JSISOLATE: Lightweight In-Browser JavaScript Isolation

Mingxue Zhang
Chinese University of Hong Kong
Hong Kong SAR, China
mxzhang@cse.cuhk.edu.hk

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

ABSTRACT

Modern web applications commonly include third-party scripts from external hosts. While enabling code reuse and enhancing the functionalities, the reliability of client-side JavaScript code can be impaired by the inclusion of other scripts. Since all scripts run in the same execution environment in the browser, executing them all together may cause unexpected effects. For example, global variables with the same name might be defined by multiple scripts, causing the actual value to be unpredictable.

In this paper, we design a lightweight browser-based framework, JSIsolate, that provides an isolated and reliable JavaScript execution environment. JSIsolate injects scripts into different isolated environments based on their dependency relationship. In this way, it executes scripts with independent functionalities in different contexts, effectively preventing them from interfering with each other. We further evaluated the compatibility and performance overhead of JSIsolate on Alexa top 1K websites, and showed that it can efficiently isolate scripts while preserving the functionalities.

CCS CONCEPTS

• **Software and its engineering** → **Software organization and properties**; **Software reliability**.

KEYWORDS

JavaScript; Script isolation; Web browser

ACM Reference Format:
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 '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468577

1 INTRODUCTION

It is a common practice in developing web applications to include scripts from different hosts. A recent study of Alexa top 75K websites shows that external scripts are prevalent, with a median number of 9 and the maximum number of 202 per site [15]. Over 90% of the websites include at least one external script. Including scripts from third-party hosts allows a developer to reuse the code in third-party libraries. For example, a developer can include a third-party jQuery script to facilitate DOM traversing and manipulation.

Unfortunately, including multiple external scripts together may cause undesired effects. In the browser, all the scripts in the same frame (*e.g.*, the main frame) run in a shared environment. This means any script can interfere with the execution of other scripts in the same frame. Existing works have revealed that name conflicts could exist between JavaScript libraries and any normal scripts, and are prevalent on the web [24, 44, 45]. Builtin methods and properties could also be overriden and affect the default behaviors of all scripts [2]. Meanwhile, some sensitive data could be stored at the client side and can be accessed and modified by any included script. Executing scripts from different parties in the same context could significantly affect the functionalities hence the reliability of web applications.

Prior works have investigated the problem of protecting JavaScript code integrity by isolating the namespace, *i.e.*, execution environment, of scripts. In [10, 13, 19, 33, 42], the authors sandboxed untrusted third-party scripts in iframes, preventing them from accessing objects in the main frame. However, as the iframes use a separate DOM from the embedding page, special care must be taken to handle the rendering of contents generated by the sandboxed scripts. Also, the events from the embedding page must be specifically forwarded to the iframes. They therefore introduced a high latency in page rendering. Another branch of research works emulated an isolated execution environment by restricting the accesses to objects defined by first-party scripts from untrusted scripts [3, 23]. They rely on a runtime access monitor, which incurs a significant overhead on the basic operations like function invocation.

In this paper, we aim to design a lightweight mechanism that provides an isolated JavaScript execution environment in the browser, to prevent functionality interference in mashups. We face the following challenges. First, scripts included on the same page may be functionally dependent. We need to carefully determine the execution environment for each script to maintain the functionalities. Second, JavaScript code can be dynamically included in various ways, for example, through JavaScript URLs, `eval()`, or asynchronously invoked as event listeners. The event listeners and JavaScript URLs can be defined as attributes of DOM elements, which can also be dynamically modified by JavaScript. It is non-trivial to capture all the dynamically included scripts and isolate them in an appropriate environment. Finally, in order to make our scheme deployable and practical, we need to minimize the performance overhead and the required modification to applications, which is challenging.

To overcome the above challenges, we develop JSIsolate, a lightweight browser-based isolated JavaScript execution framework for improving script reliabilities. It operates in two modes: a dynamic script-dependency analysis mode, and a policy enforcement mode

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8562-6/21/08...$15.00
https://doi.org/10.1145/3468264.3468577

that separates JavaScript in isolated execution environments (contexts) to provide higher reliability.

In the first mode, JSIsolate analyzes the dependency relationship between scripts. It first uses an open-source domain relationship list as the ground truth of script dependencies. It then dynamically mediates all script accesses to JavaScript objects to infer the dependency between scripts not on the list. Based on the dependency analysis, it automatically generates policies that specify the execution context for each *statically included* third-party script. It also generates auxiliary information to help the developers validate the isolation policies. JSIsolate can generate two types of isolation policies: domain-level policies, where scripts of the same domain are assigned the same context for easy management; and URL-level policies, where each script identified by its URL is assigned a context for fine-grained isolation.

In the second mode, for each group of functionality-dependent scripts, JSIsolate spawns an isolated execution environment—the *isolated world*, which has been used to confine content scripts of browser extensions. It also tracks the dynamic inclusion of JavaScript code to identify the initiator script of each *dynamically included* script, which would be executed in the same context as the initiator script. This prevents an isolated script from escaping its execution environment by dynamically injecting new scripts. The isolation is seamlessly and securely performed without the overhead of intercepting each JS object access.

We implemented a prototype of JSIsolate based on the Chromium browser version 71, and tested it on Alexa top 1K websites. We will release the prototype as an open-source software. We demonstrate with real world and synthetic examples that JSIsolate is able to separate functionality-independent JavaScript code. We further show JSIsolate can isolate scripts while preserving the intended functionalities of real world websites–using URL-level (resp. domain-level) policies, script isolation caused exceptions on 1 (resp. 0) out of the top 100 websites. JSIsolate also introduces limited performance overhead. Using URL-level policies, we observed a 1.91% average increase in memory consumption and a 7.95% average slowdown on page loading. When using domain-level policies, the average memory and page loading overhead is 1.34% and 6.66%, respectively. The results show that JSIsolate can *effectively* and *efficiently* provide a reliable execution environment for client-side JavaScript.

In summary, we make the following contributions.

- We design and develop JSIsolate, a light-weight isolated JavaScript execution framework for improving the reliability of client-side JavaScript.
- We systematically analyze the dependency relationship between scripts on the Alexa top 1K websites. The analysis results show the feasibility of isolating scripts in separate execution environments.
- We evaluate the effectiveness, compatibility and the performance overhead of JSIsolate on real world websites, and demonstrate the practicality of our design.

2 OVERVIEW

In this section, we first introduce the background knowledge (§2.1), we then describe the motivating examples of our design (§2.2), and finally discuss the research challenges (§2.3).

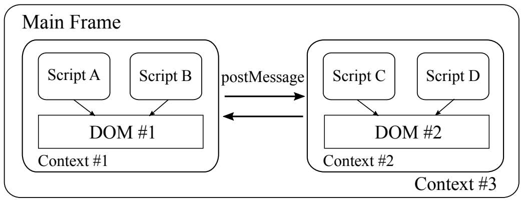

Figure 1: A simplified browser architecture.

2.1 Background

JavaScript Execution Environment in the Browser. The high-level architecture of a web browser is depicted in Figure 1. In modern web browsers, a browsing context is created for each opened tab, and can be nested using iframes [39]. Each frame (the main frame or an iframe) has its own window object, and the cross-window communication is usually accomplished via the `postMessage` interface. The browsing context encloses the web content to be displayed to the users. It also includes a JavaScript execution environment, in which JavaScript code is interpreted and executed on a JavaScript engine. Scripts included in the same frame run in a single execution context. As a result, a script can access any global object defined by another script that resides in the same frame.

Security Policies in the Browser. The Same-origin Policy (SOP) [40] specifies that a frame in one origin should not be allowed to access the resources in a different origin, which is recognized as a tuple of scheme (*e.g.*, `https://`), host name (*e.g.*, `xyz.com`) and port number (*e.g.*, `443`). It prevents cross-frame script access, but still permits intra-frame script access or interference. The Cross-origin resource sharing (CORS) relaxes the SOP by allowing a restricted set of resources from one origin to be accessed by another origin [38]. It enables the developers to embed cross-origin resources, *e.g.*, images and videos, in a more flexible way.

The Content Security Policy (CSP) allows developers to specify the origins of resources that are permitted to be loaded on a website [37]. It is designed to mitigate the cross-site scripting (XSS) attacks and other content injection attacks. However, it has suffered from misconfiguration of developers and its static nature, and therefore receives only a low adoption rate on the web [5, 6]. Compared with CSP, script isolation provides a more flexible approach to limiting privileges of scripts to interfere with each other.

The Subresource Integrity (SRI) [41] was introduced as a W3C recommendation to protect data integrity during the network transmission. A developer can specify an `integrity` attribute, which is a cryptographic hash value, with an external resource such as a script or a style sheet. The browser, upon fetching the resource, would then validate the hash value to ensure the resource has not been tampered with. However, the SRI cannot protect JavaScript objects from being overwritten by other scripts already running in the same frame.

2.2 Motivating Examples

Global Identifier Conflicts. The use of a shared namespace grants scripts the privilege to redefine global objects (*i.e.*, functions or variables) defined by other scripts. Such an overwrite could result

```
/* ocanvas.js */
(function(a, b, c) { a.logs = []; //array })(window, document);

/* aframe.js */
window.logs = function(e) {...}; // function

/* client.js */
logs.push('log'); // runtime TypeError
```

Listing 1: An example of conflicting variable definitions between JavaScript libraries.

```
/* a.js */
RegExp.prototype.test = function() { return true; }

/* client.js */
if((/https?:/i).test(...)) { // security check evaded }
```

Listing 2: Polluting JavaScript prototype to bypass security checks.

in disruptive behaviors, consequently impairing the reliability of web applications. One such case is discussed in [24], where two JavaScript libraries define `window.logs` with values of different types. As shown in Listing 1, when these libraries are included in a specific order, a runtime exception would be thrown because of inconsistent types. Similar examples are also presented in [44], *e.g.*, the global function `addHTML()` is defined differently by two scripts, which causes different HTML contents to be injected.

Prototype Poisoning Attacks. Each JavaScript object has an associated prototype object, from which it inherits its properties and methods. JavaScript prototype allows developers to easily define properties and methods on all the objects initiated from the same constructor. However, it can also be overwritten and cause undesired effects. One such vulnerability was reported on MyVidoop, which is a bookmarklet-based password manager [2]. As shown in Listing 2, MyVidoop uses a simple regular expression `/https?:/i` to check if a URL starts with 'http' or 'https'. However, the regular expression is evaluated at the client side, and therefore is subject to prototype poisoning attacks. Once the native function `RegExp.prototype.test()` is overriden as shown, the security check would be bypassed and make the bookmarklet susceptible to XSS attacks. This demonstrates the unexpected interference between first-party and third-party code makes it hard for web applications to reliably perform the intended functionalities.

The root cause of the above problems is the use of a single context (*i.e.*, namespace) in the browser. Any (not necessarily malicious) script can potentially interfere with the execution of all other scripts executed in the same context. This indicates the need of an isolation mechanism that runs scripts with independent functionalities in separate contexts, such that scripts cannot interfere with the objects and code in the other execution environments.

2.3 Research Challenges

We face the following challenges in isolating client-side scripts.

Context Assignment. Scripts included in the same frame usually depend on each other to accomplish their tasks. For example, a script may call jQuery methods to facilitate DOM manipulation and event handling. It is non-trivial to determine the context in which a script should be executed. Separating scripts depending on each other would break the functionality of a website. In particular, some scripts, *e.g.*, library scripts, might be required for multiple functionalities.We need a way to precisely assign scripts to appropriate contexts. Further, JavaScript code can be dynamically loaded. It is insufficient to statically determine the context for each script.

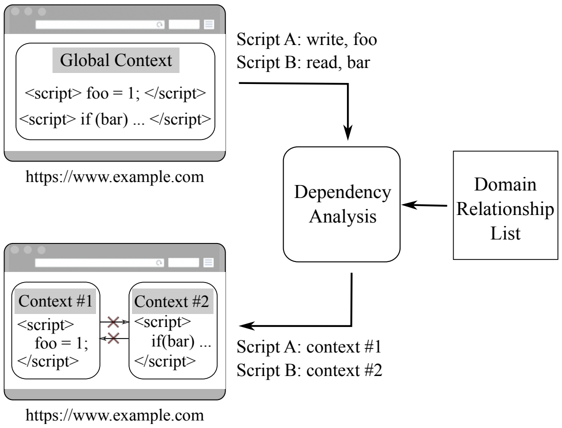

Figure 2: Overview of JSIsolate.

Coverage. There are many ways to execute JavaScript code in the browser. Except for the inline and external scripts included via `<script>` tags, JavaScript code can also be loaded through JavaScript URLs, inline event handlers and the `eval()` function, *etc.* For example, when a user clicks on an anchor element <a src='javascript:alert("clicked")'>, the method `window.alert()` will be invoked. In order to capture and isolate all the code executed in the browser, we need to cover all the above cases, which is difficult. JavaScript URLs can be defined with various types of DOM elements, which could also be dynamically created by other scripts. Similarly, the inline event handlers can be registered and modified by multiple scripts. Therefore, we need to precisely track the creation and modification of any JavaScript code.

Deployability. Many existing works (*e.g.*, [19, 33, 42]) sandbox scripts in iframes to provide an isolated execution environment. These mechanisms require many modifications to the original page, and extra efforts must be made to handle the cross-frame interaction, especially on event forwarding and content rendering. We aim to develop a system that requires minimal changes on the web pages and minimizes the performance overhead to lower the deployment difficulty, which is challenging.

3 DESIGN AND METHODOLOGY

In this section, we present JSIsolate, a browser-based framework that isolates scripts in separate execution contexts. We consider in our design two execution contexts: a first-party context for scripts serving first-party functionalities, and a third-party context for scripts providing other independent functionalities. Nevertheless, more contexts can be created for providing a finer-grained isolation mechanism when feasible. We assume that scripts from different entities are not related and shall be separated. The overview of JSIsolate is depicted in Figure 2.

JSIsolate determines the execution contexts of scripts based on their dependency relationship. A domain relationship list that groups domains belonging to the same entity, is used as the ground truth. It is maintained by Disconnect [25], which is a tracker blocking application. For instance, facebook.com and fb.me are both in group "Facebook". JSIsolate considers two situations when handling context assignment for a script. First, both the script domain and the website domain are on this list. If the script and the website are in the same group, it is executed in the first-party context; else if it is statically included, it is isolated in the third-party context to prevent cross-entity interference; otherwise, it is assigned to the context of its initiator script. Second, the website/script domain is not on the list, *i.e.*, the ground truth is unavailable. JSIsolate analyzes the object dependency between a *statically included* third-party script with the first-party scripts and assigns the context accordingly; and executes a *dynamically included* third-party script in the same context as the initiator script. With such a strategy, JSIsolate allows the dependent scripts (including the dynamically loaded ones) to function correctly, while prohibiting cross-context interference through dynamic script inclusion.

In the following, we describe our design using the Chromium browser as an example. Specifically, we show how JSIsolate records JavaScript object accesses to analyze dependency (§3.1), and how it tracks the dynamic inclusion of a script (§3.2). We then describe our methodology to generate the isolation policies (§3.3). Finally, we demonstrate how JSIsolate creates separate execution contexts based on the policies (§3.4).

3.1 Recording JavaScript Object Accesses

Reading and writing objects (*i.e.*, functions and variables) defined in each other is the most explicit evidence of the dependency relationship between scripts. For instance, a script relying on a cookie-management library script to add/remove cookies would usually call (*i.e.*, read) cookie-related functions defined in the library script. JSIsolate aims to record any access to the objects defined by any script to determine the dependency relationship.

In the Chromium V8 JavaScript engine, every JavaScript object is represented as an instance of the `Object` class, which could have multiple properties. For instance, a global function `foo` is represented as `window.foo`, which is a property of the `window` object. Therefore, to intercept all the accesses to objects, JSIsolate monitors the getter and setter methods of the `Object` properties in V8, from which it can get the names of objects and properties being accessed. However, variables of non-primitive type are copied/passed by reference in JavaScript, thus scripts do not necessarily need to access an object using the same name. For example, `objY = objX` makes variable `objY` an alias of object `objX`, and any script can thereafter access `objX` via `objY`. In particular, the keyword `this` in JavaScript points to different objects when used in different scopes. For example, when accessed in an object method, `this` refers to the owner object, and when used in a normal function scope, it points to the global object `window`. JSIsolate uses the memory address of the accessed object in V8 to uniquely identify an object. As long as an identifier points to the same object, the memory address will be the same, regardless of the variable name being used.

For each write operation to an object property, which is accomplished by invoking the setter method, JSIsolate also records the value to be assigned to the property. Once an access to an object is intercepted, it further inspects the current JavaScript call stack to locate the bottom script as the script that initiates the access. JSIsolate records the `scriptID` of a script as its unique identifier. Note that the declaration of a global function or variable is also represented as a write operation to the global object `window`. Therefore, JSIsolate is able to capture all the declarations and write operations to the objects that might be referenced by other scripts.

For read operations, however, JSIsolate records the read to objects but not the read to their named properties. We make this design choice to mitigate the prototype poisoning attacks. If a script overwrites a prototype object, *e.g.*, by adding new properties or overwriting builtin methods that are later accessed by another script, JSIsolate does not record the read to the overwritten properties/methods; otherwise, the two scripts would be considered as dependent, allowing the prototype to be poisoned. For example, when `owner.property` is being read by script `A`, JSIsolate only logs a read to `owner`. This does not affect our ability in capturing object dependencies between scripts. We will demonstrate the details with our dependency analysis in §3.3.

3.2 Tracking Dynamic Script Inclusion

There are many ways to include JavaScript code in a web page. First, a script can be loaded via `<script>` tags, either as an inline (embedded as part of the HTML code) or an external (loaded from an external JavaScript file) script. The `<script>` tags can also be dynamically created by other scripts via `document.createElement("script")`, *etc.* Second, JavaScript code can be defined as an attribute of a DOM element, *e.g.*, as a JavaScript URL or an event handler. In both cases, the attribute can be modified by JavaScript after creation. In addition, by calling `eval()`, a string will be executed as JavaScript code. We need to handle all the possible cases that JavaScript code can be loaded in a web page.

3.2.1 *HTML Script Elements.*

A script can create new `<script>` elements by calling APIs like `document.createElement("script")` and `document.write("<script src=...></script>")`, or directly setting the `outerHTML` attribute of an existing element to replace it with a script element. Alternatively, a script can also inject new script tags via `innerHTML`, *e.g.*, by replacing the inner content of a <div> element with "`<script>...</script>`".

JSIsolate monitors all the above APIs in the browser to find the script that creates a script element. Each time such an API is called, JSIsolate locates the script that initiates the API call at the bottom JavaScript call stack frame, and records the corresponding `scriptID` as its unique identifier. JSIsolate also intercepts any access to the `outerHTML` and `innerHTML` attributes to track the replacement of existing elements. The `parentScriptID` attribute—that is initialized as `null`—of a dynamically created script element is set as the `scriptID` of the initiator script. In this way, it can attribute the dynamic creation of any `<script>` element to a specific script. This allows us to differentiate between script elements created dynamically and those inserted statically by developers.

3.2.2 *JavaScript URLs.*

In the browser, JavaScript URLs could be defined with multiple types of DOM elements. We list the tagnames and the corresponding attributes in Table 1. These attributes can be statically defined by the web developers, or dynamically defined and modified by scripts. For example, a script can modify the `href`

```
// #1: modifying on-event attribute
a.onclick = function() { console.log('clicked'); }

// #2: calling addEventListener()
a.addEventListener('click', function() {console.log('clicked')
     ;})

// #3: creating elements with an inline event listener
document.write('<a onclick=function(){...}></a>');
```

Listing 3: 3 ways to register event listeners on an element.

attribute of an anchor element in various ways. Firstly, a script sets the `href` attribute of an anchor element `a` to a JavaScript URL by calling `a.setAttribute(‘href’, ‘javascript:url’)`. A script can also assign a new value to `href` through `a.href=‘javascript:url’` or `a.attributes[‘href’]=‘javascript:url’`. Moreover, a script can directly create a new anchor element with a `href` attribute in the same way as it creates script elements.

JSIsolate hooks all the DOM APIs that could be used to modify the attributes in Table 1. Specifically, it assigns an `nid` attribute as a unique ID to each DOM element. Every time an invocation to these APIs is intercepted, it checks if the attribute value is a JavaScript URL. If so, it identifies the `scriptID` of the accessing script, and updates a map from `nid` to `scriptID` to keep track of the script that last modifies the corresponding attribute of a DOM element. The dynamic creation of elements in Table 1 are monitored in the same way as described in §3.2.1. JSIsolate specifically records the `scriptID` as the `initiator` attribute of these DOM elements. When any of these elements is created, JSIsolate records the `nid` and `initiator` in the map if the attribute is a JavaScript URL.

3.2.3 *eval().*

In JavaScript, `eval()` is a commonly used function that evaluates the argument string as JavaScript code. The code generated by `eval()` is granted the same privilege as the caller function, thus can also overwrite other scripts. Fortunately, the code generated by `eval()` is intrinsically executed in the same context as the caller script. Therefore, a script cannot escape the context assigned to it by generating new code via `eval()`.

3.2.4 *JavaScript Event Listeners.*

Instead of calling `eval()` and creating/modifying `<script>` tags and JavaScript URLs, a script can also register an event listener on an HTML element to introduce new JavaScript code. The event listener is a JavaScript code snippet that will be executed when the corresponding event is fired. Specifically, a script can register an event listener by: 1) modifying the on-event attributes of a DOM element; 2) calling the `addEventListener()` API to add a new event listener; or 3) directly creating an element with an inline event listener (see Listing 3). Fortunately, JSIsolate does not need to monitor JavaScript event listeners specifically, because they are registered in the same JavaScript execution environment as the scripts register them in Chromium.

3.3 Generating Isolation Policies

In this section, we describe our method to determine the object dependency between scripts and generate script isolation policies according to the object dependency.

3.3.1 *Object Dependency.*

JSIsolate identifies dependent scripts based on the JavaScript object access logs. There are four possibilities that two scripts can read or write the same object.

- ***Read after write (RAW)***: one script writes an object and another script reads it afterwards.
- ***Read after read (RAR)***: one script reads an object and another script also reads it afterwards.
- ***Write after write (WAW)***: one script writes an object and after that, another script also writes it.
- ***Write after read (WAR)***: one script reads an object and after that, another script writes it.

We consider a script depends on another if we detect a *RAW* condition. The *RAR* and *WAR* conditions, however, do not necessarily suggest direct object dependency, although the scripts are also accessing the same object. For example, suppose script A reads an object after script B reads it. In this case, both scripts directly depend on another script C that defines or declares that object. Further, the *WAW* condition indicates a conflicting write that could impair the code reliability. JSIsolate reports any detected conflicts to the developers to help them adjust the generated script isolation policies when necessary.

Recall that JSIsolate does not record reads to named object properties, which still allows it to capture all dependencies. First, when there is a *RAW* on an object, JSIsolate can capture the dependency as it records all the reads and writes to objects. Second, when there is a *RAW* on a named property (*e.g.*, `A` reads a property written by `B`), both `A` and `B` have to firstly get a reference to the owner object. If the owner object is defined by `B`, there is a *RAW* on the owner object where `A` reads `B`; if the owner object is defined by another script `C`, both `A` and `B` will be considered as dependent on `C` as they both read an object defined by `C`. In a prototype poisoning attack, however, the prototype object is not explicitly defined by any script thus no dependency will be captured. Therefore, the attack can be mitigated as the poisoned prototype can be isolated.

3.3.2 *Context Assignment.*

To determine the context of a script, we first classify scripts into three categories based on how they are loaded in a web page:

- ***First-party scripts***: scripts that are included from the first-party domain, including inline scripts written by the developer;
- ***Static third-party scripts***: scripts that are statically included from a third-party domain;
- ***Dynamic third-party scripts***: third-party scripts that are dynamically included by another script.

We assume that the *first-party scripts* serve the first-party functionalities. We determine the context of a *static third-party script* based on the ground truth domain relationship list and whether its functionality relates to first-party functionalities. The *static third-party scripts* can be further divided into three classes: 1) scripts loaded from domains belonging to the first-party entity; 2) scripts that read or are read by *first-party scripts*; 3) scripts that do not have object dependency with *first-party scripts*. Scripts of the first two classes are functionality-dependent with first-party scripts and are put in the first-party context. The dependency is *transitive*, therefore, a third-class script that has dependency with a script in the first two classes is also executed in the first-party context. The rest statically included scripts shall be executed in a separate context, as their functionalities are independent from the first-party ones.

A static third-party script could intentionally read objects defined by a first-party script to get its way into the first-party context.

Table 1: DOM attributes that could include JavaScript URLs.

Tagname	Attribute	Example	Invoke Condition
anchor	href	`<a href='javascript:alert("anchor clicked")'></a>`	Click
area	href	`<area shape="rect" coords="0,0,82,126" href='javascript:alert("area clicked")'>`	Click
button	formaction	`<button formaction='javascript:alert("button clicked")'>Click</button>`	Click
form	action	`<form action='javascript:alert("form submitted")' method='get'>`	Submit
iframe	src	`<iframe src='javascript:alert("iframe loaded")'></iframe>`	Load
input	formaction	`<input type='submit' formaction='javascript:alert("input clicked")'/>`	Click

Consequently, not only the static script itself but also any dynamic script it includes would be capable to make arbitrary access to the first-party objects. Nevertheless, it is very difficult, if possible, to exclude these intentional accesses without the knowledge of the intended functionalities of each script. Therefore, we rely on the developers to validate and adjust the context assigned to static scripts. To do that, we also summarize the cross-script reads that lead to the context assignments. This helps the developers with the validation, which is described at length in §3.3.3. We will demonstrate in §5.3 that the number of scripts that need a manual inspection is limited, and they make a moderate number of cross-script reads. Therefore, it is practical for the developers to validate the isolation policies to prevent such scripts from being executed in the first-party context.

In order to prevent scripts from escaping its context by *dynamically* injecting other scripts, we execute the *dynamic third-party scripts* in the same context as the script that creates them, unless the dynamic scripts are loaded from a domain that belongs to the first-party entity. In particular, JavaScript URLs are executed in the context where the scripts that last set them are executed.

JSIsolate maintains a `contextid` attribute with every script in a web page, which is a unique ID of the context assigned to the script. By default, the scripts are divided into two groups, with a `contextid` of "1" for scripts implementing first-party functionalities and "3" for the other scripts. In the following, a context with `contextid` of "1" and "3" are called a *first-party context* and *third-party context*, respectively. One special case is library scripts, *e.g.*, jQuery, which might define objects that are read by scripts in both contexts. In order to preserve the functionalities of such library scripts, JSIsolate in particular sets the `contextid` attribute of them as "both". It then loads a copy of these scripts in both the first-party and third-party contexts. Note that the developers can also configure the isolation policies to separate scripts in a finer-grained manner.

An example. Consider the HTML page in Listing 4, which contains an anchor element and three statically included scripts. In the following, we call the scripts A (L4), B (L5) and C (L6), respectively. Script B depends on script A, as B calls function `myFunc` defined by A in line 11. Therefore, A shall be executed in the same context as B, which is a first-party script. Script A also creates another script D in line 12, which will be executed in the same context as A. Script B and C both modify the `href` attribute of the anchor element by assigning different JavaScript URLs to it. As script C is the one that last modifies the attribute, the JavaScript URL shall be executed in the same context as C. Similarly, the click event listener registered by script C in line 22 shall be executed in the same context as C. In summary, the context assigned to each script is as follows.

- First-party context: script A, B and D.
- Third-party context: script C, JavaScript URL in line 20 and event listener in line 22.

```
1  <html>
2    <body>
3      <a id="anchor1">Click Me</a>
4      <script src='https://www.lib-1.com/a.js'></script>
5      <script src='https://www.example.com/b.js'></script>
6      <script src='https://www.lib-2.com/c.js'></script>
7    </body>
8  </html>
9
10 /* a.js */
11 function myFunc() { return {}; }
12 document.write("<script>alert(\"new script\");<\/script>");
13
14 /* b.js */
15 var myObj = myFunc();
16 document.getElementById("anchor1").href = 'javascript:alert("
         second script")';
17
18 /* c.js */
19 var myAnchor = document.getElementById("anchor1");
20 myAnchor.href = 'javascript:alert("third script")';
21 myAnchor.click();
22 myAnchor.onclick = function() {alert('event listener')};
23 myAnchor.click();
```

Listing 4: HTML page https://www.example.com.

Note that we do not generate policies for script D, the JavaScript URL and the event listener in the above example, as they are generated on-the-fly by script A and C. Rather, the dynamic creation of these script code are tracked by JSIsolate, which will determine the context for them according to the initiator scripts.

3.3.3 *Isolation Policies.*

JSIsolate creates an isolation policy for a web page based on the previous dependency analysis. Once finalized, the policy can be sent with the page source HTML code (*e.g.*, like the CSP header) to instruct the browser to enforce the policy. In addition to the context assigned to each script, JSIsolate also includes in the policies necessary auxiliary information, which helps the developers understand and adjust the policies. In the following, we describe the auxiliary information we report in the policies.

To facilitate the interpretation and enforcement of isolation policies, for each statically included third-party script, JSIsolate reports in the policy its script source URL and the corresponding `contextid`. At enforcement time, JSIsolate will then determine the context for each third-party script by matching the URL. However, the script source URL might contain random strings, which could change frequently. For example, website https://chaturbate.com includes multiple scripts from ssl-ccstatic.highwebmedia.com, which have almost the same URLs except for a random suffix. In order to fix this, JSIsolate also performs an approximate match on script URLs, which we describe in §4. Meanwhile, as statically included inline scripts are all by default considered as first-party scripts, JSIsolate does not include them in the isolation policies but directly executes them in the first-party context. Note that a web page may contain multiple frames, *i.e.*, a main frame and multiple iframes. A script might be included in both the main frame and iframes, and could be assigned different contexts in different frames. In order

```
{
  "https://www.example.com": [
    {
      "match": "https://www.lib-1.com/a.js",
      "context": "1",
      "read": [],
      "read by":  [["myFunc", "B"]],
      "ID": "A"
    },
    {
      "match": "https://www.example.com/b.js",
      "context": "1",
      "read": [],
      "read by": [],
      "ID": "B"
    },
    {
      "match": "https://www.lib-2.com/c.js",
      "context": "3",
      "read": [],
      "read by": [],
      "ID": "C"
    }
  ]
}
```

Listing 5: Isolation policy of https://www.example.com.

to distinguish the scripts included in different frames, JSIsolate further groups the policies based on the URL of the frame in which the scripts are included. We also perform an approximate match on frame URLs, as they may contain random strings. Details are in §4.

To help the developers understand and adjust the isolation policies, JSIsolate further includes a list of read operations that lead to the decision of the context for static third-party scripts. Each read operation is represented as a read target and the unique ID of the script that defines the read target.

Listing 5 presents the isolation policy generated for scripts A, B and C in Listing 4. We include script B here to demonstrate the auxiliary information generated for web developers. In practice, we do not generate policies for static first-party scripts as they are all executed in the first-party context.

3.4 Creating Isolation Environments

JSIsolate leverages the *isolated world*s in the Chromium browser to isolate JavaScript in separate contexts. In order to isolate content scripts from browser extensions, Chromium executes content scripts from each extension in an *isolated world*, while the normal scripts fetched from the web are executed in the *main world*. The *isolated world* is a concept in V8 binding and is used to isolate the global variable scopes and DOM wrappers for each extension. Scripts in an isolated world use a separate variable scope and prototype chain. Meanwhile, each isolated world has its own DOM wrapper while sharing the same underlying C++ DOM object. The event listeners registered by scripts in one isolated world will be executed in the context of that specific world.

JSIsolate is inspired by the idea of confining content scripts in isolated worlds. It creates an isolated world for each context using the `contextid` specified in the isolation policies, and injects the corresponding JavaScript code in the world. As the interaction between different worlds is prohibited by the browser, the scripts in one world cannot interfere with scripts in another world. Also, they cannot escape their worlds by injecting other JavaScript code, *e.g.*, by registering event listeners, because scripts in different worlds work on different DOM wrappers.

4 IMPLEMENTATION

We implemented a prototype of JSIsolate in the Chromium browser (version 71.0.3578.98) using around 2K lines of C++ code. We plan to open source our prototype implementation.

We implemented an `IsolationInjectionHost` object to inject scripts in different isolated worlds. We added several custom attributes (*e.g.*, `nid`, `initiator`, `parentScriptID` and `contextid`) in the Blink rendering engine to record the creation relationship and the execution context assigned to each script. The custom attributes are not exposed to V8, therefore cannot be modified by JavaScript.

We used the gremlins.js[1] library to perform a monkey testing after a full page loading. The library emulates real user visits by performing random inputs, scrolls and clicks on a page. This allowed us to trigger as much JavaScript code as possible and consequently capture more dependencies in an automated yet scalable fashion.

Parsing and Enforcing Isolation Policies. Our prototype parses a script isolation policy file when it starts to load a page. We plan to implement a new HTTP header to enclose the isolation policies in our future work. When parsing and rendering the HTML page, it loads a script in an isolated world according to the policy.

JSIsolate generates script isolation policies at two granularities: domain-level and URL-level. When using the domain-level policies, it matches scripts using only the domain name. If any script from one domain is assigned to the first-party context, JSIsolate loads all scripts from that domain in the first-party context. We adopt such an approach because scripts from the same domain are usually written to cooperate by the same developer. As script URLs might contain random strings, this also helps to eliminate the mismatches.

When configured to use the URL-level policies, JSIsolate matches each static script against the policy using the whole URL. Specifically, to tackle with the random script URLs, JSIsolate matches a script/frame in two modes, *i.e.*, the strict and loose matching mode. In the strict matching mode, a match is found if the URL (excluding the fragment) of an external script/frame is present in the policy. When no match is found, JSIsolate switches to the loose matching mode, where it performs a character-by-character comparison under the constraint that the domain names should be the same. The upper bound of different character ratio in script/frame URLs was selected as 0.15, which was determined based on our experiments on real-world websites. Nevertheless, the developers or users can configure a different threshold.

Developers could configure JSIsolate to enforce policies in either level according to their preferences. We will discuss in §5 how different policy granularities affect the compatibility of JSIsolate.

Note that if there is no exact or approximate match for a script, JSIsolate assigns a fallback context for it. In §5.3, we will demonstrate how different choices of fallback context affect the compatibility of JSIsolate on real-world websites. In some cases, JSIsolate may fail to find a match for the embedding frame URL and therefore cannot locate the isolation policies. JSIsolate executes all scripts in that frame in the first-party context for compatibility concerns. In a real deployment, the policies will be included in an HTTP response header (see §6), which would avoid the mismatches on embedding frame URLs. To accurately measure the overhead and compatibility in our current evaluation, we disabled script isolation in iframes.

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

Table 2: Context assignment results (#scripts / #websites) of static third-party scripts.

	w/ ground truth	w/o ground truth
URL-level	**12,516 / 960**	
3rd-party context	2,437 / 544	2,696 / 568
1st-party context	10,009 / 889	9,742 / 888
Both contexts	70 / 56	78 / 62
Domain-level	**12,516 / 960**	
3rd-party context	1,600 / 464	1,621 / 469
1st-party context	10,879 / 889	10,858 / 888
Both contexts	37 / 33	37 / 33

Nonetheless, the developers can easily deploy JSIsolate to isolate scripts in all frames.

5 EVALUATION

In this section, we first describe our analysis on script dependency relationship of popular websites (§5.1). We then present the evaluation, answering the following questions: 1) Whether JSIsolate can improve client-side JavaScript reliability (§5.2); 2) Whether JSIsolate is compatible with real-world websites (§5.3); and 3) What the performance overhead of JSIsolate is (§5.4).

5.1 Dependency Analysis

Among the Alexa top 1K website domains, 142 are present on the domain relationship list. On these websites, if the script domain is also on the list, JSIsolate determines execution context for it based on the domain dependencies in the list. For the other scripts whose domains are not on the list, JSIsolate analyzes script dependencies as described in §3.3. For instance, domain yimg.com belongs to the same entity Yahoo! as yahoo.com, but gstatic.com does not. Therefore, script https://s.yimg.com/.../tdv2-wafer-featurebar.js was assigned to the first-party context on website https://www.yahoo.com, while script https://www.gstatic.com/.../firebase.js was isolated in the third-party context.

We also used JSIsolate to collect the JavaScript object access logs on Alexa top 1K websites for analyzing the script dependency. Excluding inaccessible and timeouted websites, we gathered valid log files on 978 websites. The logs and the automatically generated isolation policies are available at https://zenodo.org/record/4892853. In total, we observed 12,516 static third-party scripts on 960 websites. We analyzed the access logs and generated URL and domain-level isolation policies. We also compared the results with and without the domain relationship list, and found the list helped to identify more scripts that belong to the first-party organization on 72 (resp. 11) websites in the URL-level (resp. domain-level) policies. For example, on website https://vimeo.com, a static script from vimeocdn.com would be assigned to the first-party context with the list as ground truth. We list the context assignment results of static third-party scripts in Table 2. All static scripts from the first-party domain are executed in the first-party context.

In Figure 3, we visualize the dependency relationship between all statically included scripts on the website https://www.rakuten.co.jp. Numbers in grey circles are the `scriptID` of static first-party scripts, which are executed in the first-party context. The white circles represent other scripts statically included from a third-party domain. An arrow from A to B indicates some object(s) defined by B is read by A. As shown, script `48` defined a global object `googletag`, which was read by script `74` via `googletag.fifWin`. Specifically, script `74` and `48` were statically included from a third-party domain googletagservices.com and the first-party domain rakuten.co.jp, respectively. In this case, script `74` and `48` were apparently dependent. Therefore, we assigned `74` to the first-party context. In contrast, scripts `17` and `154` were put in the third-party context because they did not have dependency with any scripts in the first-party context.

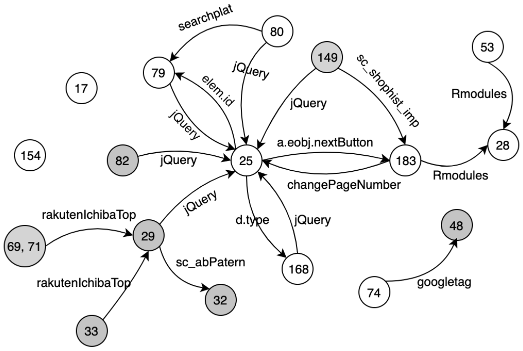

Figure 3: Script dependency relationship on https://www.rakuten.co.jp.

Note that a jQuery script `25` read objects defined by script `79`, `168` and `183`. We believe script `25` is a customized jQuery script that relied on other scripts to achieve its functionalities. As it was also read by first-party scripts, JSIsolate loaded it in the first-party context. As a result, scripts `79`, `80` and `168` were also executed in the first-party context, as they all called the `jQuery` function defined by `25`. Script `183` was read by both `25` and the first-party script `149`. Thus it was also executed in the first-party context.

We also detected on website https://www.tribunnews.com one static script https://www.gstatic.com/firebasejs/5.5.6/firebase-app.js that was read by another static script `A` from cdn-3.tstatic.net. As script `A` read multiple objects defined in a script from the first-party domain, we classified `A` as a first-party context script. Meanwhile, script firebase-app.js was also read by multiple scripts that did not have any functional dependency with first-party scripts. We therefore treated these scripts as third-party scripts. Since firebase-app.js did not read any script and was read by both first-party and third-party scripts, we assigned a `contextid` of "both" to it.

5.2 Reliability Improvement

We evaluate the effectiveness of JSIsolate in mitigating the reliability problems described in §2 using several web mashup examples.

Avoiding Global Object Overwrites. For the global identifier conflict in Listing 1, as the libraries are not expected to interact with each other, JSIsolate naturally run them in different contexts. Although `client.js` will be automatically recognized as dependent on `aframe.js`, JSIsolate will report the conflicting writes to developers, who have sufficient knowledge to determine whether `aframe.js` shall be allowed to overwrite `ocanvas.js`.

Further consider the example in Listing 6, where two scripts assign values of different types to the same global identifier `dropdown`. As the external script redefined `dropdown` as a boolean value, a

```
function dropdown() {...} // a.js

var dropdown = false; // b.js

if(dropdown()) {...} // client.js: Runtime TypeError
```

Listing 6: An example of conflicting variable definitions.

call to it would cause runtime exceptions and may lead to Denial-of-Service (DoS). Similarly, given knowledge of detected conflicts, developers may decide whether `b.js` should run in a separate execution context. Therefore, JSIsolate effectively avoids conflicting definitions of global objects across different scripts, and is able to improve the reliability. However, a script can intentionally read first-party objects to build fake dependencies. We mitigate the risk through developer validation of the isolation policies. Specifically, JSIsolate reports each cross-script read operation that leads to the context assignment of third-party scripts in the policies. The developers thus can identify any unexpected reads and adjust the `contextid`s. According to our observation, in the domain-level (resp. URL-level) policies, each page contains on average 8.07 (resp. 7.96) static third-party scripts that have dependency relationship with first-party scripts and are involved in read operations; each such script is involved in on average 3.80 (resp. 3.83) read operations. Therefore, it is practical for web developers to validate the isolation policies as it is a one-time cost.

Mitigating Prototype Poisoning Attacks. Consider the prototype poisoning attack in Listing 2, where `a.js` reads `RegExp.prototype` and then writes `RegExp.prototype.test`. After that, `client.js` calls (*i.e.*, reads) method `test` from `RegExp.prototype`. JSIsolate does not record the read to object properties to avoid recognizing `a.js` and `client.js` as dependent. Therefore, JSIsolate can prevent the attack by isolating `a.js` in the third-party context, so that the prototype accessed by the first-party scripts would not be affected. In this way, we mitigate the unexpected interference and provide a higher level of reliability. Again, a script can deceive JSIsolate by forging the dependencies with first-party scripts. We report any conflicting write and read operations in each static third-party script to the developers, who would decide whether to adjust the policies.

Similar vulnerabilities have also been discovered on other password managers, *e.g.*, PassPack and MashedLife [2], and have been discussed in [17]. They could be mitigated in a similar way.

5.3 Compatibility

The concern on the compatibility of JSIsolate lies in two folds: 1) Whether JSIsolate is compatible with websites that do not enforce script isolation; and 2) Whether JSIsolate breaks the functionality of websites after script isolation. The answer to the first question is obviously *yes*, because JSIsolate by default executes all scripts in the same main world when no policy is explicitly specified. To measure the compatibility with real-world websites after script isolation, we used a Vanilla Chromium browser and our prototype to visit Alexa top 1K websites, and recorded the runtime exceptions separately. We enforced the default policies automatically generated from the script dependency logs. We study how different choices of fallback context in JSIsolate can affect the website compatibility.

To make the evaluation scalable, we opted for an automatic testing approach that performs basic interaction such as page scrolling after loading. We did not use monkey testing in the experiment as it can randomly trigger different code (hence different exceptions) in different runs, which makes it difficult to make a fair comparison. We acknowledge that some exceptions might not be observed due to incomplete code coverage. As we discuss in §6, this can be solved by integrating the dependency analysis and compatibility tests in the application regression testing process. We did manually inspect the exceptions on top 100[2] websites and identified the websites on which we observed new exceptions after isolating scripts.

First-party Context as Fallback for URL-level Policies. We firstly assigned a fallback `contextid` of “1” to unmatched scripts. Out of the top 100 websites, we observed on 1 website new JavaScript exceptions. With our manual analyses, the exceptions did not break any critical functionality and all the compatibility issues could be resolved. Specifically, website https://www.pornhub.com generated exceptions because scripts dynamically generated implicitly depended on scripts in a different context. This shows a current limitation of JSIsolate as it did not consider inline event listeners and the dynamic scripts in the dependency analysis, which we will discuss in §6. The exceptions could be mitigated by changing the `contextid` of the third-party script from `“3”` to `“1”` or `“both”`. We leave this as an option to the developer, who has sufficient information to decide the context of scripts. The problem can also be fixed by allowing developers to specify the related origins using First-Party Sets [11]. More details are discussed in §6.

Third-party Context as Fallback for URL-level Policies. We further tested JSIsolate by assigning a `contextid` of “3” to unmatched scripts. Except for the above website, we observed 4 more websites that threw exceptions after script isolation. The exceptions were generated because a long random string is contained in the URL of an external script from a third-party domain, or in the URL of the embedding iframe of the script. Although these scripts were explicitly dependent on first-party scripts, they were mistakenly classified as third-party scripts. For instance, the URL of script https://static.xx.fbcdn.net/rsrc.php/v3/yg/r/..._nc_x=... contained a random query string that was 11 characters long, which made our URL match fail because the query string was updated periodically. We could not exclude the query strings during the matching, as a frame may include multiple scripts whose URLs only differ in the query strings. As a result, the script from the third-party CDN domain was misclassified as a third-party script on website https://www.facebook.com, although it was read by first-party scripts. Nevertheless, the exceptions did not affect other scripts, and no critical functionality was broken. These problems can be fixed by recollecting the logs and generating the latest isolation policies. Alternatively, we could leverage mechanisms like SRI to match the content of a script (by computing a hash value) instead of only its URL. We could also perform an approximate match on external script source code in the browser to tackle with the random strings. In addition, as those scripts are included by the website developers, they could permanently set the script `contextid` attributes just once regardless of the URLs that will be used.

Domain-level Policies. When JSIsolate is deployed to enforce domain-level isolation policies, we did not find any of the top 100 website throw more exceptions. Note that the choice of fallback

[2] Except for http://www.17ok.com that is inaccessible, we gathered valid data on 99 websites.

Table 3: Average log collection overhead on page loading.

Round	Vanilla (s)	JSIsolate (s)	%Inc (%)
1	5.74	7.73	34.67
2	6.14	7.93	29.15
3	6.69	8.35	24.81

Table 4: Average log collection memory overhead.

Round	Vanilla (MB)	JSIsolate (MB)	%Inc (%)
1	67.72	80.24	18.49
2	67.48	79.26	17.46
3	67.90	79.89	17.66

context does not affect the results, as the mismatches can only be caused by scripts from new domains absent from the policies, which we did not encounter on the top 100 websites.

Summary. Our evaluation shows that JSIsolate entails very few and fixable compatibility issues and works well with popular real world web applications in general.

5.4 Performance

To measure the performance overhead of JSIsolate, we used the selenium chromedriver to run a Vanilla Chromium browser and JSIsolate to visit the Alexa top 1K websites. We set a timeout of 160 seconds and for websites that timeouted after several retries, we enlarged the timeout to 360 seconds. We recorded the average page loading time based on the `window.performance.timing` interface, and computed the average memory consumption in 3 runs.

5.4.1 Log Collection Overhead. We ran JSIsolate in the log collection mode to measure the overhead when collecting the JavaScript object access logs and script initiator records. We successfully gathered valid data on 994, 994 and 995 websites in the 3 runs. The results are presented in Table 3 and Table 4.

In the log collection mode, the slowdown on page loading varies from 24.81% to 34.67% and is averaged at 29.54%. The memory overhead ranges from 17.46% to 18.49%, with an average of 17.87%.

We further tested JSIsolate on industry-standard benchmarks, and the evaluation results are listed in Table 5. As shown, the most significant overhead was observed on the SunSpider benchmark, which tests the core JavaScript language features like hashing and matrix processing, *etc.* Overall, 45 scripts were statically included on the test page, and JSIsolate recorded over 230K read/write operations. On the other two benchmarks, however, the overhead was comparable with what we observed on real world websites.

Note that a developer would need to recollect the logs only if she/he changes the included scripts, or a script changes its interaction with other scripts significantly. Any runtime errors caused by the changes can be reported to help the developers quickly set a new isolation policy and/or easily identify a suspicious script. Therefore, the overhead is acceptable as the log collection needs not to be performed frequently.

Table 5: Log collection overhead on industry-standard JavaScript benchmarks. The test results represent the geometric mean of run/sec for each individual test.

Benchmark[3]	Vanilla	JSIsolate	%Dec (%)
Dromaeo	1,957.18	1,736.79	11.26
SunSpider	2,855.82	2,024.38	29.11
V8 Test Suite	1,804.71	1,759.29	2.52

Table 6: Slowdown on average page loading time during script isolation (URL-level policies).

Round	Vanilla (s)	JSIsolate (s)	%Inc (%)
1	5.67	5.99	5.64
2	6.14	6.71	9.38
3	5.67	6.17	8.82

Table 7: Increase of average memory consumption in script isolation (URL-level policies).

Round	Vanilla (MB)	JSIsolate (MB)	%Inc (%)
1	66.19	67.49	1.96
2	67.43	69.00	2.33
3	67.75	68.73	1.45

Table 8: Slowdown on average page loading time during script isolation (domain-level policies).

Round	Vanilla (s)	JSIsolate (s)	%Inc (%)
1	4.90	5.19	5.92
2	5.29	5.57	5.29
3	4.68	5.09	8.76

5.4.2 Script Isolation Overhead. We ran JSIsolate in policy enforcement mode to evaluate the performance overhead incurred for isolating scripts. JSIsolate reads the isolation policies from local files and assigns `contextid` to scripts according to the strategy in §3. We successfully collected performance data for all 978 websites. Specifically, we measured the overhead using both the URL-level and domain-level policies. The experiment results are presented in Table 6, Table 7, Table 8 and Table 9.

When using URL-level policies, the average slowdown is 7.95%, and on 47.51% websites the page loading time increased by no more than 1%. For domain-level policies, the average slowdown is 6.66%. This demonstrates that JSIsolate incurs small overhead on page loading. Using URL-level policies, the average memory overhead is 1.91%, and the overhead is no greater than 1% on 48.88% websites. For domain-level policies, the average memory overhead is 1.34%. As shown, the memory overhead is also negligible.

Summary. The experiments show that JSIsolate can efficiently isolate scripts with very limited overhead.

6 DISCUSSION AND FUTURE WORK

We discuss our work's current limitations and our future work.

Policy Deployment. JSIsolate reads the isolation policies from local files in our current prototype. In the future, we plan to enable developers to use a new HTTP response header for the policies. Alternatively, the policies can be embedded in the HTML page, like internal CSS. Moreover, the policies can be integrated with existing websites through minimal changes, *e.g.*, by a simple update of the template used in web development frameworks (*e.g.*, Angular).

[3] All the test cases are available at: https://github.com/jeresig/dromaeo

Table 9: Increase of average memory consumption in script isolation (domain-level policies).

Round	Vanilla (MB)	JSIsolate (MB)	%Inc (%)
1	66.39	67.68	1.94
2	66.72	67.32	0.90
3	67.42	68.22	1.19

Dependency Analysis. JSIsolate excludes dynamic scripts and event listeners from the dependency analysis, which may cause exceptions after script isolation. We adopt this design for preventing cross-context interference through dynamic script inclusion, and event listeners are intrinsically isolated in different DOM wrappers, as explained in §3. Nevertheless, to mitigate the exceptions, developers can adjust the policies with limited manual efforts (see §5.2). We leave it as a future work to investigate other possible solutions.

Code Coverage. JSIsolate performs monkey testing to capture and analyze script dependencies in a scalable way. The monkey testing, however, does not promise a full code coverage. Therefore, the dependency analysis might be incomplete and cause some exceptions after script isolation. However, the automatic dependency analysis can be integrated with the current website regression testing process, which thoroughly tests the application. This introduces very limited overhead in the development cycle and could help identify possible compatibility issues and improve code reliability.

Classification of Scripts. JSIsolate relies on the domain relationship list from Disconnect to identify scripts from domains related to the first-party domain. As shown in §5.1, only 142 of the evaluated domains were present on the list. However, we expect that with the contribution of the community, the list would be more comprehensive in the future. Meanwhile, Google has proposed First-Party Sets [11], that allows developers to explicitly specify related domains in a .json file, and will be implemented in the Chromium browser in the near future. We believe the First-Party Sets would further improve our ability to precisely classify scripts.

Applicability in Other Browsers. In addition to the Chromium browser, other full-fledged browsers also provide JavaScript code isolation mechanisms. For instance, the Safari browser also executes scripts injected by App Extensions in isolated worlds. Mozilla Firefox supports the `Xray vision` feature [21], that uses separate contexts for content scripts and normal scripts. Therefore, the design of JSIsolate can be easily extended to other popular browsers.

7 RELATED WORK

JavaScript Isolation. [23, 27] confined malicious JavaScript code using a reference monitor, and [26] restricted the interaction between Adobe ActionScript and JavaScript code. Magazinius *et al.* fixed several design flaws of [27] in [16]. Agten *et al.* sandboxed sensitive DOM objects to isolate third-party scripts, with the page loading overhead of over 30% [3]. Treehouse [12] and SafeJS [7] used web workers to run scripts in separate threads. However, they required the developers to specify the permitted operations of each isolated script, which is non-trivial for normal web developers. Some other works isolated scripts using iframes [10, 13, 19, 33, 42]. They need to interpose the interaction between scripts and DOM of the original page. In contrast, JSIsolate utilizes isolated worlds to separate functionally independent scripts, which does not affect the way that scripts are executed. Further, it is able to generate isolation policies automatically, which is more efficient and easier to deploy. Adam *et al.* proposed to isolate content scripts of browser extensions using *isolated world* in Firefox browser [4], which has a different target from JSIsolate. BREAKAPP confines JavaScript modules in protected compartments [36]. JSIsolate can also isolate normal web scripts and any JavaScript code created dynamically.

Limiting JavaScript Privileges. ESCUDO organized JavaScript principals and resources in rings with different privileges [14]. It prevents principals with lower privileges from accessing resources with higher privileges. Meyerovich *et al.* registered advice for security-sensitive functions like `eval` to restrict the calls to them [18]. WebJail uses deep advice to enforce least-privilege integration of scripts [35]. Snyder *et al.* developed an extension to block unsafe JavaScript features based on pre-defined policies [31]. Flowfox intercepts DOM API calls through secure multi-execution to protect data confidentiality [9]. COWL enforces label-based access control to prevent scripts from exfiltrating sensitive data [32]. Chudnov *et al.* implemented an inlined information flow control (IFC) monitor to protect data confidentiality and integrity [8]. BrowserShield [29] and JaTE [34] rewrite HTML pages and scripts to safe equivalents. These works have a different target from JSIsolate, which separates scripts with different functionalities to provide higher reliability.

Isolation in the Browser. Prior works have proposed to render untrusted web content in lower privileged renderer processes [1, 28, 43]. They limited the capability of attackers that compromise the renderer. However, the browser still loads content from different websites in the same renderer process, thus attackers can access cross-site data. A recent work of Google renders untrusted content from different websites in separate processes [30]. Fission [20] further enforces the isolation at the granularity of origins. Narayan *et al.* sandboxed vulnerable C/C++ libraries used by the renderer in lightweight WebAssembly sandboxes [22]. These works are orthogonal to JSIsolate, which assumes a trusted browser.

8 CONCLUSION

Third-party scripts are prevalently used on web application to enhance the functionalities. However, the current browser architecture allows all scripts to access a shared context, causing unexpected effects. In this paper, we present JSIsolate, a browser-based framework that improves JavaScript code reliability by executing scripts in isolated execution contexts. JSIsolate analyzes the dependency relationship between scripts on a web page, and identifies the scripts from third-party domain that are not functionally dependent on first-party scripts. It then executes the scripts in a separate isolated environment to prevent them from interfering with first-party scripts. We demonstrate with real world examples that JSIsolate offers a practical approach to enhancing JavaScript reliability and separating functionality-independent code. Our experiments on Alexa top 1K websites further prove that JSIsolate is backward compatible, and introduces limited performance overhead.

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] [n.d.]. WebKit2. https://trac.webkit.org/wiki/WebKit2.

[2] Ben Adida, Adam Barth, and Collin Jackson. 2009. Rootkits for javascript environments. In *3rd USENIX Workshop on Offensive Technologies (WOOT '09)*.

[3] 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 (ACSAC)*.

[4] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. 2010. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[5] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content security problems? evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[6] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2018. Semantics-based analysis of content security policy deployment. *ACM Transactions on the Web* 12, 2 (2018), 1–36.

[7] Damien Cassou, Stéphane Ducasse, and Nicolas Petton. 2013. Safejs: Hermetic sandboxing for javascript. *arXiv preprint arXiv:1309.3914* (2013).

[8] Andrey Chudnov and David A Naumann. 2015. Inlined information flow monitoring for JavaScript. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado.

[9] 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 19th ACM Conference on Computer and Communications Security (CCS)*. Raleigh, NC.

[10] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. 2008. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of the 17th International World Wide Web Conference (WWW)*. Beijing, China.

[11] Google. [n.d.]. First-Party Sets. https://github.com/privacycg/first-party-sets.

[12] Lon Ingram and Michael Walfish. 2012. TreeHouse: JavaScript sandboxes to helpWeb developers help themselves. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[13] Collin Jackson and Helen J Wang. 2007. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th International World Wide Web Conference (WWW)*. Banff, Alberta, Canada.

[14] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J Chapin. 2010. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS)*.

[15] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[16] Jonas Magazinius, Phu H Phung, and David Sands. 2010. Safe wrappers and sane policies for self protecting JavaScript. In *Nordic Conference on Secure IT Systems*. Springer, 239–255.

[17] Leo A Meyerovich, Adrienne Porter Felt, and Mark S Miller. 2010. Object views: Fine-grained sharing in browsers. In *Proceedings of the 19th International World Wide Web Conference (WWW)*. Raleigh, NC.

[18] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[19] James Mickens. 2014. Pivot: Fast, synchronous mashup isolation using generator chains. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.

[20] Mozilla. [n.d.]. Project Fission. https://wiki.mozilla.org/Project_Fission.

[21] Mozilla. [n.d.]. Xray vision. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/Xray_vision.

[22] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2019. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[23] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. 2011. Towards fine-grained access control in javascript contexts. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*.

[24] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden.

[25] Jackson Patrick, Mills Codi, Oppenheim Casey, and Toyens Victor. [n.d.]. Disconnect. https://disconnect.me.

[26] Phu H Phung, Maliheh Monshizadeh, Meera Sridhar, Kevin W Hamlen, et al. 2014. Between worlds: Securing mixed javascript/actionscript multi-party web content. *IEEE Transactions on Dependable and Secure Computing* 12, 4 (2014), 443–457.

[27] 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.

[28] Charles Reis, Adam Barth, and Carlos Pizano. 2009. Browser security: lessons from google chrome. *Queue* 7, 5 (2009), 3–8.

[29] 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* 1, 3 (2007), 11–es.

[30] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[31] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[32] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado.

[33] Mike Ter Louw, Karthik Thotta Ganesh, and VN Venkatakrishnan. 2010. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Security Symposium (Security)*. Washington, DC.

[34] Tung Tran, Riccardo Pelizzi, and R Sekar. 2015. Jate: Transparent and efficient javascript confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*.

[35] 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 (ACSAC)*.

[36] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[37] W3C. [n.d.]. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/.

[38] W3C. [n.d.]. Cross-origin resource sharing (CORS). https://www.w3.org/wiki/CORS.

[39] W3C. [n.d.]. HTML5. https://www.w3.org/TR/2009/WD-html5-20090423/browsers.html.

[40] W3C. [n.d.]. Same-origin Policy. https://www.w3.org/Security/wiki/Same_Origin_Policy.

[41] W3C. [n.d.]. Subresource Integrity (SRI). https://www.w3.org/TR/SRI/.

[42] Saman Zarandioon, Danfeng Yao, and Vinod Ganapathy. 2008. Omos: A framework for secure communication in mashup applications. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*.

[43] Andy Zeigler. [n.d.]. IE8 and Loosely-Coupled IE (LCIE). https://docs.microsoft.com/en-gb/archive/blogs/ie/ie8-and-loosely-coupled-ie-lcie.

[44] Mingxue Zhang and Wei Meng. 2020. Detecting and understanding JavaScript global identifier conflicts on the web. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Sacramento, CA.

[45] Mingxue Zhang, Wei Meng, and Yi Wang. 2019. Poster: Finding JavaScript Name Conflicts on the Web. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK.