pcd-exemption-heuristics/blob/main/explainer.md, accessed: 2024-11-02.

[75] Z. Yang and C. Yue, "A comparative measurement study of web tracking on mobile and desktop environments," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, 2020, https://par.nsf.gov/servlets/purl/10175641.

[76] R. Binns, U. Lyngs, M. Van Kleek, J. Zhao, T. Libert, and N. Shadbolt, "Third party tracking in the mobile ecosystem," in *Proceedings of the 10th ACM Conference on Web Science*, 2018, pp. 23–31.

[77] A. Das, N. Borisov, and M. Caesar, "Tracking mobile web users through motion sensors: Attacks and defenses." in *NDSS*, 2016, https://www.ndss-symposium.org/wp-content/uploads/2017/09/tracking-mobile-web-users-through-motion-sensors-attacks-defenses.pdf.

[78] P. Papadopoulos, N. Kourtellis, and E. P. Markatos, "The cost of digital advertisement: Comparing user and advertiser views," in *Proceedings of the 2018 World Wide Web Conference*, 2018, https://dl.acm.org/doi/abs/10.1145/3178876.3186060.

[79] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 541–555, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6547132.

[80] R. Upathilake, Y. Li, and A. Matrawy, "A classification of web browser fingerprinting techniques," in *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, 2015, https://ieeexplore.ieee.org/abstract/document/7266460.

[81] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Transactions on the Web (TWEB)*, vol. 14, no. 2, pp. 1–33, 2020, https://dl.acm.org/doi/pdf/10.1145/3386040.

[82] X. Pan, Y. Cao, and Y. Chen, "I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*

(NDSS), San Diego, CA, 2015, https://users.cs.northwestern.edu/~ychen/Papers/trackingfree_NDSS15.pdf.

[83] M. Kretschmer, J. Pennekamp, and K. Wehrle, "Cookie banners and privacy policies: Measuring the impact of the gdpr on the web," *ACM Transactions on the Web (TWEB)*, vol. 15, no. 4, pp. 1–42, 2021, https://dl.acm.org/doi/pdf/10.1145/3466722.

[84] M. Degeling, C. Utz, C. Lentzsch, H. Hosseini, F. Schaub, and T. Holz, "We value your privacy... now take some cookies: Measuring the gdpr's impact on web privacy," *arXiv preprint arXiv:1808.05096*, 2018, https://arxiv.org/pdf/1808.05096.

[85] K. Kollnig, R. Binns, M. Van Kleek, U. Lyngs, J. Zhao, C. Tinsman, and N. Shadbolt, "Before and after gdpr: tracking in mobile apps," *arXiv preprint arXiv:2112.11117*, 2021, https://arxiv.org/pdf/2112.11117.

[86] S. McQuistin, P. Snyder, H. Haddadi, and G. Tyson, "A first look at related website sets," *arXiv preprint arXiv:2408.07495*, 2024, https://dl.acm.org/doi/pdf/10.1145/3646547.3689026.

[87] N. Demir, M. Große-Kampmann, T. Urban, C. Wressnegger, T. Holz, and N. Pohlmann, "Reproducibility and replicability of web measurement studies," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 533–544, https://dl.acm.org/doi/pdf/10.1145/3485447.3512214.

[88] J. Jueckstock, S. Sarker, P. Snyder, A. Beggs, P. Papadopoulos, M. Varvello, B. Livshits, and A. Kapravelos, "Towards realistic and reproducibleweb crawl measurements," in *Proceedings of the Web Conference 2021*, 2021, pp. 80–91, https://dl.acm.org/doi/pdf/10.1145/3442381.3450050.

[89] J. Jueckstock, S. Sarker, P. Snyder, P. Papadopoulos, M. Varvello, B. Livshits, and A. Kapravelos, "The blind men and the internet: Multi-vantage point web measurements," *arXiv preprint arXiv:1905.08767*, 2019, https://arxiv.org/pdf/1905.08767.

[90] S. S. Ahmad, M. D. Dar, M. F. Zaffar, N. Vallina-Rodriguez, and R. Nithyanand, "Apophanies or epiphanies? how crawlers impact our understanding of the web," in *Proceedings of The Web Conference 2020*, 2020, pp. 271–280, https://dl.acm.org/doi/pdf/10.1145/3366423.3380113.

[91] W. Aqeel, B. Chandrasekaran, A. Feldmann, and B. M. Maggs, "On landing and internal web pages: The strange case of jekyll and hyde in web performance measurement," in *Proceedings of the ACM Internet Measurement Conference*, 2020, pp. 680–695, https://dl.acm.org/doi/pdf/10.1145/3419394.3423626.

[92] T. Urban, M. Degeling, T. Holz, and N. Pohlmann, "Beyond the front page: Measuring third party dynamics in the field," in *Proceedings of the web conference 2020*, 2020, pp. 1275–1286, https://dl.acm.org/doi/pdf/10.1145/3366423.3380203.

[93] F. Hantke, S. Calzavara, M. Wilhelm, A. Rabitti, and B. Stock, "You call this archaeology? evaluating web archives for reproducible web security measurements," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3168–3182, https://dl.acm.org/doi/pdf/10.1145/3576915.3616688.

[94] S. M. Mirtaheri, M. E. Dinçktürk, S. Hooshmand, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut, "A brief history of web crawlers," *arXiv preprint arXiv:1405.0749*, 2014, https://arxiv.org/pdf/1405.0749.

[95] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE symposium on security and privacy*. IEEE, 2010, pp. 332–345, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5504795.

[96] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2010, pp. 111–131, https://link.springer.com/content/pdf/10.1007/978-3-642-14215-4.pdf.

[97] J. Rautenstrauch, M. Mitkov, T. Helbrecht, L. Hetterich, and B. Stock, "To auth or not to auth? a comparative analysis of the pre-and post-login security landscape," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1500–1516, https://swag.cispa.saarland/papers/rautenstrauch2024auth.pdf.

[98] Microsoft, "Page | playwright python," https://playwright.dev/python/docs/api/class-page, accessed: 2024-10-27.

[99] N. Demir, J. Hörnemann, M. Große-Kampmann, T. Urban, N. Pohlmann, T. Holz, and C. Wressnegger, "On the similarity of web measurements under different experimental setups," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023, pp. 356–369, https://dl.acm.org/doi/pdf/10.1145/3618257.3624795.

[100] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," *arXiv preprint arXiv:1806.01156*, 2018, https://arxiv.org/abs/1806.01156.

[101] Y. Beugin and P. McDaniel, "A public and reproducible assessment of the topics api on real data," *arXiv preprint arXiv:2403.19577*, 2024, https://arxiv.org/pdf/2403.19577.

[102] K. Ruth, A. Fass, J. Azose, M. Pearson, E. Thomas, C. Sadowski, and Z. Durumeric, "A world wide view of browsing the world wide web," in *Proceedings of the 22nd ACM Internet Measurement Conference*, 2022, pp. 317–336, https://dl.acm.org/doi/pdf/10.1145/3517745.3561418.