Beast in the Cage: A Fine-grained and Object-oriented Permission System to Confine JavaScript Operations on the Web

Anonymous Author(s)

ABSTRACT

JavaScript plays a crucial role on web. However, the inclusion of unknown, vulnerable, or malicious scripts on websites and in browser extensions and the use of browsers' developer tools often leads to undesired web content manipulations and data acquisitions. To restrict JavaScript operations on web content and data, we introduce a fine-grained, mandatory access control-based, and object-oriented permission system for browsers. With our system, web developers can define policies for sensitive web elements on their web pages to allow or deny scripts' operations on web content and data within browsers. The system substantially thwarts many web threats and attacks, and offers benefits to personal data governance. We developed a tool for automatic policy generation and demonstrated the usability and compatibility of the system in a three-month study. Our system is a reasonable and practical solution, bolstering the security and trustworthiness on the internet.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

HTML, JavaScript, permission

ACM Reference Format:

Anonymous Author(s). 2025. Beast in the Cage: A Fine-grained and Object-oriented Permission System to Confine JavaScript Operations on the Web. In *Proceedings of The Web Conference (WWW '25).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

JavaScript plays a crucial role on web, offering dynamic functions and interacting with HTML documents. However, the inclusion of unknown, compromised, or malicious scripts on websites [9, 19, 34] and in browser extensions [15, 20, 23, 28, 33, 36] often leads to undesired web content manipulations and data acquisitions. In 2018, Feedify's JavaScript library was repeatedly compromised to siphon off victims' payment card details on e-commerce websites globally. Recent studies have unveiled a wide range of abuses among browser extensions, from ad injection [20] to the insertion of rogue web elements for malware installation [28, 33].

Browsers' developer tools, particularly its DOM inspector and JavaScript console, have become another vector to compromise the integrity of web applications. In the emerging online refund scams [1], the scammer remotely controls a victim's computer, manipulates transaction records displayed on the victim's online banking page using the DOM inspector in the victim's browser, shows the victim has been overpaid on a refund, and urges the victim to return the faux overpayment.

To restrict JavaScript operations on web applications and data, we introduce a fine-grained, mandatory access control-based, and object-oriented permission system to browsers. With the HTML tag and attributes offered by the system, web developers can define policies for sensitive web elements on their web pages to allow or deny scripts' operations on web content and data within browsers. Operations on web content include updating and removing nodes on DOM trees, reading and writing HTML properties and attributes, updating HTML event listeners, interacting with HTML elements, and dispatching HTML events. Operations on data include writing data to the HTML document stream, accessing data in cookies and local storage, and sending data to the network. The system can also restrict manual updates to web content via the DOM inspector.

We implemented the system in the Chromium browser's Blink engine and ensured permission policies are immutable in the browser. We enforced permissions in a list of JavaScript APIs that can be used to operate HTML documents and web data. We carefully addressed challenges brought by the dynamic content, JavaScript eval, web workers, and potential attacks.

We conducted micro-benchmarks to demonstrate the minimal performance overhead introduced by the system's key components. A macro-benchmark was performed to assess the system's overall performance overhead on Tranco's top 300 websites under the extreme condition, showing that our system incurs limited overhead.

Case studies then demonstrates that the system can be used as both whitelist and blacklist-based solutions to counter many web threats and attacks with no impact to the functions of the browser and browser extensions. The system also offers benefits to personal data governance, complying with data privacy laws and legislation.

To enhance usability and reduce maintenance cost when using the system, we developed a tool which monitors scripts' operations on web elements and data and automatically generates policies for web pages. We successfully generated policies for the index pages and sign-in pages of 30 popular websites, and validated their correctness through manual inspections, illustrating the system's usability and compatibility. A three-month study showed that the generated policies are usable within a certain period of time even on frequently updated news and shopping websites.

Researchers have proposed various solutions to restrict JavaScript features and protect user data on the client side. Unlike function-centric approaches that isolate HTML elements and scripts [10, 16–18, 21, 22, 31], enforce information flows [4, 7, 8, 27], and replace

WWW '25, April 28–May 2, 2025, Sydney

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

[1] https://www.youtube.com/watch?v=X4PllvUowaQ

resources [25], our approach is object-oriented and can avoid significant runtime overhead. It does not require the knowledge of data legitimacy used for traffic-based approaches [6, 24, 26] or the knowledge of data types used for the type checking-based approach [14]. It can fulfill diverse security requirements at a finer granularity than [11, 12]. Most recently, a permission system was proposed to confine JavaScript operations on HTML documents [32] but our work differs from it in several aspects, as discussed in Section 8.1. Meanwhile, existing HTML features cannot meet our needs.

In summary, we have made the following contributions: (1) We introduced a permission system to restrict JavaScript operations on web applications and data. (2) We implemented the system as a ready-to-use solution with limited performance overhead. (3) We demonstrated its effectiveness in preventing many web-based threats and attacks. (4) We developed a tool for automatic generation of policies when using the system and showed the system's usability and compatibility in a three-month study.

2 DESIGN

This section starts with the threat model, followed by the policy syntax and rules required by the permission system to ensure the confidentiality and integrity of web applications and data.

2.1 Threat Model

We consider first-party and third-party scripts on web pages and in browser extensions as well as scripts submitted to the JavaScript console of browsers' developer tools as attacking sources. They could be compromised, over-privileged, or accidentally included to manipulate web applications and obtain sensitive data in browsers without authorization, compromising the confidentiality and integrity of the targeted web applications. We also consider web page manipulations manually through the DOM inspector of browsers' developer tools as attacks to web applications.

We do not focus on code injection attacks, that can be mitigated by mechanisms like Content Security Policy (CSP). However, CSP is not sufficient to defend against the attacks we are targeting. We do not focus on web content mutations in HTML hijacking either, where malicious entities can control and manipulate HTML documents; it is essential to explore other security measures to address this threat. Non-JavaScript operations on HTML documents (e.g., transitions and animations in CSS) and unknown attack vectors to compromise web applications and data are also out of our scope.

2.2 Policy

We introduce the following syntax for web developers to define two types of permission policies for HTML elements and data in HTML documents.

2.2.1 For Operations on Elements. The first type of policies enforces permissions for read and write operations on HTML elements, as shown below. We use `"-"`, `"r"`, `"w"`, and `"*"` in `<permission>` to allow none, read, write, and all operations, respectively. Operations include updating and removing nodes on DOM trees, reading and writing HTML properties and attributes (e.g., identifier, name, type, style, event, text, and value), updating HTML event listeners, interacting with HTML elements, and dispatching HTML events.

These operations can be performed by JavaScript on web pages and in browser extensions, denoted as `<scripts>` and `<extensions>`, as well as via the DOM inspector. We correspondingly use script and extension identifiers `<script_id>` and `<extension_id>` to uniquely label scripts. We can also leverage the URL `<url>` or domain name `<domain>` to refer to a particular script or a set of scripts from a domain. We use `"-"` and `"*"` to represent no operator and every operator. The syntax currently cannot match a set of browser extensions; we leave it for future work. We will extend the syntax later in Section 3.3.

Policy identifiers `<policy_id>` can be concatenated by `","` in the `pid` attribute of an HTML element. The `<script_id>`, `<extension_id>`, and `<policy_id>` facilitate the reuse of scripts among policies and the reuse of policies among HTML elements. When an element is not assigned any policy, every operation will be allowed on it.

```
// policy for elements
<policy_id> : {
  <policy_item>, ...
}
<policy_item> ::= <permission> : {
  "sids" : "-"|<scripts>|"*",
  "eids" : "-"|<extensions>|"*"
}
<permission> ::= "-"|"r"|"w"|"*"
<scripts> ::= [<script_id>|<url>|<domain>]
<extensions> ::= [<extension_id>]
```

To create a policy for an HTML element operated by scripts on a web page, a web developer needs to: (1) create the unique identifier `<script_id>` in scripts' `sid` attribute, especially for inline scripts, (2) create a policy in the policy file with a unique identifier `<policy_id>`, (3) use the `<script_id>`, `<url>`, or `<domain>` in the policy to match a particular script or a group of scripts and set their permissions, (4) include the policy file on the page using the HTML `<policy>` tag, and (5) place the `<policy_id>` in the policy attribute `pid` of the operated HTML element. An example policy `"pwd"` for operating password fields is shown below. The fields can be read by the script with `sid` `"s"` and scripts from www.google.com. They can also be updated by the browser extension with `eid` `"hdokiejnpimakedhajhdlcegeplioahd"`.

```
"pwd": {
  "r": { "sids": ["s"], "eids": "-" },
  "r": { "sids": ["www.google.com"], "eids": "-"},
  "w": { "sids": "-", "eids": ["
       hdokiejnpimakedhajhdlcegeplioahd"] } }
```

2.2.2 For Operations on Data. The other type of policies enforces permissions for JavaScript's read and write operations on cookies and local storage, and for JavaScript's write operations on the HTML document stream and web domains, as shown below. We consider IP addresses as alternatives to web domains. When a script element is not assigned any policy regarding data operations, its code is free to access and dispatch data.

```
// policy for script elements
<location> : {
  <policy_item>, ...
}
<location> ::= "cookie"|"storage"|"stream"|<domains>
<domains> ::= [<domain>]
```

To set a policy for an HTML script element operating data, a web developer needs to create a unique identifier `<script_id>` for the script element and update the policy file. Example policies are

shown below. Only the script with `sid "s"` can read and write cookies but no script can send data to www.google.com.

```
"cookie": {
  "*": { "sids": ["s"], "eids": "-" } },
"www.google.com": {
  "w": { "sids": "-", "eids": "-" } }
```

2.3 Policy Rules

The enforcement of policies is following the rules defined below. Examples for these rules are provided in Appendix A.

Rule 1. The policies, the HTML `<policy>` tag, and the `sid`, `pid` attributes cannot be added, altered, or removed by JavaScript or in the developer tools, otherwise the permission system is invalidated.

Rule 2. When an HTML element is not assigned any policy, it will inherit the policy from its closest ancestor that has one.

Rule 3. An HTML element's own policy overrides that of its ancestors. In the rest of this paper, we refer to an HTML element's policy as either its own or the one inherited from its ancestors.

Rule 4. An HTML element can be read or written by a script when itself and all the involved descendants are readable or writable to the script, otherwise inoperable descendants would be accidentally operated. Similarly, an HTML element can be updated in the DOM inspector when itself and all the involved descendants are writable to every operator.

Alternatively, when an element is being operated, the permission system can force the operator to ignore the inoperable descendants. However, this approach requires more complicated instrumentation, brings extra operations on the descendants, and it can be expensive to maintain the atomicity. We leave it for future work.

Rule 5. User actions on an HTML element (e.g., click a button and send a form) can be simulated in a script when the element is writable to the script.

Rule 6. A data location can be read or written by a script when it is readable or writable to the script.

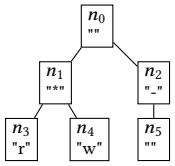
Rule 7. There is no restriction on JavaScript operations on an HTML element when no policy is assigned to the element, and no restriction on a script's operations on any data location when no policy is assigned to the script.

3 IMPLEMENTATION

We implemented the system in the Blink engine of the Chromium browser of version 107.0.5299.0. In this section, we will provide implementation details about policies, HTML attributes and properties, and the permission enforcement in our interested JavaScript APIs. We will also discuss our solutions to the challenges brought by the dynamic content, JavaScript eval, web workers, and potential attacks.

3.1 Policy File

We created the HTML `<policy>` element and use its `src` attribute to include the policy file in the HTML document. We correspondingly added the `HTMLPolicyElement` class to the Blink engine. In its constructor, we parse the policy file and cache policies using the three tuple `<policy_id|location, permission, [script_id|extension_id|url|domain]>`. We added the `getPolicy` API to the `Document` class to obtain the `HTMLPolicyElement` object globally. In our interested JavaScript APIs operating HTML elements and data, we use the `getPermission` API of the `HTMLPolicyElement` object to obtain the permission (i.e., `"-"`, `"r"`, `"w"`, or `"*"`). Such an API takes in the associated policies of the operated element or the data location, and the operating scripts.



(a) A DOM tree with permissions.

	$P_\perp$	$P^\top$
n_0	*	-
n_1	*	-
n_2	-	-
n_3	r	r
n_4	w	w
n_5	-	-

(b) $P_\perp$ and $P^\top$ in Figure 1a.

Figure 1: $P_\perp$ and $P^\top$'s computation example.

3.2 Immutable Policies and Attributes

To ensure policies are immutable, we (1) define policies as a JSON object in the policy file, (2) abort the addition and removal of the HTML `<policy>` element, (3) reject the update to the HTML `<policy>` element, especially its `src` attribute, in the `setAttribute` and `removeAttribute` APIs, and (4) do not export the members of the `HTMLPolicyElement` class as JavaScript APIs.

To ensure the attributes `sid` and `pid` are immutable, we (1) reject the operation on them in the `setAttribute` and `removeAttribute` APIs, (2) examine whether they will be modified when setting the properties of prop_1 in Table 2, and abort the process if true, and (3) do not export them in the Interface Definition (IDL) files.

3.3 Policy Enforcement

To enforce the first type of policies in a JavaScript API operating an HTML element, we call the `getPermission` API to obtain the element's permission $P_\perp$ given the element's associated policies and the operating scripts. The associated policies can be identified according to Rules 2 and 3. The operating scripts can be extracted on the call stack in the V8 engine using `v8::StackTrace::CurrentStackTrace()`; we will provide more details later in this section.

When descendants $c_1, \ldots, c_n$ are involved in the operation on the element p, we further compute $P^\top$ with Formula 1, to inspect the permissions of the element and its descendants and determine whether they can be operated at the same time according to Rule 4.

Figure 1a shows a DOM tree with six nodes and their permissions. n_0 is granted the default permission `"*"`, n_1 and n_2's permissions override n_0's, n_3 and n_4's permissions override n_1's, and n_5 inherits the permission from n_2, as shown in the second column of Figure 1b. The $P^\top$ of those nodes is computed using Formula 1 and shown in the last column of Figure 1b.

To enforce the second type of policies, we compute $P_\perp$ with the data location and the operating scripts according to Rule 6.

$$
\begin{aligned}
P^\top(p) = & P_\perp(p) \cap P_\perp(c_1) \cap \ldots \cap P_\perp(c_n), \text{ where} \\
& c_1, \ldots, c_n \text{ are } p\text{'s descendants, and } P \cap * = P, \\
& P \cap - = -,\ P \cap P = P \text{ when } P \in \{*, w, r, -\}, \\
& P \cap P' = - \text{ when } P \in \{w, r\},\ P' \in \{w, r\},\ P \neq P'.
\end{aligned} \tag{1}
$$

Instrumented JavaScript APIs. We instrumented a list of our interested JavaScript APIs, HTML properties and attributes in the

	JavaScript API	Operation	Condition to Execute
1	parent.[insertBefore, appendChild, append, insertAdjacentElement] (...)	write	$P_{\perp}(parent)$ == '*' | w'
2	parent.replaceChild (new, old)		$P^{\top}(old)$ == '*' | w'
3	old.replaceWith (new, ...)		
4	parent.replaceChildren (new, ...)		$P^{\top}(parent.children)$ == '*' | w'
5	parent.removeChild (child)		$P^{\top}(child)$ == '*' | w'
6	node.remove ()		$P^{\top}(node)$ == '*' | w'
7	element.$prop_1$	write	$P^{\top}(element)$ == '*' | w'
8	element.setAttribute ($attr_{style}$, ...)		
9	element.removeAttribute ($attr_{style}$)		
10	element.$prop_2$		$P_{\perp}(element)$ == '*' | w'
11	element.setAttribute ([$attr_{value}$, $attr_{url}$], ...)		
12	element.removeAttribute ([$attr_{value}$, $attr_{url}$])		
13	element.$prop_1$	read	$P^{\top}(element)$ == '*' | r'
14	element.$prop_2$		$P_{\perp}(element)$ == '*' | r'
15	element.getAttribute ($attr_{value}$)		
16	element.setAttribute (<on+$attr_{event}$>, ...)	write	$P_{\perp}(element)$ == '*' | w'
17	element.removeAttribute (<on+$attr_{event}$>)		
18	element.addEventListener ($attr_{event}$, ...)		
19	element.[submit, click] ()		
20	element.dispatchEvent (...)		
21	document.[write, writeln] (...)	write	$P_{\perp}(script)$ == '*' | w'
22	document.cookie		
23	localStorage.setItem (...)		
24	XMLHttpRequest.open (url, ...)		
25	navigator.sendBeacon (url, ...)		
26	window.fetch (url, ...)		
27	window.postMessage (..., url)		
28	element.setAttribute($attr_{url}$, url)		
29	document.cookie	read	$P_{\perp}(script)$ == '*' | r'
30	localStorage.getItem (...)		

Table 1: The instrumented JavaScript APIs, HTML properties and attributes, and their permission enforcement.

Blink engine with the help of IDL files, as shown in Table 1. The list may be incomplete and we will continue improving it.

We instrumented JavaScript APIs for adding, replacing, and removing nodes on DOM trees. When an API updates a node without affecting its children, the enforcement is based on the node's permission, as shown in the item 1. When an API updates a node and perhaps its descendants, the enforcement is based on the node and its descendants' permissions, as shown in items 2-6.

We arranged HTML properties and attributes into six groups - those affect the operated element and its descendants (i.e., $prop_1$ and $attr_{style}$) and those affect the operated element only (i.e., $prop_2$, $attr_{value}$, $attr_{url}$, and $attr_{event}$), as shown in Table 2.

The permission enforcement on setting and getting HTML properties and attributes is shown in items 7-12 and 13-15. We did not instrument the writing to the `src` attribute in `element.src = <url>`, as the `setAttribute` API is eventually invoked in the callback function `SrcAttributeSetCallback`. We ensured there is no conflict between our system and the HTML `contenteditable` attribute, which is for editing the content of HTML `<div>` and `<p>` elements in the browser.

Event listeners are commonly used for sniffing user inputs in input, textarea, and summary elements. The permission enforcement on updating an HTML element's event listeners relies on the element's permission, as shown in items 16-18.

The enforcement on submitting or clicking an HTML element and dispatching events to an element is in the same way, as shown in items 19 and 20. Please refer to Section 8.2 for how to differentiate script-triggered actions from user actions in the item 20.

The permission enforcement on writing data to the HTML document stream, cookies, local storage, and network is illustrated in items 21-28. It relies on the permission granted to the executed script. The enforcement in the item 28 is for data leakage through URL parameters, not contradicting the item 11. Items 29 and 30 illustrate the enforcement on reading cookies and local storage.

Permissions are also enforced on JavaScript operations on the shadow DOM and those performed through the Netscape Plugin Application Programming Interface and Web Assembly.

Dynamic Content. To enforce permissions on an operation on dynamic content constructed by a first-party script, we suggest developers to assign policies to the content during its construction. However, to enforce permissions on an operation on dynamic content constructed by a third party, we suggest developers to assign policies to the HTML element where the constructed content is attached to, so that the policy can be inherited.

To enforce permissions on operations by dynamically constructed inline scripts, we further instrumented several JavaScript APIs. In APIs of items 1-4 (Table 1), we use the `querySelector` to search the inserted inline script elements. In APIs of items 7 and 21, we use regular expressions to locate inline `<script>` tags in the inserted text content. For every inserted inline script, we create the `sid` attribute with its constructor script's identifier or URL. We do not need to handle dynamic external scripts, as their URLs can be directly used in policies. Our approach remains robust to obfuscated scripts. We will discuss our strategy about handling JavaScript eval in dynamic content construction.

The permission system overwrites the `sid` attribute of dynamic inline scripts and removes the `sid` attribute of dynamic external scripts during their construction. This strategy prevents the dynamically loaded scripts from masquerading as other scripts such as a legitimate sign-in script which can access the login form.

Group	HTML Properties and Attributes
$prop_1$	innerText, outerText, innerHTML, outerHTML, textContent
$prop_2$	text, nodeValue
$attr_{value}$	value
$attr_{url}$	src, href
$attr_{style}$	class, style, type, width, height, hidden
$attr_{event}$	change, input, keydown, keypress, keyup

Table 2: Groups of HTML properties and attributes based on their effects.

Scripts in the Call Chain. For $P_\perp$ and $P^\top$'s computation, we can extract only the initiator script of the operation from the stack frame at the bottom of the V8 engine's call stack. However, this approach could introduce vulnerabilities illustrated as below. The policy `"pwd"` aims to allow the access to the password field by the script `"s"` but deny the access by the library `"lib"`. Unfortunately, there is no restriction to the library's access to the password field when its `foo` function is manipulated and invoked by the script `"s"`, because the stack frame of `"s"` is at the bottom of the call stack.

```
/** in policy file **/
"pwd": {
  "*": { "sids": ["s"], "eids": "-" },
  "-": { "sids": ["lib"], "eids": "*" } }
/** HTML code snippet **/
<input type='password' id='pwd' pid='pwd'/>
<script sid='lib'>
  function foo() {
    document.getElementById("pwd").value; // injected
  }
</script>
<script sid='s'> document.getElementById("pwd").value;
    foo();</script>
```

Therefore, we extend `<script_id>`, `<url>`, and `<extension_id>` with the name and location of the invoked functions in the form of `<script_id|url|extension_id>:[<func_name>@<func_location>]` to enforce permissions at the function level. The location consists of line and column numbers (e.g., `"12:20"` for line 12 and column 20) to locate anonymous functions and mitigate the ambiguity among functions in the same name. We then permit operations on HTML elements and data when all scripts and functions on the call stack exactly match those specified in policies and have the permissions. When scripts and functions on the call stack conflict over permissions, we conservatively deny the operation.

In such an approach, the policy in the previous code would allow the access by only the root function of the script `"s"` while block the access directly by the library `"lib"`. In Section 5.2, we will provide another example to show the use of functions in policies. We plan to explore the feasibility of enforcing permissions at the statement level to provide the possibly highest level of security.

To facilitate the identification of scripts and functions involved in HTML and data operations and to avoid conflicting permissions when using our system, we introduce a tool for automatic policy generation and demonstrate our system's usability and compatibility in Section 6.

JavaScript Eval. To enforce permissions on operations by eval'ed code, we skip the stack frames of the eval'ed code when iterating through the call stack. The executor script of eval will participate the permission enforcement. This approach is equivalent to having the eval'ed code inherit its executor's permission.

When eval is used in the construction of inline scripts, we skip the stack frames of the eval'ed code, identify the executor script of eval, and set the identifier of the constructed inline scripts with the executor's `sid` value. The inline script constructed through `document.write` in the code below will inherit the `sid` `"s"`.

```
<script sid='s'> window.eval("document.write(\"<script>
    document.getElementsByName('pwd')[0].value;</script
    >\")"); </script>
```

Web Workers. They are for running scripts in the background without interfering with the user interface. They can make network requests using fetch() or XMLHttpRequest APIs. When an external script is running in a web worker, its URL can be directly used for the permission enforcement. When a web worker is created from a data blob, the script in the data blob will inherit the `sid` value of the creator script identified on the call stack.

Developer Tools. To enforce permissions on web page manipulations through the DOM inspector of the Chromium browser's developer tools, we instrumented selected APIs of the `InspectorDOMAgent` class in the same way as those in Table 1, according to Rule 4.

To recognize the code submitted to the JavaScript console, we examine the name and source URL of the operating scripts using `StackFrame::GetScriptNameOrSourceURL` when iterating through the call stack. A returned empty string for an operating script indicates that the script was launched in the JavaScript console. The system then makes a decision based on the policies.

4 PERFORMANCE OVERHEAD

We evaluate the computational overhead of $P_\perp$ and $P^\top$, measure the time for iterating the call stack, and investigate the system's overall overhead under the extreme condition. The experiments were conducted on an Ubuntu machine with 2.40 GHz CPU cores and 50 GB of memory. The results showed that the system introduces very limited overhead to the browser.

4.1 Micro-benchmark

$P_\perp$ and $P^\top$ Computation. To generate testing web pages, we learned the depth and the width of HTML documents from 346,736 web pages of Tranco's top 10K websites [3]. We define the depth of an HTML document as the maximum number of parent traversals needed to reach the document root from any node in the document. We define the width of an HTML document as the maximum number of children under any node in the document. The average, median, and mode of the HTML document's depth are 6, 3, and 3, respectively, and over 95% of the HTML documents have no more than 21 layers. The average, median, and mode of the HTML document's width are 18, 3, and 2, respectively, and the width of over 95% of the HTML documents is less than 80.

To measure $P_\perp$'s computation time on an HTML document at the depth of d, we created a web page with d recursively nested `<div>` elements. We then set a policy on the outermost `<div>` element. When reading the inner HTML of the innermost `<div>` element, all the d `<div>` elements were recursively traversed in the $P_\perp$'s computation. We had the instrumented Chromium browser visit such a page 30 times and recorded $P_\perp$'s computation time. Table 3 shows that the time for $P_\perp$'s computation is trivial regardless of the HTML document's depth.

Depth	Width	$P_\perp$		$P^\top$	
		Traversed Nodes	Time(ms)	Traversed Nodes	Time(ms)
25	25	25	≈ 0	625	1.47
50	50	50	≈ 0	2,500	2.63
75	75	75	≈ 0	5,625	6.44
100	100	100	≈ 0	10,000	10.47
21	79	21	≈ 0	1,659	2.75

Table 3: Time for $P_\perp$ and $P^\top$'s computation on HTML documents in different depth and width.

To measure $P^\top$'s computation time on an HTML document at the depth of d and the width of w, we recursively nested d `<div>` elements under a root element, spread out w `<div>` elements at each layer, and set a policy on every `<div>` element. When reading the inner HTML of the root element, all the `<div>` elements were traversed in $P^\top$'s computation. As shown in Table 3, $P^\top$'s computation took longer time when the number of traversed nodes increased; however, its overhead is negligible, especially when the HTML document was at the maximum depth of 21 and the maximum width of 79, learned from over 95% of real-world web pages.

Call Stack Iteration. We ran the code below to pile up stack frames until the call stack overflows in the instrumented Chromium browser. We repeated it 30 times and recorded the size of the call stack and the time for iteration through the call stack. The size of the call stack in our experiment was 8,260 and the iteration took 1,418 milliseconds on average, indicating our approach is less likely to introduce significant performance overhead.

```
/** in policy file **/
"pwd": {
  "*": { "sids": ["s"], "eids": "-" } }
/** HTML code snippet **/
<input type='password' name='pwd' pid='pwd'/>
<script sid='s'>
  function callStackSize() {
    try {
      return 1 + callStackSize();
    } catch (e) {
      document.getElementsByName('pwd')[0].value;
      return 1; // overflow
    }
  }, callStackSize();
</script>
```

4.2 Macro-benchmark

To assess the system's overall performance overhead under the extreme condition, we revised its implementation. When loading a web page, the system aggressively computes $P_\perp$ and $P^\top$ in every call interception of JavaScript APIs in Table 1, regardless of the `sid` and `pid` attributes, while the rest of the system remains the same. We instrumented `ClassicScript::RunScriptOnScriptStateAndReturnValue` to record the executed scripts and their execution time.

We visited the index page of Tranco's top 300 websites [3] in this revised browser for six rounds. We cleared the browser cache before each visit and waited 90 seconds for each visit. We did not observe any exception. Statistics of the numbers of the executed scripts, reads, writes, the iterated stack frames, and dynamic scripts is shown in Table 4.

To learn the baseline and compute the performance overhead, we disabled the permission system and visited the same set of pages in the same browser. Our system did not introduce any overhead on 28% websites, while brought the maximal ×0.25, ×0.50, ×0.75, and ×1.00 overhead on 73.67%, 89.67%, 95.67% and 98.33% of all the websites, respectively, under the extreme condition. Appendix D provides the detailed data for top 50 websites.

		Max	Average	Median	Mode
Scripts		249	32	13	8
Reads	$P_\perp$	21,805	605	67	12
	$P^\top$	5,379	237	10	10
Writes	$P_\perp$	11,929	1,033	111	18
	$P^\top$	11,429	514	45	10
Stack Frames		1,752,070	110,061	4,432	137
Dynamic Scripts		163	15	2	0
Overhead		×1.85	×0.18	×0.10	×0.00

Table 4: Statistics of the numbers of the executed scripts, reads, writes, the iterated stack frames, and dynamic scripts, and the overhead on the index pages of top 300 websites.

We analyzed the cause of the overhead and found that the number of the executed scripts, the amount of $P_\perp$ and $P^\top$ computation in read operations, and the number of dynamic scripts impact our system's performance with the Pearson correlation coefficients of 0.13, 0.18, 0.20, and 0.12 to the overhead. Therefore, we suggest developers to perform the finest permission control on essential web elements and scripts to minimize the performance impact. Such a suggested setting is very different from our experiment setting under the extreme condition.

5 SECURITY ENHANCEMENT

We then conduct case studies to illustrate the security enhancements brought by our system. Note again that the system aims to prevent unauthorized operations on web content and data in many web attacks.

5.1 Web Data Access

To demonstrate the system's capability to prevent data from being stealthily collected and exfiltrated in browsers, we set up a web page with a password field and embedded an inline script reading that password field. We assigned the `sid` `"s"` to the inline script and defined a policy to allow that script to read the password field.

We also developed two browser extensions. One reads the password field in its content script while the other one's content script injects an inline script for reading the field, as illustrated below.

```
// the inline script and the 1st extension
var pwd = document.getElementById("pwd").value;
// the 2nd extension
var injScript = document.createElement("script").text = "
    var pwd = document.getElementById('pwd').value;";
document.body.appendChild(injScript);
```

We installed the two extensions in our instrumented Chromium browser and visited that page. The embedded inline script functioned correctly. However, the content script of the first extension and the injected inline script by the second extension failed to obtain the password. We then updated the content script of the second extension to `document.write("<script sid='s'>var leak = document.getElementById('pwd').value;</script>");`, which assigns the `sid` `"s"` to the injected inline script. Thanks to our design for dynamic content in Section 3.3, the injected script failed either.

Auto-save and Auto-fill. To learn the system's impact on these features, we created a website with the sign-in feature and set the password field to read-only or write-only to third-party password managers that are available as browser extensions. We then automatically saved and filled in our login information with the built-in password manager of the instrumented Chromium browser and four popular third-party password managers, i.e., LastPass, RoboForm, KeeVault, and 1Password.

Our system had no impact on both the features of the Chromium browser since they are implemented in native code. The two features of all the third-party password managers were not affected either when policies were correctly defined, although they are implemented in JavaScript.

5.2 Web Content Manipulation

The code below illustrates the defense against unauthorized web content manipulations that could introduce potentially harmful content. We created a policy `"in"` to allow the function `"myFunc"` of the script `"s"` to update the input element only through the jQuery's `val` function. The policy contained a chain of invoked functions and their locations in the script `"s"` and the jQuery library. `"@8381:15"` refers to the location of an anonymous function in the jQuery, while `"each@204:16"` and `"each@382:18"` are two functions in the same name. Thanks to our design for the call stack in Section 3.3, any update to the input element directly by the script `"s"` or through other APIs of the jQuery library will be rejected due to the mismatched chain of scripts and functions. Appendix B shows the chain of scripts and functions involved in such a write to `<input>`.

```
/** in policy file **/
"in": {
  "w": { "sids": ["s":["myFunc@19:20"], "jquery-3.7.1.js"
       :["val@8351:14","each@204:16","each@382:18","@8381
       :15"]], "eids": "-"}}
/** HTML code snippet **/
<script sid='s'> function myFunc() { $('#input').val("
     Geek"); } </script>
<button onclick="myFunc()">Click Me</button>
<input id="input" type="text" value="" pid='in'/>
```

Content Injection. Our system supports both whitelist-based and blacklist-based solutions to block unexpected content injection on websites. For the highest level of protection, developers can set the topmost `<html>` element to inoperable and define policies to whitelist HTML and data operations by legitimate scripts only. Alternatively, developers can blacklist operations by scripts outside of the legitimate domains for the entire web page. According to our observations in Section 4, these approaches are unlikely to incur noticeable performance overhead.

Online Refund Scams. They rely on the ability to edit transaction records on banking web pages in the developer tools. Our system can thwart such scams when only legitimate scripts and their operations on those records are whitelisted in policies.

5.3 Personal Data Governance

Our system benefits both websites and individuals in terms of the personal data governance. By highlighting operating scripts, operations, and the operated data fields defined in policies, we enhance the transparency of the collection and use of personal data on websites, complying with data privacy laws and legislation such as the General Data Protection Regulation [2] and the California Consumer Privacy Act [1]. Individuals are then empowered to make informed decisions about disclosing their data.

6 USABILITY AND COMPATIBILITY

We also develop a tool to automatically generate permission policies for websites and demonstrate the system's compatibility and usability with the policies generated for 30 popular websites in a three-month study.

6.1 Policy Generator

When loading a web page or operating on a page in our tool, it intercepts the use of JavaScript APIs in Table 1 for selected HTML and data operations. During the interception, it iterates through the V8 engine's call stack, identifies the invoked JavaScript functions, and extracts the URL or source code of the executed scripts. It also records the constructor script of every dynamic inline script.

For an operated HTML element, the tool creates a policy with a unique `pid` and includes the involved scripts (and functions) in the policy, according to the syntax in Sections 2.2 and 3.3. It uses script URL, if possible, in the policy. It assigns a unique `sid` to every involved hard-coded inline script and uses that `sid` in the policy. For each involved dynamic inline script, it assigns a unique `sid` to the constructor script and uses that `sid`. The policy generation for data operations is in the same manner. Our tool reuses policies to reduce the maintenance cost. For signing into amazon.com, the tool generated a policy with a chain of three involved scripts, as shown below, for reading the password field at the script level.

```
/** in policy file **/
"JG8MG": {
  "r": { "sids": ["https://images-na.ssl-images-amazon.
       com/images/I/8135BpGZX3L...", "https://images-na.
       ssl-images-amazon.com/images/I/21ZMwVh4T0L...", "
       https://images-na.ssl-images-amazon.com/images/I
       /61yXDIPmT-L..."], "eids": "-" } }
```

6.2 Policy Generation and Validation

Index Pages. We generated policies for every HTML and data operation by JavaScript on the index page of 30 popular websites, as shown in Appendix C. We then deployed the generated policies on the retrieved index pages with mitmproxy [2]. When those pages were fully loaded in our system, we visually compared them with the genuine ones rendered in the Google Chrome browser. We also inspected and compared the cookies, local storage, and outgoing network traffic.

Our system nor the generated policies incurred any exception, demonstrating the system's compatibility and usability. As it is unrealistic to maintain the huge number of policies on some websites (e.g., yahoo.com, cnn.com, and aol.com), we suggest developers to focus on operations on essential web elements and data to reduce the potential performance overhead and maintenance cost.

Sign-in Pages. To generate policies for password access on the sign-in page of those 30 websites, we first identified the password fields, where the `type` attribute is set to 'password' or the `autocomplete`

[2] https://mitmproxy.org/

attribute is set to 'new-password' or 'current-password'. We then generated policies on 27 websites, as shown in Appendix C. No policy was generated on github.com, salesforce.com, and sourceforge.net since no script was involved in their signing-in process. We manually signed into those 27 websites with the deployed policies and no exception occurred.

Longitudinal Study. We re-validated the generated policies on those web pages three months later to assess the maintenance cost of the deployed policies over time. We did not observe any exception nor any new HTML or data operation. We also re-generated policies for those web pages and found that none of the previous policies needs to be updated. The experiments show that polices can be automatically generated and are usable within a certain period of time even on frequently updated news and shopping websites.

7 DISCUSSION

The permission system has not been implemented outside the Blink engine of the Chromium browser. We have not yet enforced permissions for cascading style sheet transitions and animations, that can be used to alter the appearance and behavior of web elements. We leave them for future work.

We will continue improving the JavaScript API list of Table 1. We will validate the feasibility of reading and writing HTML elements with inoperable descendants, as discussed in Section 2.3, and the possibility of permission enforcement on JavaScript statements, as discussed in Section 3.3. We will also optimize the implementation for a better performance.

To use the policy generator introduced in Section 6.1, we require developers to label sensitive HTML and data operations. The automatic and comprehensive identification of those operations remains an open question.

Lastly, we plan to conduct a study involving both developers and users, and seek valuable feedback for continuous improvements.

8 RELATED WORK

In this section, we introduce related work and other HTML features and compare our system with them.

8.1 JavaScript Confinement

Solutions have been proposed to prevent undesired data access and flow on web pages by isolating JavaScript execution [10, 16–18, 21, 22], isolating web elements [31], and enforcing data flows [4, 7, 8, 27]. Meanwhile, [13] aims to restrict JavaScript execution instead of the individual access to web resources. Researchers have also studied the automatic generation of mock web APIs to replace web resources [25]. Unlike these function-centric solutions to protect data, our approach is object-oriented to protect both HTML and data. It does not suffer from the increased system complexity and the significant performance overhead.

Researchers have proposed to sanitize network traffic [6, 24, 26] to prevent data leakage. Unlike them, our approach does not rely on the knowledge of legitimate or suspicious data.

Researchers have also worked on a policy-based approach to prevent DOM tampering via Cross-Site Scripting [11] and conducted type-checking to stop undesired DOM mutations [14]. Unlike [11], our system works at the script and even function level instead of the domain level. It protects a broader range of web resources and does not require the knowledge of data types on web, compared to [14]. [12] used protection rings to coarsely restrict scripts' access to web resources but failed to further differentiate principals and resources in the same ring. [30] aims to confine JavaScript behavior in Node.js but not browsers.

A permission system has been proposed to restrict JavaScript operations on HTML documents [32]. Different from it, we do not trust first-party scripts as they can be altered by third parties [29, 35]. We do not trust browser extensions either; a typical example is the content injection discussed in Section 5.2. To mitigate the vulnerability associated with the call chain discussed in Section 3.3, our system requires all scripts (and functions) on the call stack to have permissions to perform operations, inevitably increasing the complexity and performance overhead. Last but not least, we enforced permissions on a boarder range of JavaScript interfaces including but not limited to the network, storage, cookie-related ones and those in the developer tools. A similar work, [37], aims to generate security policies for JavaScript but it was difficult to deploy due to the significant runtime overhead.

8.2 Other HTML Features

The HTML `readonly` attribute renders input and textarea elements non-editable to users but it does not prevent the inputs from the access by JavaScript or in the developer tools. The `isTrusted` property distinguishes between user-driven and scripted events, returning true for user-initiated actions or false for script-triggered events. We work on a different research problem.

MutationObserver allows developers to attach observers to HTML elements and to monitor mutations on the elements. It can be used to restore web content to its original state as a detection & recovery approach. It requires more development efforts, may lead to performance issues, and can capture mutations only when it was set. In contrast, we work on the prevention instead of the detection.

Subresource Integrity (SRI) validates the authenticity of the retrieved subresources such as scripts and stylesheets with checksums. Differently, we work on the behavior rather than checksums of the scripts. SRI suffers from several limitations, while our system does not. (1) It cannot cope well with dynamic scripts. (2) Scripts that failed SRI check will not be executed, disrupting web functions. (3) Any change to subresources require checksum updates, which are often manually done and prone to errors [5].

Content Security Policy aims to prevent cross-site scripting and injection attacks. It blocks the entire scripts from untrusted sources but our system aims to restrict scripts' access to objects on the web.

9 CONCLUSION

We introduced a in-browser, fine-grained, mandatory access control-based, and object-oriented permission system. It enables web developers to confine JavaScript operations on web content and data. It was implemented in the Blink engine of the Chromium browser as a ready-to-use solution. With comprehensive experiments and analyses, we demonstrated its limited performance overhead, the security enhancements it offers, its usability, and its compatibility to websites. Our system is a reasonable and practical solution, bolstering the security and trustworthiness on the internet.

REFERENCES

[1] Which states will consider ccpa-like consumer privacy bills in 2022? https://www.bytebacklaw.com/2022/01/which-states-will-consider-ccpa-like-consumer-privacy-bills-in-2022, 2022.

[2] General data protection regulation (gdpr). https://gdpr-info.eu/, 2023.

[3] Tranco: A research-oriented top sites ranking hardened against manipulation. https://tranco-list.eu/, 2023.

[4] W. Chang and S. Chen. Extensionguard: Towards runtime browser extension information leakage detection. In *Proceedings of the IEEE Conference on Communications and Network Security*, pages 154–162, 2016.

[5] B. Chapuis, O. Omolola, M. Cherubini, M. Humbert, and K. Huguenin. An empirical study of the use of integrity verification mechanisms for web subresources. In *Proceedings of the Web Conference*, page 34–45, 2020.

[6] L. Chen, Y. Zhou, and D. Evans. Redactdom: Preventing sensitive data leaking through embedded scripts (poster). In *Proceedings of the IEEE Security and Privacy*, 2013.

[7] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, page 1687–1700, 2018.

[8] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the ACM Conference on Computer and Communications Security*, page 748–759, 2012.

[9] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi. The chain of implicit trust: An analysis of the web third-party resources loading. In *Proceedings of the Web Conference*, page 2851–2857, 2019.

[10] L. Ingram and M. Walfish. Treehouse: Javascript sandboxes to helpweb developers help themselves. In *Proceedings of the USENIX Conference on Annual Technical Conference*, 2012.

[11] J. Iqbal, R. Kaur, and N. Stakhanova. Polidom: Mitigation of dom-xss by detection and prevention of unauthorized dom tampering. In *Proceedings of the International Conference on Availability, Reliability and Security*, 2019.

[12] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 231–240, 2010.

[13] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International Conference on World Wide Web*, page 601–610, 2007.

[14] Z. Jin, S. Chen, Y. Chen, H. Duan, J. Chen, and J. Wu. A security study about electron applications and a programming methodology to tame dom functionalities. 2023.

[15] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the USENIX Security Symposium*, pages 641–654, 2014.

[16] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the USENIX Conference on Security*, 2010.

[17] W. Luo, X. Ding, P. Wu, X. Zhang, Q. Shen, and Z. Wu. Scriptchecker: To tame third-party script execution with task capabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2022.

[18] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 481–496, 2010.

[19] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security*, page 736–747, 2012.

[20] N. Pantelaios, N. Nikiforakis, and A. Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, page 477–491, 2020.

[21] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards fine-grained access control in javascript contexts. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 720–729, 2011.

[22] G. Richards, C. Hammer, F. Zappa Nardelli, S. Jagannathan, and J. Vitek. Flexible access control for javascript. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, page 305–322, 2013.

[23] J. C. S. Santos, A. Sejfia, T. Corrello, S. Gadenkanahalli, and M. Mirakhorli. Achilles' heel of plug-and-play software architectures: A grounded theory based approach. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 671–682, 2019.

[24] A. Senol, G. Acar, M. Humbert, and F. Z. Borgesius. Leaky forms: A study of email and password exfiltration before form submission. In *Proceedings of the USENIX Security Symposium*, pages 1813–1830, 2022.

[25] M. Smith, P. Snyder, B. Livshits, and D. Stefan. Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, page 2844–2857, 2021.

[26] O. Starov, P. Gill, and N. Nikiforakis. Are you sure you want to contact us? quantifying the leakage of pii via website contact forms. pages 20–33, 2016.

[27] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining javascript with cowl. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, page 131–146, 2014.

[28] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. Mccoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 151–167, 2015.

[29] T. Tran, R. Pelizzi, and R. Sekar. Jate: Transparent and efficient javascript confinement. In *Proceedings of the Annual Computer Security Applications Conference*, page 151–160, 2015.

[30] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel. Preventing dynamic library compromise on node.js via rwx-based privilege reduction. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, page 1821–1838, 2021.

[31] X. Wang, Y. Du, C. Wang, Q. Wang, and L. Fang. Webenclave: Protect web secrets from browser extensions with software enclave. *IEEE Transactions on Dependable and Secure Computing*, 19(5):3055–3070, 2022.

[32] Z. Wang, W. Meng, and M. R. Lyu. Fine-grained data-centric content protection policy for web applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, page 2845–2859, 2023.

[33] X. Xing, W. Meng, U. Weinsberg, A. Sheth, B. Lee, R. Perdisci, and W. Lee. Unraveling the relationship between ad-injecting browser extensions and malvertising. In *Proceedings of the International World Wide Web Conference*, 2015.

[34] C. Yue and H. Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the International Conference on World Wide Web*, page 961–970, 2009.

[35] M. Zhang and W. Meng. Detecting and understanding javascript global identifier conflicts on the web. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 38–49, 2020.

[36] R. Zhao, C. Yue, and Q. Yi. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proceedings of the International Conference on World Wide Web*, page 1384–1394, 2015.

[37] Y. Zhou and D. Evans. Understanding and monitoring embedded web scripts. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 850–865, 2015.

A POLICY RULE EXAMPLES

Rule 2. In the code below, the <td> element inherits the policy "read_table" from its ancestor <table> and the identifier "s" is created for an inline script. Consequently, the <td> element can be read by that inline script only.

```
/** in policy file **/
"read_table": {
  "r": { "sids": ["s"], "eids": "-" } }
/** HTML code snippet **/
<table pid='read_table'> <tr>
  <td id='td'>...</td>
</tr> </table>
<script sid='s'>
  document.getElementById('td').innerText;
</script>
```

Rule 3. In the code below, the first <td> element's own policy "read_td" overrides its ancestor <table>'s policy "read_table"; consequently, it can be read by the script "s2" only. The second <td> element is assigned its ancestor's policy in addition to its own, thus can additionally be read by the script "s1".

```
/** in policy file **/
"read_table": {
  "r": { "sids": ["s1"], "eids": "-" } },
"read_td": {
  "r": { "sids": ["s2"], "eids": "-" } }
/** HTML code snippet **/
<table pid='read_table'> <tr>
  <td pid='read_td'> </td>
  <td pid='read_table,read_td'> </td>
</tr> </table>
```

Rule 4. The first code snippet below shows that the <table> element's outer HTML cannot be read by the script "s" because no operation by the script "s" on the descendant <td> element is allowed. In the next code snippet, the <table> element's outer HTML can be read by the script "s" only, since its descendant <td> element is read-only to the script "s".

```
/** in policy file **/
"read_table": {
  "r": { "sids": ["s"], "eids": "-" } },
"no_read_td": {
  "-": { "sids": ["s"], "eids": "-" } }
/** 1st HTML code snippet **/
<table pid='read_table'> <tr>
  <td pid='no_read_td'> </td>
</tr> </table>

/** in policy file **/
"all_table": {
  "*": { "sids": ["*"], "eids": "-" } },
"read_td": {
  "r": { "sids": ["s"], "eids": "-" } }
/** 2nd HTML code snippet **/
<table pid='all_table'> <tr>
  <td pid='read_td'> </td>
</tr> </table>
```

Rule 5. In the first code snippet below, the form can be filled out, except for the password field, and submitted programmatically by the script "s" only. Notably, the policy "no_fill_pwd" on the password field does not disable the form submission. In the next code snippet, the form and its fields are inoperable to scripts thus it can only be submitted manually.

```
/** in policy file **/
"fill_form": {
  "w": { "sids": ["s"], "eids": "-" } },
"no_fill_pwd": {
  "-": { "sids": "*", "eids": "*" } }
/** 1st HTML code snippet **/
<form pid='fill_form'>
  Username:<input type='text' name='usr'/>
  Password:<input type='password' name='pwd' pid='
      no_fill_pwd'/>
</form>

/** in policy file **/
"no_fill": {
  "-": { "sids": "*", "eids": "*" } }
/** 2nd HTML code snippet **/
<form pid='no_fill'>
  Username:<input type='text' name='usr'/>
  Password:<input type='password' name='pwd'/>
</form>
```

Rule 6. The policies, as shown below, permit data dispatches to www.google.com and cookies by the script "s". However, they block the last two lines of code due to the domain mismatch and no data flush to the HTML document stream.

```
/** in policy file **/
"www.google.com": {
  "w": { "sids": ["s"], "eids": "-" } },
"cookie": {
  "w": { "sids": ["s"], "eids": "-" } },
"stream": {
  "-": { "sids": ["s"], "eids": "-" } }
/** HTML code snippet **/
<script sid='s'>
  // allowed
  window.postMessage(data, 'www.google.com');
  document.cookie = 'id=' + data;
  // denied
  img.src = 'www.tracking.com/img?id=' + data;
  document.write(data);
</script>
```

B THE CHAIN OF SCRIPTS AND FUNCTIONS FOR CODE EXAMPLE IN SECTION 5.2

```
[3294:1:0207/102315.678761:INFO:element.cc(6576)] Element
    ::canThisWrite--1707322995678--"INPUT"--"null"--"
    text"
[3294:1:0207/102315.706451:INFO:element.cc(6607)]
    1707322995678--"http://10.66.131.145:8080/jquery
    -3.7.1.js"--973c35bf97e8f87d23608e46d2530cc7--"<
    anonymous>"--8381:15
[3294:1:0207/102315.717424:INFO:element.cc(6607)]
    1707322995678--"http://10.66.131.145:8080/jquery
    -3.7.1.js"--973c35bf97e8f87d23608e46d2530cc7--"each"
    --382:18
[3294:1:0207/102315.726612:INFO:element.cc(6607)]
    1707322995678--"http://10.66.131.145:8080/jquery
    -3.7.1.js"--973c35bf97e8f87d23608e46d2530cc7--"each"
    --204:16
[3294:1:0207/102315.735682:INFO:element.cc(6607)]
    1707322995678--"http://10.66.131.145:8080/jquery
    -3.7.1.js"--973c35bf97e8f87d23608e46d2530cc7--"val"
    --8351:14
```

```
[3294:1:0207/102315.735846:INFO:element.cc(6607)]
    1707322995678--"http://10.66.131.145:8080/index2.
    html"--db6b170ad3c3d5feb7c24b8b4444d7b4--"myFunc"
    --19:20
```

C STATISTICS OF THE GENERATED POLICIES

	Website	Index Page		Sign-in Page	
		Policy	Involved Scripts All / Dynamic	Policy	Involved Scripts All / Dynamic
1	google.com	8	8 / 2	4	5 / 0
2	amazonaws.com	21	18 / 2	3	4 / 0
3	facebook.com	13	14 / 0	2	2 / 0
11	twitter.com	16	14 / 1	9	6 / 0
14	instagram.com	7	7 / 0	2	2 / 0
17	linkedin.com	14	13 / 0	1	1 / 0
21	netflix.com	11	9 / 1	2	2 / 1
27	amazon.com	98	65 / 3	6	6 / 2
41	yahoo.com	187	105 / 20	1	1 / 0
48	github.com	30	22 / 0	0	0 / 0
57	spotify.com	10	8 / 1	3	2 / 1
64	zoom.us	78	61 / 11	4	4 / 1
90	reddit.com	8	8 / 0	2	2 / 0
104	comcast.net	8	8 / 0	3	2 / 0
105	dropbox.com	36	31 / 22	6	5 / 5
110	baidu.com	53	32 / 7	5	5 / 1
117	flickr.com	34	25 / 3	2	2 / 0
119	nytimes.com	33	28 / 13	2	2 / 0
150	paypal.com	19	14 / 1	6	5 / 0
156	cnn.com	130	86 / 30	2	2 / 0
176	salesforce.com	6	6 / 1	0	0 / 0
179	ebay.com	77	62 / 23	4	4 / 0
184	wellsfargo.com	48	23 / 3	7	5 / 1
202	kaspersky.com	31	20 / 2	3	4 / 1
205	sourceforge.net	29	27 / 3	0	0 / 0
212	discord.com	11	9 / 1	2	2 / 0
215	slack.com	64	55 / 9	2	2 / 0
216	shopify.com	28	24 / 5	3	3 / 0
244	researchgate.net	50	34 / 17	4	7 / 2
282	aol.com	243	110 / 27	1	1 / 0

Table 5: The numbers of generated policies, all involved scripts, and involved dynamic scripts for the index pages and sign-in pages of 30 popular websites.

D PERFORMANCE DETAIL UNDER THE EXTREME CONDITION ON TOP 50 WEBSITES

These columns are the website, number of executed scripts, numbers of intercepted JavaScript read and write operations, number of iterated stack frames, and number of dynamic script constructions when visiting the index pages. The last three columns are the execution time of all the scripts with and without our permission system, and the system's performance overhead. Numbers less than 0.00 in the last column indicate the possible browser performance fluctuation.

	Website	Scripts	Reads		Writes		Stack Frames	Dynamic Script Construction	Time (ms)	$\text{Time}_{\text{baseline}}$ (ms)	Incurred Overhead (x)
			$P_{\perp}$	$P^{\intercal}$	$P_{\perp}$	$P^{\intercal}$					
1	google.com	12	34	6	109	30	1,899	5	2,018	1,456	0.39
2	amazonaws.com	21	423	142	748	272	53,016	12	4,802	3,053	0.57
3	facebook.com	33	46	5	66	9	9,496	0	585	470	0.24
4	microsoft.com	86	432	199	2,550	1,367	51,511	27	2,993	2,866	0.04
5	googleapis.com	0	12	4	0	0	0	0	0	0	0.00
6	apple.com	12	277	242	2,101	1,177	165,496	0	1,781	789	1.26
7	youtube.com	44	111	85	6,759	3,655	635,527	1	4,384	5,511	-0.20
8	a-msedge.net	8	12	10	18	10	137	0	108	72	0.50
9	akamai.net	8	12	10	18	10	137	0	86	80	0.07
10	akamaiedge.net	8	12	10	18	10	149	0	94	125	-0.25
11	twitter.com	58	38	5	517	194	29,181	44	5,242	5,850	-0.10
12	azure.com	92	8,955	116	1,476	867	72,677	47	14,815	9,558	0.55
13	googlevideo.com	15	26	6	58	11	1,205	8	714	687	0.04
14	instagram.com	24	445	270	536	251	156,940	0	257	261	-0.02
15	gstatic.com	0	12	4	0	0	0	0	0	0	0.00
16	cloudflare.com	93	366	174	4,294	1,782	261,908	78	11,219	6,694	0.68
17	linkedin.com	16	63	6	536	272	7,116	7	1,787	1,790	0.00
18	tiktokcdn.com	8	12	10	18	10	149	0	89	81	0.10
19	live.com	78	1,199	65	2,371	1,092	36,321	35	1,546	1,513	0.02
20	doubleclick.net	13	1,570	2	513	279	102,105	4	3,992	2,148	0.86
21	netflix.com	24	280	131	672	293	21,528	3	3,859	3,497	0.10
22	office.com	21	62	31	156	63	9,283	2	2,424	2,031	0.19
23	windowsupdate.com	8	12	10	18	10	137	0	173	143	0.21
24	akadns.net	8	12	10	18	10	137	0	89	82	0.09
25	googletagmanager.com	0	12	4	0	0	0	0	0	0	0.00
26	apple-dns.net	8	12	10	18	10	149	0	113	102	0.11
27	amazon.com	199	1,993	404	1,708	1,207	167,207	53	3,520	2,359	0.49
28	trafficmanager.net	8	12	10	18	10	137	0	98	96	0.02
29	fbcdn.net	33	33	5	64	9	9,496	0	656	380	0.73
30	gtld-servers.net	8	12	10	18	10	137	0	137	130	0.05
31	wikipedia.org	4	373	5	29	13	2,346	0	182	167	0.09
32	domaincontrol.com	0	12	4	0	0	0	0	0	0	0.00
33	icloud.com	12	65	13	282	151	13,022	3	978	954	0.03
34	fastly.net	111	594	238	1,927	543	453,221	117	5,616	4,551	0.23
35	bing.com	0	0	0	0	0	0	0	0	0	0.00
36	microsoftonline.com	8	12	10	18	10	137	0	158	158	0.00
37	googleusercontent.com	0	12	4	0	0	0	0	0	0	0.00
38	wordpress.org	17	459	37	82	41	5,896	3	2,100	1,114	0.89
39	root-servers.net	8	12	10	18	10	137	0	91	78	0.17
40	mail.ru	64	313	62	6,577	3,127	775,543	36	11,741	10,228	0.15
41	yahoo.com	86	1,119	948	540	299	33,257	8	6,665	6,396	0.04
42	youtu.be	44	117	91	6,760	3,668	645,662	1	5,054	6,355	-0.20
43	aaplimg.com	8	12	10	18	10	137	0	90	90	0.00
44	digicert.com	68	1,485	1,205	746	406	64,499	46	9,632	9,158	0.05
45	pinterest.com	0	0	0	0	0	0	0	0	0	0.00
46	ui.com	29	780	462	1,103	561	109,802	20	6,697	6,813	-0.02
47	l-msedge.net	8	13	11	20	11	149	0	116	94	0.23
48	github.com	57	82	38	329	284	711	17	1,976	2,441	-0.19
49	windows.net	24	1,726	89	563	319	26,963	6	1,184	1,115	0.06
50	tiktokv.com	0	8	4	0	0	0	0	0	0	1.00

Table 6: Performance detail under the extreme condition on top 50 websites.