Fine-Grained Data-Centric Content Protection Policy for Web Applications

Zilun Wang
The Chinese University of Hong Kong
Hong Kong SAR, China
zlwang22@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong SAR, China
lyu@cse.cuhk.edu.hk

ABSTRACT

The vast amount of sensitive data in modern web applications has become a prime target for cyberattacks. Existing browser security policies disallow the execution of unknown scripts but do not restrict access to sensitive web content by "trusted" third-party scripts. Prior works have observed that over-privileged third-party scripts can compromise the confidentiality and integrity of sensitive user data in the applications, which introduces vital security issues to web applications.

This paper proposes Content Protection Policy (CPP), a new web security mechanism for providing fine-grained confidentiality and integrity protection for sensitive client-side user data. It enables object-level protection instead of page-level protection by taking a data-centric design approach. A policy specifies the access permission of each script on individual sensitive elements. Any unauthorized access is denied by default to achieve the least privilege in the browser.

We implemented a prototype system—DOMinator—to enforce the content protection policies in the browser, and an extension—policy generator—to help web developers write basic policy rules. We thoroughly evaluated it with popular websites and showed that it could effectively protect sensitive web content with a low performance overhead and great usability. CPP complements existing security mechanisms and provides web developers with a more flexible way to protect sensitive data, which can further mitigate the impact of content injection attacks and significantly improve the security of web applications.

CCS CONCEPTS

• **Security and privacy → Browser security, Web Application Security**.

KEYWORDS

Browser security; JavaScript; Security policy

ACM Reference Format:
Zilun Wang, Wei Meng, and Michael R. Lyu. 2023. Fine-Grained Data-Centric Content Protection Policy for Web Applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623217

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623217

1 INTRODUCTION

With the rapid development of the Internet, web applications have influenced many aspects of our lives. A lot of sensitive information, *e.g.*, personally identifiable information (PII), authentication tokens, and online transaction data are presented in modern web applications that we use on a daily basis. The security—particularly the confidentiality and integrity—of critical data and web applications has become extremely important to not only website administrators but also end users. Traditionally, attackers launch code injection attacks, *e.g.*, Cross-Site Scripting (XSS), to gain control over such data and the victim websites. To mitigate content injection attacks, modern browsers all have implemented the Content Security Policy (CSP) to control what scripts can be executed in a document. However, CSP supports all-or-nothing restriction, *i.e.*, a remote script is allowed either full or no privilege. In other words, the "trusted" scripts can abuse their privilege to completely access any data, including sensitive user data.

While providing convenience to web developers and offering rich features to the end users, the over-privileged third-party JavaScript code also introduces vital security and privacy issues to websites and end users. In practice, web developers often have to trade security for the functionalities provided by trusted scripts. However, the over-privileged, trusted scripts can potentially compromise the security of sensitive client content. In fact, some included *trusted* scripts themselves are not trustworthy and have been found to perform malicious activities in the past. For instance, prior works have revealed that the over-privileged scripts would exfiltrate sensitive user data such as email address, password, credit card information, and health information and leak it to external parties [3, 23]. Researchers have also discovered that the over-privileged third-party scripts would modify website content to intercept user clicks for performing ad click fraud and distributing malicious content [32]. Past research has demonstrated the severity of the third-party script privilege abuse problem, yet billions of web users' sensitive data is still at risk due to the all-or-nothing permission model in today's browsers.

The research community has proposed many web security mechanisms to restrict the privilege of third-party scripts and protect sensitive user data on the client side. These mechanisms [2, 11, 17, 18, 28] are usually functionality-centric rather than data-centric, *i.e.*, they limit the available functionalities of third-party JavaScript code rather than protecting the sensitive data from malicious code directly. For instance, some mechanisms [2, 11, 17] block the use of sensitive APIs such as `eval()`. They, however, cannot stop the above

data confidentiality and integrity attacks because the included scripts are still granted access to the entire DOM, which is needed for providing many legitimate functionalities such as content customization, data validation, *etc.* Some works [6, 10, 12, 13, 21, 25] use information flow control techniques to provide confidentiality protection. Nevertheless, these works inevitably introduce incredibly huge performance overhead. More importantly, few prior works consider the content integrity problem and would still make integrity attacks like click interception feasible. We argue that a fine-grained, data-centric, and efficient protection mechanism is needed to defeat the data confidentiality and integrity attacks by over-privileged JavaScript code. The prior work ScriptInspector [33] logs script operations and generates policies for specifying what resources can be accessed. However, it performs an offline analysis method and cannot enforce any access restriction at runtime. Further, it has some limitations in usability and high runtime overhead, which are discussed in §2.2. Nevertheless, it has made a big step towards providing fine-grained content protection in the browser.

In this paper, we propose Content Protection Policy (CPP), a new data-centric web security mechanism to provide *fine-grained* and *data-centric* content protection on the client side. We also develop a browser-based system—DOMinator—to support and enforce the protection rules, along with a policy generator to help developers define CPP policies. With CPP, web developers can easily define policy rules for specific (sensitive) elements to provide flexible access control over third-party scripts. Different from the browsers' current all-or-nothing restriction, the fine-grained access control decision in CPP is made for each DOM element and each accessing script. CPP follows the fail-safe default and least privilege security principles: only the explicitly granted scripts in the policy can be allowed access to the protected content.

To ensure complete mediation, DOMinator enforces the access control at the JavaScript and DOM binding layer in the browser, such that all DOM accesses are checked for permission. Furthermore, the CPP policies are immutable to any JavaScript code, effectively preventing any circumvention attempts via tampering with the policies. Our in-browser access control design and implementation prevent any script from bypassing the permission check. To ease the burden of writing detailed content protection policies, the policy generator in DOMinator allows the developers to easily label the sensitive elements through a GUI tool and automatically generate policy rules based on the collected access traces. To limit the runtime overhead and compatibility issues, DOMinator enforces access control for only the sensitive elements specified by the developers in the policy.

We implemented a prototype of DOMinator based on the Chromium browser with about 1,700 lines of C++ code. We systematically evaluated the effectiveness, performance, and usability of DOMinator. By implementing and deploying various confidentiality and integrity attacks via different JavaScript injection methods, we validated the effectiveness of DOMinator in protecting sensitive web content from over-privileged third-party scripts. We also evaluated DOMinator on 20 real-world websites reported in [23] with email and password exfiltration behaviors and demonstrated that it could effectively prevent sensitive data exfiltration to third parties with simple policies. We evaluated the performance and the usability of our prototype on the Tranco top 50 accessible websites. Compared to the vanilla browser, enforcing CPP in the prototype of DOMinator introduced a 0.03% average overhead in memory consumption and a 1.53% average slowdown in page loading. The performance overhead is unnoticeable for daily uses and acceptable given the added security benefits. In usability evaluation, we evaluated the compatibility and maintenance cost of DOMinator. We found that the enforcement of CPP in DOMinator did not cause new errors on the popular websites, and over 86% of the websites did not require any policy updates in a week. We also observed that the policies generated for sub-pages of the same website were mostly similar due to the use of common content layouts and templates. This indicates that the policy maintenance cost is reasonable, even considering the numerous sub-pages and frequent updates on popular websites. Our evaluation shows that DOMinator can effectively protect web content with an unnoticeable performance overhead and great usability.

In summary, we make the following contributions:

- We propose a new web security mechanism CPP to provide fine-grained confidentiality and integrity protection for client-side sensitive data.
- We implement CPP in DOMinator based on the Chromium browser to ensure complete mediation and the least privilege on third-party JavaScript code.
- We develop an automated policy generation tool to help web developers write proper protection policies.
- We demonstrate with popular real-world websites that DOMinator can effectively and efficiently prevent over-privileged content access with good usability.
- We publicly release our research artifact to facilitate future research at https://github.com/cuhk-seclab/DOMinator.

2 PROBLEM STATEMENT

2.1 Research Problem

The existing client-side security mechanisms deployed in modern browsers grant full frame access to any included remote scripts and block accesses from scripts in different origins. Although such all-or-nothing protection provides effective defense against potential attacks from untrusted origins, it is still limited due to the little or even no restriction on the included third-party scripts. The third-party scripts would have the same privileges as the first-party ones, *i.e.*, they can access any resources in the context.

The unlimited high privilege could be abused by the included third-party scripts, which can compromise the confidentiality and integrity of sensitive first-party data. For instance, the over-privileged third-party scripts have the ability to access or even exfiltrate sensitive user information (*e.g.*, PII) presented on a page; they can also alter the content displayed on the page to deliver false information or change the application's behaviors. Previous studies have demonstrated that over-privileged third-party scripts have exfiltrated PII [3, 23] and modified page hyperlinks to intercept clicks [32].

2.1.1 Motivating Examples. We use the following two examples to demonstrate the third-party script privilege abuse problem. They

```
// The first-party HTML code
<html>
  <head>
    <script src="https://analytics.com/a.js"></script>
  </head>
  <body>
    <input name="username" type="email" class="auth">
    <input name="password" type="password" class="auth">
  </body>
</html>

/* a.js */
var e = document.getElementsByTagName("input");
var values = new Array(e.length);
for (var i = 0; i < e.length; i++) {
  e[i].onkeypress = function (k) {
    values[i] += k.key;
  };
}
```

Listing 1: An example of the attack on content confidentiality.

```
// A third-party script modifies hyperlinks
var b = document.getElementsByTagName("a");
for (var l = 0; l < b.length; l++) {
  var h = b[l].href;
  var p = "..." + encodeURIComponent(h) + "...";
  b[l].href = p;
  if (b[l].id == "target") {
    b[l].href = "https://attack.com";
  }
}
```

Listing 2: An example of the attack on content integrity.

are simplified from known attacks studied in [3, 32]. They respectively show the threats posed by third-party scripts to the confidentiality and integrity of sensitive client-side content.

Attack on Confidentiality. Some third-party scripts were reported to collect sensitive information without user consent [3, 9]. Listing 1 shows an example of a script that exfiltrates the password from the user input. It registers an event handler to monitor the keystroke pressed in the input elements and stores the input sequence in a string for further processing in a remote server. However, it does not exclude sensitive information (*e.g.*, password), which is at the risk of being exfiltrated. The example indicates that the first party cannot totally trust the third-party scripts and should actively protect their sensitive data. A research [23] found password collection on 52 websites out of the top 100K websites by third-party scripts, 7 of which are in the top 20K. It also highlights that thousands of websites have email leaks, which is a huge threat to user privacy. Although password exfiltration is less common, its serious consequences and difficulty in detection make it impossible to ignore.

Attack on Integrity. Some third-party scripts were found to modify the website contents to intercept user clicks for malicious purposes [32]. Listing 2 shows an example of a third-party script that modifies a hyperlink to direct victim users' clicks to a malicious URL. The script modifies the hyperlink of the anchor element with the id "target", which can be some key component in the website, to navigate the users to https://attack.com. A prior work [32] observed that 437 third-party scripts intercepted user clicks on 613 websites out of the top 250K websites and found malicious cases in the real world that direct users to fake anti-virus (AV) software and drive-by download pages.

These examples show that it is vital to protect both the confidentiality and integrity of client-side contents from over-privileged third-party scripts, and the protection should be fine-grained to preserve the legitimate functionality of the third-party scripts. In practice, the attackers may obfuscate the malicious code and perform the behavior occasionally to prevent inspections, which makes detection and defense even harder.

2.1.2 Threat Model. In our research, we assume an over-privileged third-party script developer as the attacker who aims to exfiltrate sensitive information and/or modify the content in a client-side web application. In other words, the attacker aims to compromise the confidentiality and/or integrity of a target client-side web application. Some traditional code injection attacks, *e.g.*, XSS, can largely be mitigated by defense mechanisms like CSP, which we assume have been correctly implemented and deployed. Thus, we do not consider code injection attacks. Instead, we assume the attacker's scripts are included, either directly or indirectly, by the target website. The attacker might act as a benign tracking service provider but exfiltrate ungranted sensitive information on the website, which is difficult to inspect. He might also hack the server of a content provider included by the target website and inject malicious code into the content provider's script. Consequently, he can gain direct access to the data of interest by abusing the full privilege in the target website frame. Enforcing CSP is not sufficient for defending against such attacks. We also assume that attackers would not attempt to steal sensitive data through any unknown side channel.

2.2 Existing Solutions

The root cause of the above privilege abuse problems is the lack of privilege restriction on third-party scripts. The security community has proposed many approaches to limiting third-party scripts' privilege. However, while these techniques offer a certain level of restriction, they do not make proper trade-offs between security, compatibility, performance, and usability, and thus are not adoptable in practice [24].

JavaScript Restriction. Many prior works [4, 11, 22, 26–28] try to confine JavaScript in an isolated environment to restrict their functionality in different ways. The scripts are isolated from the data, including the one that is necessary to their legitimate functionalities, to prevent direct manipulation, which can incur compatibility issues. Some other works restrict the untrusted scripts by statically rewriting or wrapping third-party JavaScript code [1, 2, 17, 18, 20, 29]. They generally restrict some features of JavaScript code to prevent certain malicious operations from untrusted third-party scripts. Still, they do not limit access to sensitive web content and cannot solve the privilege abuse problem fundamentally. Further, code rewriting can break legitimate functionalities and does not apply to many scripts that are dynamically included, and thus, it can hardly be deployed.

Information Flow Control. Many other mechanisms use information flow control (IFC) [6, 10, 12, 13, 21, 25] to prevent data exfiltration in a website. Although these methods can defend against data exfiltration, most of them cannot provide integrity protection to web applications. Moreover, since IFC techniques need to track the complete data flow of each piece of sensitive data, they inevitably introduce high runtime overhead (ranging from 50% to 250%), and thus have poor usability for the end users.

Script Monitoring. A prior work, ScriptInspector [33], logs script operations and defines policies for specifying what resources can be accessed. It uses an offline analysis method and does not enforce any restriction to scripts at runtime. In fact, the security policies the authors design cannot be widely utilized due to two critical limitations. First, their policy is designed for individual scripts, which means web developers need to specify security policies for every third-party script that might be included. Due to the large number of third-party scripts and the dynamic inclusion of different scripts, it is impractical to write a complete policy for a real-world website. Second, this design would introduce high runtime overhead, which cannot be simply optimized and has been clearly discussed by themselves. To enforce the policies, they need to record every access and traverse the resources defined in the policy to determine whether the access should be allowed. In brief, this design lacks good usability and runtime performance.

In summary, there is a need for an effective security mechanism to restrict JavaScript privilege at fine granularity and protect both the confidentiality and integrity of the sensitive content in web applications. It also should preserve the compatibility and not introduce high performance overhead.

2.3 Research Objectives and Challenges

2.3.1 Research Objectives. As discussed above, none of the existing techniques can well address the third-party privilege abuse issue that compromises client-side content confidentiality and integrity. To fundamentally solve the JavaScript privilege abuse problem, a desired solution is to define and enforce a data-centric permission policy restricting access to individual elements. Our research objective is to develop a practical, fine-grained, and data-centric mechanism to protect both the integrity and the confidentiality of important client-side web content. We aim to provide access control at the object level in the browser to mitigate the third-party privilege abuse issues caused by the browsers' current all-or-nothing permission model.

2.3.2 Research Challenges. We meet the following research challenges in achieving our objectives.

Effectiveness. The protection mechanism needs to effectively detect and deny *all* unauthorized content access by JavaScript. In other words, it should ensure complete mediation and cannot be bypassed. This is difficult due to the various DOM operation APIs and the dynamic features in JavaScript. For example, to read a user's name presented in an HTML element, a third-party script can invoke APIs like `Element.innerHTML` and `Node.textContent` on the element or any of its ancestor elements. It can also dynamically inject inline anonymous JavaScript code to bypass defense. To provide access control to the website contents, we need to cover the DOM APIs of all HTML elements to ensure complete mediation and consider all dynamic JavaScript inclusion methods when attributing access to the scripts.

Usability. The protection mechanism should have good usability so that it can be largely deployed in browsers and on real-world websites to offer protection. This requires it to work well with a large variety of applications that are developed using many different programming languages, development frameworks, and technologies and are in different levels of complexity. Furthermore, as a new protection mechanism in the browser, it should not break the existing browser features and the existing security mechanisms.

Performance. The protection mechanism should not negatively affect the performance of the billions of websites on the Internet, as the performance is critical to their service quality and income. It should incur a runtime overhead to a level that is mostly unnoticeable for daily uses. Otherwise, the developers might not choose to use the mechanism at all. Performance overhead is also a concern of browser vendors. Indeed, many security mechanisms are not used mainly because of their high performance overhead [6, 10, 12, 13, 21, 25].

3 CONTENT PROTECTION POLICY

In this section, we present our solution, Content Protection Policy (CPP), to address the privilege abuse problems. CPP is a new data-centric web security mechanism that restricts third-party scripts' privilege in the browser. The key advantage of CPP is that it preserves the functionality of the third-party scripts in the whole context. We noticed that the root cause of compatibility issues in existing techniques (§2.2) is that they try to solve the privilege abuse problems from a JavaScript perspective by confining some features or APIs. In our threat model, the essence of scripts' privilege is their access right to first-party resources. CPP avoids compatibility issues by providing protection to individual resources. This data-centric design lets developers focus on specifying which elements are sensitive without concern about the detailed behaviors of each third-party script. Moreover, CPP is designed as a flexible policy to provide *object-level* protection.

In the following, we introduce the policy syntax (§3.1), then describe our design of CPP in detail (§3.2).

3.1 Syntax

In our design, a CPP rule defines the *resource* of protection, *principal* that access the resources, and the *access right* of the principals. To provide good usability for web developers, CPP uses a syntax similar to CSS, which is already familiar to developers. The grammar of CPP is presented in the following.

$$
\begin{array}{lcl}
< PolicyRule > & ::= & < Resource >^{*} < Declaration >^{*} \\
< Resource > & ::= & < NodeSelector > \mid < JSAPI > \\
< Declaration > & ::= & < Principal >^{*} \ [< AccessRight >^{*}] \\
< Principal > & ::= & < URI > \mid < Origin > \mid < Domain > \\
< AccessRight > & ::= & \text{“}R\text{”} \mid \text{“}W\text{”} \mid \text{“}None\text{”}
\end{array}
$$

We introduce the three components of a CPP rule — *resource*, *principal*, and *access right* — as follows.

Resource. A web page consists of various resources in terms of sensitivity, *e.g.*, a password input field or a sensitive JavaScript API. As we discussed in §2.1, existing mechanisms allow third-party scripts to either access all resources or none at all, resulting in overly coarse-grained control. On the contrary, CPP allows web developers to define flexible access control rules for specific *resource*, which can be either a DOM node or a JavaScript API. The syntax of *NodeSelector* is the same as the CSS selector. That is, the developers can simply use id, class, or other CSS selectors to specify the content they want to protect, which provides flexible

protection for the HTML elements for the developers. For instance, they can use `input` to strictly protect all input elements or use `input[type="password"]` to protect password fields. In addition, *resource* can be *JSAPI* to restrict the invocation of sensitive JavaScript APIs, which is denoted by using `@Api` followed by the API name.

Principal. The *principal* in a CPP rule defines the third-party scripts to which the rule applies. It refers to the source of the JavaScript code, such as the URI, origin, or domain, from which the scripts are loaded. The *principal* can also accept wildcards to match multiple scripts. For instance, the principal matching `<script src="https://www.a.com/s.js">` could be https://www.a.com/s.js, https://*.a.com/ or www.a.com. Third-party scripts that match the principal are granted access to the corresponding resource. On the other hand, first-party scripts have full access rights to the resource, regardless of the principal. In our design, user scripts, such as browser plug-ins and extensions, will be treated as first-party scripts and have full rights. Although these scripts might also abuse their privilege, it is out of our research scope (discussed in §7).

Access Right. The *access right* in a policy rule specifies the type of access a principal is authorized to have to a resource. A developer can explicitly allow a principal to read or/and write the resource. By default, principals without explicitly granted access rights are prohibited from accessing the resource *i.e.*, with *access right* of "None", to protect its confidentiality and integrity. Specifically, an API policy (*i.e.*, a policy with a *JSAPI resource*) does not require the specification of *access right* because such a policy is used to restrict API invocation instead of access control.

We use an example to illustrate the syntax of CPP better. The highlighted code in Listing 3 is two CPP rules defined in an HTML page in an internal policy[1]. The first rule protects the elements with the "auth" class name. It denies any third-party script access except for those from "*.example.com". The second rule means only the third-party scripts from "*.example.com" can invoke `document.write()` API. The syntax of CPP is easy to understand and use, which does not introduce much learning cost to the web developers.

3.2 Policy Design

In our design, CPP policies are enforced on the client side to control the access of third-party scripts. To provide reliable security protection, CPP follows the fail-safe default and least privilege security principles. In other words, the mechanism allows only data accesses that are permitted based on security policies specified by the site administrator. Specifically, we define a *default policy*, which has an empty or "default" principal and a default ("None") access right. All accesses without explicit permission should be denied. Such a fail-safe default design choice effectively blocks any unauthorized access to resources, thus restricting the privileges of third-party scripts to their necessary minimum (least privilege).

Moreover, we also consider policy inheritance in our design. It is natural that the protection to a container tag should be applied to the whole subtree under it. That is, all the child nodes should be protected at least at the same level as the parent node. Therefore, if a DOM object has child objects, *i.e.*, it is the root node of a DOM sub-tree, and its policy rules should be automatically inherited by child objects that do not have one. Web developers need to define only one policy at the root node and can apply it to all objects in the tree.

[1]We will explain the three types of policies using this example in §3.2.

```
1  /* https://example.com */
2  <html >
3      <head>
4          /* External Policy */
5          <cpp src="auth.policy"/>
6
7          /* Internal Policy */
8          <cpp>
9              .auth {
10                 "*.example.com": "RW",
11             }
12             @Api document.write() {
13                 "*.example.com",
14             }
15         </cpp>
16
17     </head>
18     <body>
19         /* Inline Policy */
20         Username: <input type="email" class="auth" policy="'
       default': 'None'"/>
21         Password: <input type="password" class="auth" policy="'
       default': 'None'"/>
22     </body>
23 </html>
```

Listing 3: Policy Declaration Examples.

There are limited contents in web applications that are of great importance to end users or web administrators. Except for these contents, most of the other contents might be accessed by many third-party scripts for a legitimate purpose (*e.g.*, analytics and advertisement) and do not need strict protection. Therefore, if a DOM object does not have a specified or inherited policy, CPP treats it as insensitive and grants full access privilege ("RW") to all scripts. This design is backward-compatible and eliminates the work of writing CPP policies for resources that do not need protection.

CPP supports three different types of policies: inline, internal, and external. The inline policies can be defined via the new `policy` HTML element attribute of the corresponding elements. They are fairly simple and can be easily adopted on small websites. Inline policies might not be very suitable for complex web applications, whose web pages contain thousands of elements that are dynamically constructed from multiple templates and modules. Therefore, CPP also supports internal policies. Web developers can define policies using a new `<cpp>` element under the `<head>` element. CPP also supports external policies, which can be reused across multiple web pages and even included remotely via URL. Listing 3 shows the examples of defining policies in three ways. In the following, we mainly take the internal policy as an example for discussion unless otherwise stated.

4 DOMINATOR

In this section, we introduce DOMinator, which is a modified browser supporting the CPP mechanism, along with a policy generator extension helping developers write basic policies. In the following, we first present an overview of DOMinator (§4.1). Then, we describe how DOMinator supports (§4.2) and enforces (§4.3) the new security mechanism CPP. Finally, we describe how the policy generator extension helps web developers automatically generate preliminary policies for their sites (§4.4).

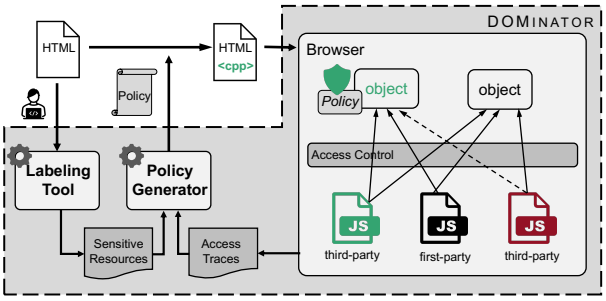

Figure 1: An architecture overview of DOMINATOR.

4.1 Overview

We implement DOMINATOR in the browser to support and enforce CPP at runtime. We opt for implementation in the browser because the browser can mediate all JavaScript accesses to the DOM, which is analogous to the operating system kernel that can mediate all system calls. Although it is technically possible to enforce the policies via other methods *e.g.*, browser extensions or JavaScript rewriting, they have non-negligible limitations. Browser extensions cannot easily mediate other scripts' access and can be easily circumvented. JavaScript rewriting requires web developers to rewrite every third-party script that might be included, which is impractical.

The architecture of DOMINATOR is depicted in Figure 1. DOMINATOR is composed of two parts: a modified web browser that integrates the CPP mechanism and an extension that helps web developers write policy rules. The browser can parse policy rules in an HTML document, associate those rules with protected objects, and enforce access control at runtime. To enforce access control, it mediates all scripts' DOM accesses and denies unauthorized content accesses and API calls according to the specified policy. Therefore, only the third-party scripts with explicitly granted permissions can access the protected resources, which effectively protect sensitive information. Meanwhile, unprotected resources can be normally accessed by all the included third-party scripts. Web developers can use the extension to write policy rules, which can significantly reduce the potential human effort. They can use the labeling tool to label the elements they want to protect. With the labeled elements, the policy generator utilizes the collected access logs to automatically generate and update a basic policy. Specifically, when updating the policy, it performs automated UI testing to increase JavaScript code coverage, which can effectively improve the compatibility of the finally generated policy.

4.2 Policy Support in DOMINATOR

In DOMINATOR, we integrate the CPP mechanism and support the policy features discussed in §3. When visiting a website with policies defined, DOMINATOR can correctly parse the policy rules and apply them to the specified resources. Specifically, when DOMINATOR parses a document, the pre-defined policies (including inline, internal, and external policies) are also parsed and then applied. All the policy rules applied to the same DOM object are managed in the access control list data structure, *PolicyData*. A *PolicyData* is a list of access control entries associated with the DOM object and its child objects (if any). Each entry defines the access right of a principal on the associated object. For the policy rule with a "default" principal or no specified principal, the corresponding DOM object is applied with a default entry. To ensure a fail-safe default and least privilege feature, we specify the default entry with the default access permission ("None") for all third-party scripts. Whereas for an API policy, DOMINATOR applies it in the whole context instead of some specific DOM object since it restricts JavaScript methods instead of protecting DOM objects. DOMINATOR stores the method name and the principal defined in the policy and associates it with the HTML Document. Since it is infeasible to support restriction to all JavaScript APIs for API policy due to limited manpower, DOMINATOR supports some representative sensitive interfaces, *e.g.*, `document.write()` and `eval()`.

Particularly, CPP policies are designed to be tamper-resistant in DOMINATOR. As a security mechanism protecting the integrity and confidentiality of sensitive resources, CPP should secure itself against potential tampering to provide reliable protection. Therefore, DOMINATOR has some designs to defend against potential attacks on our security mechanism. First, the PolicyData is implemented internally in the browser and not exposed to JavaScript API. So, it is impossible to manipulate the scripts' privilege on a DOM object by directly modifying its PolicyData via JavaScript API. Second, the pre-defined policy rules in a webpage are immutable to JavaScript. That is, attackers cannot modify the policy rules in any potential ways through JavaScript. Third, DOMINATOR only accepts CPP policies pre-defined in the HTML document. In other words, any other dynamically generated or injected policies would not be parsed and applied. These designs greatly promise the security of the CPP mechanism in practice and minimize its attack surface.

4.3 Policy Enforcement in DOMINATOR

To enforce the CPP policy when a specific script initiates a DOM access, in the corresponding DOM interface code, we check the policies of the target objects and determine if this operation by this script should be granted or not. Given JavaScript's dynamic nature and the complexity of web applications, enforcing the policy effectively and efficiently is not trivial. To achieve this, DOMINATOR needs to 1) accurately attribute access to the correct script and identify the script's principal and 2) completely mediate DOM APIs to monitor and control JavaScript access to the protected resources. The challenges of the two requirements are well discussed in §2.3.2. The following describes how our design in DOMINATOR addresses these challenges.

4.3.1 Identifying JavaScript Principal. To enforce policies and provide access control, DOMINATOR needs to identify the JavaScript principal of an access attempt. When a JavaScript interface requests access to a DOM object, DOMINATOR locates the script that initiates the access attempt and identifies its principal.

When attributing an access operation, it is not correct to straightforwardly trace the direct invocation of a function. Some scripts may make an indirect DOM access through invoking functions in other scripts, *e.g.*, the jQuery library can be invoked by another script to make the access. Following prior work [31], DOMINATOR inspects the JavaScript call stack to identify the bottom script when access is invoked. This bottom script is regarded as the script that initiates the access. DOMINATOR can potentially identify all the

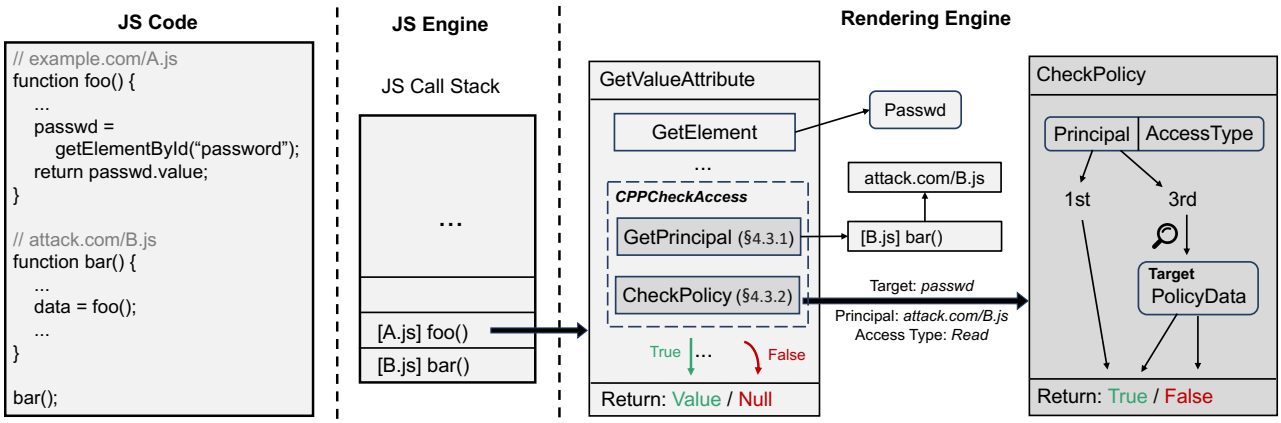

Figure 2: An example of policy enforcement.

current scripts on the call stack as the principal. However, this might greatly complicate the access control decision process, especially when scripts with conflicting permissions are involved—to ensure security, access should be granted when all scripts on the stack have access permission. Such a strict access model might potentially break a lot of real websites. We leave it to future work to study the feasibility of such a design.

When identifying its principal, we need to consider several situations. External scripts' principals can be simply identified via their source attribute. For inline scripts, it is difficult to identify their principals because they are anonymous to the browsers. We cannot simply regard them as first-party scripts because they might be dynamically injected by other third parties. Therefore, we monitor dynamic script inclusion for determining whether an inline script is dynamically injected and assigning it the proper principal (*i.e.*, the same as its including/parent script). There are multiple ways to include scripts dynamically: HTML script element, JavaScript URLs, JavaScript event listeners, and `eval()`. Prior works like JSIsolate [31] try to monitor the dynamically included scripts, but they are incomplete in covering all four methods. For the sake of completeness, we make some improvements based on the prior works and monitor all four dynamic inclusion methods. The included JavaScript code through `eval()` can be correctly identified by simply checking the call stack to find its caller (*i.e.*, the bottom script as we discussed above). For the other three methods, we identify and assign the including script's principal to the included inline script by monitoring the script inclusion process in the browser. The details of the implementation are introduced in §5.

Figure 2 shows an example to illustrate how DOMinator enforces the CPP policies in the browser. DOMinator identifies the script attack.com/B.js as the principal of the script initiating the access attempt to the `#password` element. It then checks the target element's policy to decide if this script is permitted to make the access. If B.js dynamically injects additional inline scripts, DOMinator would assign their parent script's principal (*i.e.*, B.js) to them.

4.3.2 Mediating DOM Access. DOMinator mediates JavaScript DOM interfaces for making access control decisions. Since JavaScript can access objects not only directly via DOM interfaces but also indirectly through some interfaces of their ancestor nodes, we consider both direct and indirect accesses. It should be noted that sensitive information may also be exfiltrated to a script even if it does not access the DOM. For example, the script with granted permissions can intentionally store sensitive data in a variable in the global scope to allow any other script to access it. This situation is out of our research scope. It could be addressed by IFC techniques, which provide complete information flow tracking in the JavaScript code. However, such a solution can significantly degrade the performance of the client-side code. JavaScript isolation mechanisms like JSIsolate [31] can also be deployed to block indirect sensitive data access via trusted global JavaScript objects. In the following, we first discuss how DOMinator mediates the direct and indirect access, and then introduce how DOMinator makes access control decisions at runtime (the `CheckPolicy` function shown in Figure 2).

Direct Access. There are hundreds of interfaces (including methods and properties) available in DOM. Intuitively, it is infeasible to cover all of them with limited effort. According to the HTML DOM API standard, despite most HTML objects having numerous properties, most do not have individual methods. We mediate all the methods that may read or write the contents of objects but filter out the irrelevant methods, *e.g.*, `Node.getRootNode()`. Moreover, the property interfaces are auto-generated by the code generator in the browser so that we can simply mediate all of them by simply modifying the templates. The details of the implementation are introduced in §5. Our mediation covers the most common operations in web applications, including but not limited to node insertion and removal, setting or getting attributes, and event handler registration, *etc.*

Indirect Access. Consider such an HTML code snippet: `<div><span>secret</span></div>`, where the `<span>` is protected from a third-party script, but the `<div>` is not. In this case, the third-party script can simply get the secret via `getInnerHTML()` interface of the `<div>` since it has full privilege to the unprotected node. Similar interfaces also include `Node.textContent`, `Element.innerText`, *etc.* When such properties or methods are invoked, DOMinator also checks the accessing script's permission on the accessed object and its child objects. If the principal is disallowed to access any child, DOMinator handles the access differently according to the access type. A Write operation would be entirely disallowed, and none

of the children's contents would be overwritten. While for a Read operation, the protected objects would be ignored, and only the contents of the other elements would be returned. This design takes the structure feature of web contents into consideration and is vital to ensure that the CPP mechanism is effective and non-bypassable.

Access Control Decision. When a script tries to access an object (either directly or indirectly) through a mediated interface, DOMinator checks its authorized permission on the object. DOMinator immediately grants access if the object is not associated with any PolicyData or the accessing script is first-party. Otherwise, DOMinator searches the PolicyData to find the entry whose principal best matches the accessing script ("default" matches all third-party scripts). DOMinator compares the requested access (Read and/or Write) of the invoked interface with the access right in the policy rule to determine if it should be allowed. If access should be denied, DOMinator returns `Null` for a Read operation returns and ignores a Write operation. In our design, the event registration interface requires both Read and Write permission, which may be counter-intuitive. This is because such an operation not only modifies the event property of the object but also might further obtain information through a registered event. The attack in Listing 1 is an example.

4.4 Policy Generation

The effectiveness of CPP on protecting web applications depends on the security policies defined for the applications. However, writing a policy at the object level may be non-trivial, especially for developers who are not security experts. On the one hand, they need to identify the sensitive elements and write the appropriate *resource* for them; On the other hand, they need to find all the third-party script access to the protected resources to write the policy rules. Considering that a website may have numerous contents and many scripts, it may be tedious to write a policy manually. Therefore, it is important to limit the developer's efforts in order to deploy a new security mechanism in practice widely. Thus, we develop an extension in DOMinator to help web developers generate a basic policy for their applications.

To create policy rules for a website, developers first need to specify the elements that need to be protected. However, finding these sensitive elements through the webpage source code can be challenging. The policy generator provides a labeling tool that reduces the manual effort in identifying sensitive elements. The labeling tool is a browser extension that enables developers to simply hover their mouse over an element to label it as sensitive. The tool then computes an appropriate and unique selector for each labeled element. In addition, the extension accepts pre-defined rules to label corresponding elements automatically. For instance, developers can set the rule `input` to label and highlight all input elements. The labeled elements are then highlighted, allowing developers to make further adjustments. Once labeling is complete, the extension logs all generated selectors as the *sensitive resource list*, which is used for policy generation.

Given the sensitive resource list, the policy generator can automatically generate and update a basic policy. The policy generation starts with an initial policy that defines strict policy rules to deny all access to protected resources. With the initial policy applied,

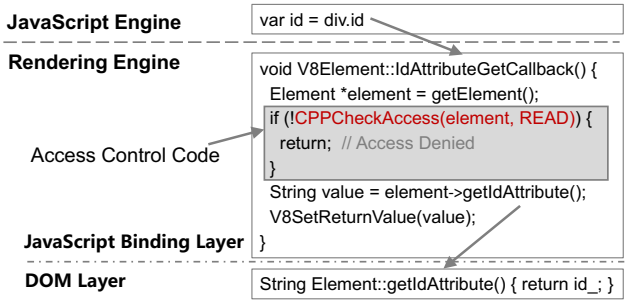

Figure 3: Implementation of DOM access mediation.

DOMinator can monitor all the accesses to the protected elements. For each protected resource, the policy generator updates policy rules with proper principal and access rights according to the collected access traces. Specifically, to trigger more JavaScript code, we use automated random UI testing to simulate user interactions so that we can observe more potential JavaScript accesses. Note that the generated policy is not the final version since there are some limitations to this policy generation process. For example, some third-party script code (*e.g.* CAPTCHA verification) may be hard to be triggered through random UI testing. In addition, without internal knowledge about one specific website, we need to make an assumption that all the accesses monitored during the policy generation process are benign. However, some of the third-party script access can be unnecessary and should be denied, which is non-trivial to determine automatically. Therefore, web developers can further adjust the basic policies generated by our tool to disallow or allow certain scripts to access specific resources with very limited human efforts.

5 IMPLEMENTATION

We implemented a prototype of DOMinator in the Chromium browser of version 88.0.4303.1 using around 1700 lines of code. The extension consists of a labeling tool and a policy generator, which are implemented in JavaScript and Python, respectively.

5.1 Policy Support and Enforcement

DOMinator is primarily implemented in the browser rendering engine. To support internal CPP policies, we added <cpp> as a new HTML element in the Blink rendering engine. We disallow any modification to the policy to ensure the integrity of the mechanism fundamentally. DOMinator matches the policy rules with the corresponding elements as how CSS rules are matched since CPP uses a similar selector. This process does not increase much overhead as it shares some underlying interfaces with CSS. The policy rules are stored as *PolicyData* in the matched objects and their child objects (if any). The PolicyData is also immutable—it cannot be modified by any JavaScript code—to prevent any script from bypassing the access control. Note that some protected elements might not be parsed yet by the HTML parser when the policy is being analyzed. Therefore, DOMinator also performs a policy match for each newly added element.

To correctly identify the principal of dynamically included scripts, DOMinator monitors the script inclusion process via methods ① HTML script element, ② JavaScript URLs, and ③ JavaScript

event listeners. Since a script is not necessarily executed immediately upon inclusion (especially for ② and ③), DOMinator needs to record and assign the initiator's principal to the included script. DOMinator monitors the script inclusion by hooking the browser's internal functions. There are many JavaScript APIs that can be used for each inclusion method, and it is tedious to hook them one by one. For example, a script can dynamically inject script through ① by calling `document.write(<script>...<script/>)` or `document.createElement("script")`, or setting `outerHTML` or `innerHTML` attribute of an existing element. We observe that in the renderer engine, all these APIs need to invoke the script object constructor. Therefore, we simply hook the constructor function to cover all possible ways to include scripts through ①. Similarly, we hook the attribute parser function of some tags (as summarized in [31]) for ② and the event handler object constructor function for ③. Such a design requires less browser code modification than JSIsolate, which hooks all possible APIs. When the HTML script element (①), JavaScript URL in `href` or `src` attribute (②), and event handler (③) are created or modified, DOMinator checks the call stack and records the initiator (if any). When the script is about to be executed, the recorded initiator is assigned as the principal of the script so that DOMinator can effectively track the dynamic script inclusion and correctly identify the actual principal. Compared to prior works, DOMinator covers all the methods and applies a more concise method, which only modifies critical functions instead of all API entries.

In our prototype, we implemented the policy enforcement code in the JavaScript-DOM bindings layer to mediate JavaScript access to the DOM objects. As shown in Figure 3, we mediate JavaScript interfaces and perform permission checks in the DOM-JS Binding layer. We need to add such a policy enforcement code to all the APIs we covered to provide a complete mediation. Chromium uses a code generator to generate the C++ implementation according to the IDL interfaces and corresponding templates. Therefore, we can directly insert code into the templates instead of changing the interfaces one by one. When mediating interfaces, we also need to decide their access type, *e.g.*, READ in the access control code. For the individual methods, we defined their access type through the IDL interfaces so that we could modify the code generator and generate the methods of Read and/or Write operation, respectively. The property interfaces are implemented with two internal functions, `AttributeGetCallback()` (as shown in Figure 3) and `AttributeSetCallback()`, which use Read and Write operation, respectively. Therefore, we can simply modify these two templates to decide the access type for the interfaces. We specially mark that event registration-related APIs require both Read and Write permissions as discussed in §4.3. To mediate the indirect access, we manually added access control codes in the internal implementation of the APIs providing indirect access in the Blink rendering engine.

5.2 Policy Generation

We implemented the labeling tool and policy generator to help reduce human efforts in writing policy rules for web developers. The labeling tool is a browser extension implemented based on Aardvark2. [2] With the extension invoked, developers can hover over the elements and simply label them as sensitive. The extension can generate a sensitive resource list by computing the unique CSS selectors for the labeled elements. Given the sensitive resource list, the policy generator automatically drives the DOMinator to visit the website and gradually updates the policy based on the collected violation traces. In our settings, it generates policy rules at the domain-level granularity, which can be easily configured at other levels, such as subdomain or URL. The policy generator uses the gremlins.js [3] library to perform the random UI testing. In addition, considering some UI actions may cause the browser to navigate to a different page, we implemented an interface in DOMinator to disable the navigation during policy generation. We use a man-in-the-middle proxy [4] during policy generation to inject the generated basic policy rules into the websites so that we can enforce the policy and collect violation logs. It should be noted that, in practice, web developers do not need to use such a proxy. They can directly insert policy into the HTML source as an additional step in their HTML code generation workflow, which can be easily supported in many common web programming frameworks.

6 EVALUATION

We comprehensively evaluate the effectiveness, performance, and compatibility of the Content Protection Policy on protecting sensitive client-side data from over-privileged third-party scripts. We seek to answer the following three research questions: 1) whether CPP can effectively protect sensitive client-side resources from confidentiality and integrity attacks by over-privileged third-party scripts; 2) what the performance overhead of enforcing CPP on real-world websites is with our prototype; 3) how good the usability of the CPP mechanism is, *i.e.*, whether it causes compatibility issues, and how much effort would be required for maintaining CPP policies.

6.1 Experiment Setup

We conduct our experiments on a desktop computer running Debian 10 with a four-core Intel Xeon W-2223 processor and 32GB memory. We use a vanilla Chromium browser in the same version as our prototype as the baseline for comparison. We select the most popular websites from the Tranco list [15] [5] for our performance and usability evaluation. We exclude the invalid domains and those that provide no content for users (*e.g.*, gltd-servers.net) from our evaluation, and we finally get the top 50 reachable websites with the lowest one ranked at 74. We evaluate using only 50 popular websites as they are representative of evaluating a new browser security mechanism: they are very complex and span across most service categories if not all. Since most sensitive information only appears after logging in, we manually registered accounts and logged in on the websites that support user accounts.

6.2 Policy Generation

We generate policies for the 50 websites and inject them as internal policies into the websites through a man-in-the-middle proxy in the experiments. When labeling sensitive elements, we use the

[2] https://chrome.google.com/webstore/detail/aardvark2

[3] https://github.com/marmelab/gremlins.js

[4] https://mitmproxy.org/

[5] Available at https://tranco-list.eu/list/Q9PN4.

following strategies since it is hard to identify the sensitive elements without sufficient knowledge of the website: 1) all <input> elements as users might provide personal data via them; 2) the elements storing apparent personal information (*e.g.*, name, email address). Using the strategies, we cannot identify sensitive elements on four websites and do not generate policies for them, which include whatsapp.com, t.co, *etc.* Nevertheless, CPP is able to protect any content from unauthorized access on these websites if a policy is specified by the developers. We perform five iterations of policy updates to get a final policy.

We could generate a final policy for 91.30 % (42/46) of the websites within three iterations, *i.e.*, after three rounds, we did not observe sensitive element accesses made by new scripts. All websites reach a stable policy within four iterations except for zoom.us, of which the policy was updated in the fifth round due to the accesses by https://www.google-analytics.com. The policies for all the websites have less than 5 policy rules. We do not observe a strong correlation between the number of policy rules and the complexity of the websites. The reason might be that the number of identified sensitive elements across the websites does not differ much. However, it is possible that the policy of some websites might have many more rules if the developers specify a lot of sensitive content for protection. The results show that on most websites, the developers can use the automated policy generator to easily generate the policy with limited effort. Again, in practice, the developers can also flexibly specify the policy rules according to their preferences and security needs, *e.g.*, they can label sensitive elements based on their criteria and directly specify the trusted third-party scripts.

6.3 Effectiveness

We design several experiments to evaluate the effectiveness of DOMinator in providing integrity and confidentiality protection to web contents from over-privileged third-party scripts on websites deployed locally. We also evaluate the effectiveness on 20 real-world websites previously reported with data exfiltration [23].

Confidentiality Protection. The confidentiality attack discussed in §2.1 is possible because third-party scripts can arbitrarily access any DOM elements in the same frame, including the sensitive ones that they do not require for providing their services. To mitigate such confidentiality attacks, we can label elements of `auth` class as sensitive and define a default policy for them, like the first rule shown in Listing 4. The developer can additionally grant access to some trusted third-party scripts (*e.g.*, the ones hosted on another domain of the same company) using a policy rule like the second one in Listing 4. Although we use a resource `".auth"` for both rules, the developers can flexibly adjust the protected resources according to their security needs, *e.g.*, `input[type="password"]` or `input`.

To evaluate the effectiveness of the above CPP rules on preventing unauthorized scripts from reading the protected element content, we implemented two attacks. In the first attack, a third-party script attempts to directly read the "value" attribute of the protected input field. In the second attack, a third-party script attempts to register an event listener that listens for the "keypress" event on the protected input field to record what the user types in that field. By deploying and enforcing the above policy with DOMinator, none of the attacks can be successfully launched.

```
<cpp>
  .auth {
    "default": "None",                  // Rule #1
    "https://example.com/b.js": "R", // Rule #2
  }
</cpp>
```

Listing 4: CPP policy example for password protection.

```
<cpp>
  #target {
    "default": "None",                  // Rule #1
    "example.com/b.js": "W",            // Rule #2
  }
  a {
    "default": "None",                  // Rule #3
  }
</cpp>
```

Listing 5: CPP policy example to mitigate click interceptions.

As discussed in §4.3.2, DOMinator provides protection from only accesses to the DOM content. If the sensitive content is stored in a global JavaScript object by an authorized script, our design does not prevent third-party scripts from accessing this JavaScript object, and it is out of our research scope. This can be solved by either dynamic taint tracking [13] or script isolation [31].

Integrity Protection. Third-party scripts can abuse their privilege to arbitrarily modify the client-side contents, including key elements like the sign-in button or payment button. For instance, they can modify existing hyperlinks to perform click interception attacks Listing 2. To mitigate the attack, developers can define CPP policy for the key objects to disallow access from third-party scripts, *e.g.*, the first rule in Listing 5. They can further relax restrictions by granting write permission to trusted scripts, *e.g.*, the second rule in Listing 5. They can also tighten restrictions by extending protection to all the anchor elements in the website as the third rule.

To evaluate the effectiveness of CPP rules for integrity protection, we implemented two special attacks. In the first attack, a third-party script attempts to modify the `"href"` attribute of an `<a>` element, which is under a protected `<div>` element. Although no policy rule is explicitly defined for the enclosed `<a>` element, by our design, it should inherit the rules from its ancestor (the `<div>` element) and thus be protected. In the second attack, a third-party script attempts to modify the hyperlink of a protected `<a>` element by overwriting its parent element through the `innerHTML` API. Unsurprisingly, both attacks were successfully blocked in our experiment.

Circumvention of Protection. Although CPP is designed as tamper-resistant, it is still necessary to validate if it is implemented and deployed correctly. To test the completeness of the protection enforced by DOMinator, we also implemented several other ways to execute the malicious code. In the first way, a third-party script inserts an inline script into the targeting frame and makes the sensitive content access in the inline script. In the second way, a third-party script encodes the sensitive content access code in a string and invokes it through the `eval()` API. In the third way, a third-party script registers an event listener on other unprotected elements and makes the access in the corresponding callback event handler function. In the fourth way, a third-party script modifies the hyperlink of an anchor element to a JavaScript URL, *i.e.*, `'javascript: ...'`, in which the injected code would be executed to make the access to

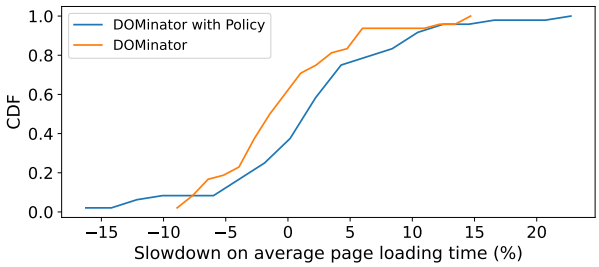

Figure 4: CDF of slowdown on average page loading time.

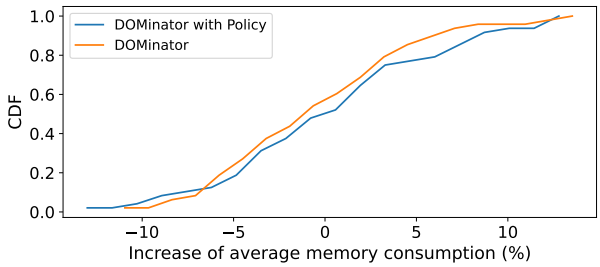

Figure 5: CDF of increase of average memory consumption.

the sensitive elements. We believe that we have covered all the feasible ways of JavaScript execution, as specified in the W3C HTML specification [6]. We found that none of the script execution attempts was successful in reading or overwriting the content (including attributes) of the protected objects.

Considering attackers may also attempt to change or add policy rules to bypass the protection dynamically, we implemented three attacks attempting to access the protected objects by changing the policy rules. In the first attack, a third-party script attempts to remove all <cpp> elements from the DOM to remove all the protection rules. In the second attack, a third-party script attempts to change the permissions of the protected objects by modifying the policy rules in an existing <cpp> element. In the third attack, a third-party script inserts a <cpp> element with a policy rule of the protected objects to grant additional access to the script. By deploying the attacks, we validated that CPP rules cannot be dynamically updated by JavaScript.

Our evaluation demonstrates that our prototype implementation can correctly prevent circumvention of the protection using the methods that we tested.

Effectiveness in Real-world Applications. We also evaluate whether DOMinator is effective in defending against real-world attacks. Since it is difficult to identify attacks on popular websites, we select 20 real-world websites reported in a recent work [23]. It was found on these websites that email and password were exfiltrated by malicious scripts even if the users did not submit the form. We consider the email and password fields as sensitive resources and define a default policy for them, and then we visit the websites and fill out the forms but without submission. We analyzed the violation logs to investigate the suspicious behaviors and gradually relaxed the policy to allow trustworthy scripts that belong to the same entity as the first party. We found the scripts accessing sensitive content were the same as reported in [23] on two websites and were different on the other 13 websites. It should be noted that in our experiment, we not only detected but also denied suspicious access attempts from the third-party scripts, which fundamentally block the attacks. This experiment also shows that even though the third-party script privilege issues are reported, neither the first nor third parties can fully address the problems, which again proves the severity of the problem and the challenges to address it. DOMinator offers an effective and reliable solution.

In summary, our CPP mechanism can effectively protect specified client-side objects from confidentiality and integrity attacks by over-privileged third-party scripts. Moreover, it cannot be circumvented by all the methods we covered. We also showed it can effectively protect from known attacks in real-world applications.

[6]https://html.spec.whatwg.org/multipage/

6.4 Performance

We compare the performance of DOMinator with the vanilla Chromium browser on loading the 50 popular websites. To automate this process, we use Selenium web driver to control the two browsers. To comprehensively evaluate the performance of DOMinator, we measure DOMinator with and without policy enforced and respectively compare the results to the vanilla browser's. As mentioned in §5, we use a man-in-the-middle proxy to inject policies when measuring DOMinator with policy. To make a fair comparison, we also use a proxy but do not inject any policy into the websites when measuring the vanilla browser and DOMinator without policy.

We measure the average page loading time and peak memory use of the two browsers for loading each website. We record the page loading time based on the interface `window.performance.timing`. It reports the time from getting the first HTTP response from the server till the DOM loading is completed. During this process, the website may still send out requests to load some dynamic content, which means we cannot entirely eliminate the impact of network fluctuation. Moreover, our proxy might increase the impact. Therefore, we repeat the measurement for each website twenty times to reduce the variation and get more reliable results. In addition, we filtered out the outliers with a z-score greater than 3. Then, we computed the average loading time and memory consumption of each website visited via Vanilla, DOMinator, and DOMinator with policy enforced, respectively.

The CDF figures of average increases in page loading time and memory usage of the websites are shown in Figure 4 and Figure 5, respectively. The average slowdown of DOMinator and DOMinator with policy is -0.71% and 1.53%. With policy enforced, on 66% websites, the page loading time increased by no more than 5%; and only on one website (https://www.baidu.com) out of the 50 websites, the additional page loading time is over 20%, which reaches 22.75%. With policy enforced, the average memory overhead is 0.03%. For DOMinator without policy, the average memory overhead is -0.78%. The slight negative overheads in the figures were caused by measurement variance or noise, which was observed even in 20 measurements.

In summary, the experiments show that DOMinator introduces little runtime overhead to popular real-world websites.

Table 1: Policy similarity among sub-pages.

Category	# URLs	# URLs with same policies	Percentage (%)	Policy Similarity
search	12	12	100	1.00
section	20	14	70	0.54
news	20	20	100	1.00
privacy and terms	4	4	100	1.00
miscellaneous	9	2	22.22	0.25

6.5 Usability

We evaluated the compatibility with policy enforced through both automated and manual testing and measured the maintenance cost when taking the various sub-pages and frequent updates in web applications into consideration.

6.5.1 Compatibility. We investigate if the enforcement of CPP in DOMinator would cause compatibility issues with real-world websites, *e.g.*, the websites might malfunction because of the privilege limitation enforced by DOMinator. Specifically, we use two approaches to test compatibility. First, we use gremlins.js to perform random UI testing to detect behavior differences. We perform three rounds of random UI testing and configure a thirty-second testing time for each visit. Second, we perform manual interaction with the websites as normal users. We enlisted the participation of three individuals with a solid academic background in computer science to access the websites with the prepared user sessions. In both settings, we apply almost the same operations using the vanilla browser and DOMinator with the policies enforced. We record the number of runtime exceptions in each visit and measure the discrepancies. Since some UI actions may cause the browser to navigate to a different page, we disabled the navigation in our analysis to ensure that the test is performed on the same page. We merged and deduplicated three rounds of results in the two experiments, respectively.

We inspected the collected runtime JavaScript exceptions in random UI testing experiments and observed new ones on four websites. However, they all occurred once in the three rounds of testing. We repeated the experiments for another five rounds but did not reproduce the new exceptions, which suggests these exceptions cannot be stably reproduced. Since the new exceptions on the four websites cannot be easily reproduced, we cannot determine the reason. However, since the random UI testing simulates random user actions in short-time intervals, these new exceptions are likely due to a series of random operations triggering pre-existing design issues on the web page (We need to note that we also observed some exceptions that only occur in the vanilla browser but not in DOMinator). For the manual testing, we did not observe new exceptions. This suggests that enforcing CPP does not break the normal operation of popular and complex real-world websites under common interactions.

6.5.2 Policy Maintenance Cost. In our above experiments, we automatically generated policies for the main page of each website with up to 5 rounds of updates, which requires not much effort if such tests need to be conducted only once. However, different pages of the same website could serve quite different content and functionalities, so they require different policies. Furthermore, as some websites serve quite dynamic content that is constantly being updated, the policy may also require frequent updates. Considering these factors, the policy maintenance cost may be significant and hinder a large-scale deployment.

Table 2: Observed violations in one week.

Day	Domain	Script Domain
1	outlook.live.com	https://cdn.adnxs.com/
2	baidu.com	https://pss.bdstatic.com/
	bing.com	https://cdn.adnxs.com/; https://assets.msn.com
3	yahoo.com	https://cdn.doubleverify.com/
5	instagram.com	https://connect.facebook.net/
	163.com	https://static.ws.126.net
7	outlook.live.com	https://resh3.public.cdn.office.net/

We design two experiments to understand the maintenance cost of CPP policies. First, we measure the similarities among policies generated for different sub-pages of the same website to find out if a small number of policies are sufficient for real large websites. Second, we track the validity of a policy over time to learn if the update of website contents would also require the update of the policies and how frequently such policy updates might be required.

Policy Similarity among Sub-pages. In this experiment, we select a popular news website—Yahoo (https://yahoo.com)—as news websites typically include quite large amounts of diverse content. Starting from its main page, we collected all URLs hosted on the same domain (including subdomains). After deduplicating them and filtering out the URLs that redirected to other websites, we finally got 65 sub-pages. According to their content types, these 65 sub-pages can be categorized into five categories—search, section, news, privacy and terms, and miscellaneous. We then generated policies for them with the same method in §4.4.

After sorting rules in each policy, we computed the pair-wise similarity using the longest common subsequence algorithm, of which the results are shown in Table 1. We found that most websites in the same category shared the same policy rules. We analyzed the six sub-pages with different policies in the second category. We found that five sub-pages with three distinct layouts had policies that differed significantly from the main policy. However, the policy of the remaining sub-page with the same layout was found to have a similarity of 0.7 to the main policy and contained the same resources, albeit with slightly different principals. These results indicate that generating policies for various sub-pages mainly depends on the page layout. The last category includes many types of sub-pages such as mail, help center, and promotions that cannot be categorized into one main category for the limited sample sizes. In practice, the developers can collect more sub-page samples and further generate per-category policies. In summary, our experiment showed that the policy similarity among sub-pages is related to the page layouts.

Policy Validity over Time. We did another experiment to test the policies for one week to evaluate the maintenance cost of deploying the generated policies across time. Similarly, we used random UI testing to trigger JavaScript code and collect violations and exceptions each day. We did not observe new exceptions compared with day 0 but observed new violations on 6 out of 46 websites. This led to the policy update for these six websites. Specifically, five of them required only one update, and one required two updates in our evaluation, as shown in Table 2. We did not observe any new sensitive elements that needed to be protected within one week. Our results show that most websites (86.96%) did not need to update the policy within one week.

We manually analyzed the violations on the other six websites and found these violations were caused by the website update. If

the developers permit the violation, the policy can be automatically updated via the policy generator. We noticed that the sources of these violation traces are mostly from the domain of the same entity as the first party, and thus, they can be permitted. However, on the website outlook.live.com and bing.com, the third-party scripts from cdn.adnxs.com, which provide tracking service, requests to access sensitive resources like profile image and user name. We believe the access requests are not necessary and should be denied. It is important to note that the websites in the experiment include some that are frequently updated, *e.g.*, news websites and shopping websites. Therefore, although our experiment only lasted one week, the result shows that the generated policies are usable in a limited time and do not need to be updated frequently on most websites. In addition, website updates may introduce potential security issues. With the recorded violation traces in DOMinator, the developers can easily monitor the newly included scripts and prevent malicious behaviors.

In summary, the two experiments show that the CPP mechanism is usable in real-world scenarios: it is compatible with real-world web applications, and it does not require much human effort to maintain the policy over time.

7 DISCUSSION AND FUTURE WORK

The third-party script privilege abuse problem is a long-standing and challenging issue that requires significant community efforts to address. We propose a data-centric content protection mechanism to protect sensitive client-side data from over-privileged read or write access. In this section, we discuss the limitations of our work and the possible future works.

Protection Scope. The CPP mechanism has some limitations and can be further improved in the following three aspects to provide comprehensive protection. First, it does not provide content protection in the JavaScript engine. An attacker might access sensitive data indirectly without calling any DOM API. Such problems can be addressed by further introducing some isolation techniques [31] in the JavaScript environment. Second, with CPP, developers can define policies for HTML DOM elements and sensitive JavaScript APIs, which theoretically can be extended to other resources in DOM, *e.g.*, Cookies, HTML5 Web Storage, IndexedDB, and files. Third, we consider browser extensions as trusted, but they might also abuse their privilege. Our design can be extended to control browser extensions' DOM access with modest engineering effort. We leave them as future work to further enhance the CPP mechanism.

Policy Deployment. By systematically evaluating the effectiveness, compatibility, and usability, we show the possibility of deploying the CPP mechanism in the real world. The deployment costs for both the browser vendor and the web developers are limited. In addition, our design only requires modifications in the rendering engine without fundamental architecture changes for extending to other browsers. Thus, the mainstream browsers can deploy the CPP mechanism with reasonable human effort. More importantly, our design explicitly defines the granted third parties to access the sensitive elements. We can further develop extensions to reveal the authorized third parties to the end users before they provide any sensitive information. It can ensure end users' right to transparency over how their PII is being handled and significantly empower end users to prevent data exfiltration.

Policy Generation. Although we designed a labeling tool and a policy generator to help developers write policy rules, it is still limited and not fully automated. First, we cannot automatically generate basic policy rules. It still takes some human effort to label sensitive elements. One possible solution is to automatically detect sensitive information in web applications, which is still an open question. Second, we cannot generate proper policy rules with our tool on some websites. For example, we observed that linkedin.com dynamically assigned `id` to the elements; consequently, the id selectors cannot be reliably used for identifying resources on this website. The dynamically generated elements can be easily supported by updating the server code for generating the HTML code. For instance, developers can simply define and assign a new CSS class name to the sensitive elements in their page templates/views.

8 RELATED WORK

Investigation of Threats from Third-party Scripts. Some papers have studied the security threats from third-party script inclusion and privilege abuse. Lauinger *et al.* [14] studied the use of vulnerable or outdated JavaScript libraries over 133K websites, which can be utilized by other scripts for malicious purposes. Zhang *et al.* [32]found that some embedded scripts perform click interception attacks. Some recent works [3, 23] investigated data exfiltration attacks from third-party scripts in the real world. These studies indicate that threats from third-party scripts widely exist and are non-trivial to resolve. Our research provides an effective method to mitigate threats from third-party scripts.

Confining JavaScript. Many prior studies restrict the functionality of untrusted scripts or constrain them in an isolated environment to limit their privilege. ConScript [17], Treehouse [11], Caja [1], ADsafe [2], ADsafety [20] and WebJail [29] support security policies that restrict available functionality of third-party scripts or mashup components. Phung *et al.* [19], BrowserShield [22], JSand [4], and JaTE [28] rewrite or wrap JavaScript code to build sandbox or restrict specific JavaScript functionality. JSIsolate [31] provides runtime JavaScript isolation to mitigate global name conflicts. While these techniques offer a certain level of protection, their restrictions are usually functionality-centric rather than data-centric, which may lead to compatibility issues and make their deployment difficult in practice.

Data Protection in Web Applications. Some other research focuses on preventing malicious scripts from exfiltrating sensitive data. Ryck *et al.* [8] places sensitive data in shadow DOM trees. ScriptInspector [33] inspects JavaScript accesses to sensitive elements to generate security policies for individual scripts. However, its design introduces high runtime overhead, which makes it difficult to be deployed for policy enforcement. In our earlier work [16], we proposed DOM-ACP to provide fine-grained confidentiality and integrity protection. However, the prototype implementation's policy support is incomplete, and it does not provide good policy generation assistance. In contrast, DOMinator can be easily deployed to provide protection for popular real websites because of its good usability. Over the past years, many papers [5–7, 10, 21, 25, 30] utilized information flow control (IFC) techniques to detect or prevent data

exfiltration attacks. Such techniques can complement DOMinator in providing confidentiality protection from indirect sensitive data access to trusted global JavaScript objects. However, these techniques generally cause high performance overhead (ranging from 50% to 250%). JavaScript isolation mechanisms like JSIsolate [31] can isolate untrusted scripts from trusted ones to protect sensitive JavaScript data. While being effective in protecting confidentiality, the aforementioned techniques usually do not provide an integrity guarantee.

9 CONCLUSION

This paper presents Content Protection Policy (CPP), a new web security mechanism for providing fine-grained confidentiality and integrity protection for sensitive client-side user data. It provides object-level protection instead of page-level protection for critical content in web applications. With the CPP mechanism, developers can define flexible policy rules to restrict third-party script privilege. We design and develop a system—DOMinator—to support the CPP mechanism in the browser. We also implemented a policy generator extension to help developers write basic policies. We conducted a range of experiments to systematically evaluate the effectiveness, performance, and usability of DOMinator using popular real-world websites.

ACKNOWLEDGMENT

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No.: CUHK 24209418).

REFERENCES

[1] 2022. GitHub - googlearchive/caja: Caja is a tool for safely embedding third party HTML, CSS and JavaScript in your website. https://github.com/googlearchive/caja.
[2] 2022. Making JavaScript Safe for Advertising. - Adsafe- gut zu wissen! https://www.adsafe.org/.
[3] Gunes Acar, Steven Englehardt, and Arvind Narayanan. 2020. No boundaries: data exfiltration by third parties embedded on web pages. *Proceedings on Privacy Enhancing Technologies* 2020, 4 (2020), 220–238.
[4] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 1–10.
[5] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium.. In *NDSS*.
[6] Andrey Chudnov and David A Naumann. 2015. Inlined information flow monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 629–643.
[7] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 748–759.
[8] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2015. Protected web components: Hiding sensitive information in the shadows. *IT Professional* 17, 1 (2015), 36–43.
[9] Steve Englehardt, Gunes Acar, and Arvind Narayanan. 2018. No boundaries for credentials: New password leaks to Mixpanel and Session Replay Companies.
[10] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.
[11] Lon Ingram and Michael Walfish. 2012. Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves.. In *USENIX Annual Technical Conference*. 153–164.
[12] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Towards precise and efficient information flow control in web browsers. In *Trust and Trustworthy Computing: 6th International Conference, TRUST 2013, London, UK, June 17-19, 2013. Proceedings 6*. Springer, 187–195.
[13] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2015. Crowdflow: Efficient information flow security. In *Information Security: 16th International Conference, ISC 2013, Dallas, Texas, November 13-15, 2013, Proceedings*. Springer, 321–337.
[14] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2018. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918* (2018).
[15] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. https://doi.org/10.14722/ndss.2019.23386
[16] Wei Meng. 2017. *Identifying and mitigating threats from embedding third-party content*. Ph. D. Dissertation. Georgia Institute of Technology.
[17] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. 481–496.
[18] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. 2022. Scriptprotect: mitigating unsafe third-party javascript practices. In *Proceedings of the 17th ACM Asia Conference on Computer and Communications Security (ASIACCS)*. Nagasaki, Japan.
[19] Phu H Phung, David Sands, and Andrey Chudnov. 2009. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. 47–60.
[20] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety:Type-Based Verification of JavaScript Sandboxing. In *20th USENIX Security Symposium (USENIX Security 11)*.
[21] Vineet Rajani, Abhishek Bichhawat, Deepak Garg, and Christian Hammer. 2015. Information flow control for event handling and the DOM in web browsers. In *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 366–379.
[22] Charles Reis, John Dunagan, Helen J Wang, Opher Dubrovsky, and Saher Esmeir. 2007. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)* 1, 3 (2007), 11–es.
[23] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. 2022. Leaky Forms: A Study of Email and Password Exfiltration Before Form Submission. In *Proceedings of the 31st USENIX Security Symposium (Security)* (2022). Boston, MA, USA, 1813–1830.
[24] Steven Sprecher, Christoph Kerschbaumer, and Engin Kirda. 2022. SoK: All or Nothing-A Postmortem of Solutions to the Third-Party Script Inclusion Permission Model and a Path Forward. IEEE, 206–222.
[25] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. 2014. Protecting Users by Confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 131–146.
[26] Mike Ter Louw, Karthik Thotta Ganesh, and VN Venkatakrishnan. 2010. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements.. In *USENIX Security Symposium*. 371–388.
[27] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. 2012. JavaScript in JavaScript (js. js): Sandboxing Third-Party Scripts.. In *WebApps*. 95–100.
[28] Tung Tran, Riccardo Pelizzi, and R Sekar. 2015. Jate: Transparent and efficient javascript confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference*. 151–160.
[29] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2011. WebJail: least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 307–316.
[30] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. 2009. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the 4th ACM European conference on Computer systems*. 233–246.
[31] Mingxue Zhang and Wei Meng. 2021. JSISOLATE: Lightweight In-Browser JavaScript Isolation. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece.
[32] Mingxue Zhang, Wei Meng, Sangho Lee, Byoungyoung Lee, and Xinyu Xing. 2019. All Your Clicks Belong to Me: Investigating Click Interception on the Web. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA, USA.
[33] Yuchen Zhou and David Evans. 2015. Understanding and monitoring embedded web scripts. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 850–865.

A APPENDIX

A.1 Policy Generation

The list of tested websites and the number of generated policy rules are shown in Table 3 and Figure 6, respectively.

Table 3: Top 50 accessible websites.

Rank	Domain	Rank	Domain	Rank	Domain	Rank	Domain	Rank	Domain
1	google.com	13	linkedin.com	28	wordpress.org	44	vimeo.com	65	163.com
3	youtube.com	14	wikipedia.org	29	office.com	45	adobe.com	66	webex.com
4	facebook.com	15	cloudflare.com	30	bing.com	46	yandex.ru	67	vk.com
5	microsoft.com	16	yahoo.com	31	pinterest.com	48	zoom.us	68	blogspot.com
7	netflix.com	19	qq.com	32	github.com	50	gandi.net	69	mozilla.org
8	twitter.com	20	live.com	37	doubleclick.net	52	wordpress.com	70	sina.com.cn
9	amazonaws.com	21	amazon.com	39	mail.ru	54	goo.gl	71	intuit.com
10	instagram.com	22	bilibili.com	40	reddit.com	58	bit.ly	72	t.co
11	baidu.com	23	azure.com	41	whatsapp.com	59	windows.net	73	tiktok.com
12	apple.com	27	fastly.net	42	zhihu.com	62	taobao.com	74	googledomains.com

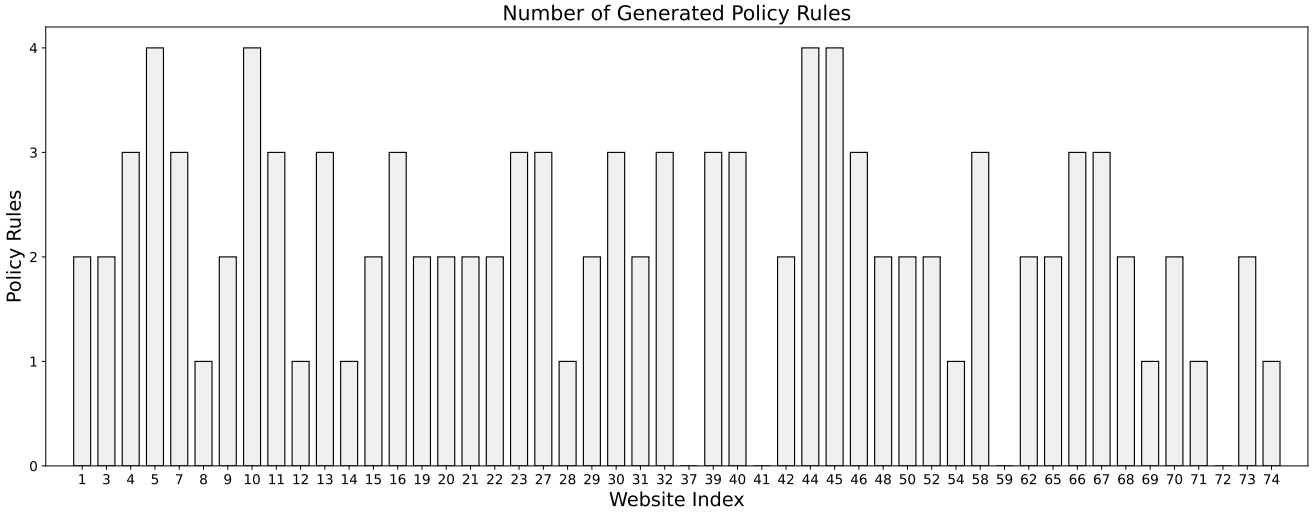

Figure 6: Number of generated policy rules.