




MyWebGuard: Toward a User-Oriented Tool for Security and Privacy Protection on the Web

Panchakshari N. Hiremath¹, Jack Armentrout¹, Son Vu², Tu N. Nguyen³,
Quang Tran Minh⁴, and Phu H. Phung¹ 

¹ Intelligent Systems Security Lab, Department of Computer Science,
University of Dayton, Dayton, OH, USA

{hiremathp1, armentroutj2, phu}@udayton.edu

² Truman State University, Kirksville, MO, USA

slv8887@truman.edu

³ Purdue University Fort Wayne, Fort Wayne, IN, USA

nguyent@pfw.edu

⁴ Ho Chi Minh City University of Technology, VNU-HCM,
Ho Chi Minh City, Vietnam

quangtran@hcmut.edu.vn

<https://issec-lab-udayton.github.io>

Abstract. We introduce a novel approach to implementing a browser-based tool for web users to protect their privacy. We propose to monitor the behaviors of JavaScript code within a webpage, especially operations that can read data within a browser or can send data from a browser to outside. Our monitoring mechanism is to ensure that all potential information leakage channels are detected. The detected leakage is either automatically prevented by our context-aware policies or decided by the user if needed. Our method advances the conventional same-origin policy standard of the Web by enforcing different policies for each source of the code. Although we develop the tool as a browser extension, our approach is browser-agnostic as it is based on standard JavaScript. Also, our method stands from existing proposals in the industry and literature. In particular, it does not rely on network request interception and blocking mechanisms provided by browsers, which face various technical issues.

We implement a proof-of-concept prototype and perform practical evaluations to demonstrate the effectiveness of our approach. Our experimental results evidence that the proposed method can detect and prevent data leakage channels not captured by the leading tools such as Ghostery and uBlock Origin. We show that our prototype is compatible with major browsers and popular real-world websites with promising runtime performance.

Keywords: Privacy · Web security · Online tracking

S. Vu—Work performed while the author was visiting the Intelligent Systems Security Lab, Department of Computer Science, University of Dayton.

© Springer Nature Switzerland AG 2019

T. K. Dang et al. (Eds.): FDSE 2019, LNCS 11814, pp. 506–525, 2019.

https://doi.org/10.1007/978-3-030-35653-8_33

1 Introduction

Privacy is a big challenge and risk today for Internet users [7]. This risk is mostly due to the presence of online trackers on almost all websites [17]. These trackers typically collect users' sensitive information such as personal data, browsing history, activities, and interests [33], mostly without the awareness of users [33, 35]. Standard web security mechanisms such as same-origin policy [41] or Content-Security policy (CSP) [60] could not prevent these privacy risks since the trackers, either from the first- or third parties, are included by the developers [36]. Do Not Track (DNT) [61] is a mechanism to prevent tracking by sending an HTTP header indicating that a browser does not want to be tracked. Although most web browsers support DNT, there is currently no policy or mechanism about how a website has to respond upon receiving a DNT header. This technical issue of DNT leaves it a currently ineffective solution for security on the web [6].

Advanced users concerning their privacy usually adopt browser-based blocking tools for privacy protection such as ad or tracker blocker extensions [33]. While these tools are effective in blocking third-party trackers [36], there are a couple of limitations in these mechanisms. First, existing blocking browser extensions only enforce the “all-or-nothing” rule. This rule either blocks or allows a tracker or an ad network based on a URL defined in a filter list, generally identified by the tool or set by users [5]. This blocking approach creates challenges and dilemmas for users as they do not know if they should block an ad or a tracker [33]. Also, several studies show that not every user wants to block ads or trackers [9, 59], and a big crowd desire advanced methods to control their footprint [1, 30]. Second, the existing blocking mechanisms implemented in browser-based extensions face several technical issues outlined below.

The first technical issue is that the implementation of the blocking mechanism is browser-dependent. Extensions usually implement the blocking mechanism by intercepting all network requests from a web page and blocking it if the URL is in a filter list [13, 36]. This method relies on a web API `webRequest` provided by major browsers [22, 40], which developers have to follow the requirements of a specific browser. For example, recently, Google has just announced to replace their `webRequest` API by the `declarativeNetRequest` API together with the Manifest specification V3 [13, 55]. This change will restrict the functionality of many of the current blockers on the marketplace. For example, it limits the maximum number of rules and domains that can be blocked by extensions like uBlock Origin [55]. Another technical issue is that extensions are dependent on the filter list predefined by developers. Therefore, new trackers or other third-party content are not captured or blocked by these extensions. For example, in [64], the authors demonstrated that existing extensions could not catch requests that utilize a DNS CNAME alias in order to prevent detection. Our experiments also confirm this technical issue. As demonstrated in Sect. 5.1, popular extensions such as Ghostery and uBlock Origin do not detect trackers and data leakage channels simulated in our test suite.

The limitations discussed above motivate us to develop a novel approach to protecting web users. Different from existing tools that block a particular

network request, our method is to monitor the code execution in a browser. The goal of our code monitoring is to ensure that users' sensitive information cannot be leaked to the outside of a browser without the concern of users. As JavaScript is the language executed in browsers to perform all of the activities on a web page, we propose to intercept critical privacy-related JavaScript operations. In particular, we intercept JavaScript operations that can read data belonging to the user, e.g., cookies, and can send out data to the Internet. We then define and enforce context-aware policies on these operations to prevent possible privacy leakages. Our proposed method also stands from the state-of-the-art of JavaScript monitoring by tracking the origin of the code, i.e., where the code comes. From this origin tracking mechanism, we can enforce distinct policies for each origin of the code. This mechanism advances (and in contrast to) the traditional same-origin policy [41], which treats all the code, even included from an external server, within a web page as the same origin. Our main contributions in this work include:

- We introduce a novel approach to controlling the behaviors of JavaScript code within a web page to detect and prevent potential privacy leakage.
- We implement a proof-of-concept prototype as a browser extension, named **MyWebGuard**. Our **MyWebGuard** tool monitors the source of data and the sinks where the data can be sent to in order to detect potential data leakage precisely. Our mechanism is in contrast with existing blocking browser extensions that only focus on the web request interception with a static filter list.
- We perform evaluations and report practical experimental results on various aspects. Our approach is browser-agnostic, as it is compatible with major browsers. We show that our prototype implementation can detect and prevent potential data leakage while preserving the web page functionality. We also report the performance overhead of our prototype that is quite low for popular real-world websites.

We proceed as follows. In the next section, we review the background, including the landscape of browsers and their security, browser extension, and web security standards. We survey the literature and discuss related work compared with our work in Sect. 2.4. In Sect. 3, we describe our technical approach and the design of our method. We present our prototype implementation in Sect. 4. In Sect. 5, we demonstrate our evaluations and discuss the experimental results. Section 6 concludes our contributions and outlines the future work.

2 Background and Related Work

2.1 Security and Privacy in Browsers

In this subsection, we review the landscape of existing browsers and their security and privacy features. We relate these features to our work.

There are many browsers in the industry, including popular ones such as Google Chrome, Apple Safari, Mozilla Firefox, and more niche browsers such

as Chromium and Brave. Almost all browsers have and implement security and privacy standards to ensure degrees of users' privacy either by default or within the browser settings. Among popular browsers, Google Chrome is considered as a leader in browser security with automatic updates and a Web Authentication API [10]. However, due to its closed-source structure and Google's incident for collecting user information, some users feel that Google has the incentive to break its stance on privacy [8]. Microsoft Edge uses sandboxing to contain browser processes, and it retains Internet Explorer's filtering of suspicious websites. However, Edge has not retained IE's tracking protection, and biannual updates can leave plenty of time for attackers to utilize unpatched exploits [38]. Mozilla Firefox offers users a wide range of security features that are comparable to Chrome. These include add-on warnings, malware protection, content blocking, and filtering of reported malicious websites. Also, thanks to its open-source, users can be confident that their browser is functioning without the inclusion of malicious code [44].

Looking at more niche browsers that advertise a greater emphasis on user privacy, Chromium retains the positives of Chrome while being open-source [10]. Privacy concerns with Chromium include the lack of automatic updates that could leave some users open to attacks, and WebRTC leaks are a concern since it cannot be disabled without third-party software [8]. Brave is a relatively new browser that focuses heavily on privacy; features include a built-in ad-blocker, chrome extension functionality, and fingerprinting blocker, to name a few. While Brave is still new and gains limited market share of browsers, its open-source structure and firm stance on privacy are gearing it up to be a strong contender in the browser landscape [8].

Although our work is to build a tool added-on in a browser, our approach can be integrated into a browser to enhance its security and privacy features for protecting web users.

2.2 Browser Extensions

A browser extension is an open-source software module, developed using web technology, i.e., HTML, CSS, and JavaScript, to be integrated into a browser by users. When loading a web page, a browser also loads and executes installed extensions enabled for that web page [43]. Therefore, JavaScript code in a browser extension has the same privilege as the web page, i.e., it can access and modify the data or content of the web page [43]. A typical extension contains a manifest specification in JSON format, HTML pages, and JavaScript files. A browser extension can access general web APIs, available for every web page, and a special set of JavaScript APIs provided by browsers but are not available for web pages. For example, Google Chrome and Opera browsers provide the extension API (<https://developer.chrome.com/extensions>), which is also supported in other major browsers such as Firefox or Microsoft Edge [43].

In this work, we develop our tool as a browser extension. In Sect. 2.4, we survey and discuss related work in browser extensions and their security.

2.3 Web and JavaScript Security

Our work focuses on the security and privacy of web pages within a browser. Therefore, server-side security such as SQL Injection, command injection, or denial-of-service attacks is out-of-scope of this work. In this subsection, we briefly review the web page security model and JavaScript security standards, including Same-Origin Policy and Content Security Policy.

Nowadays, almost all web pages contain JavaScript code. Statistics in [62] show that 95.2% of all websites contain JavaScript code. Among these, websites include JavaScript code from external third-party servers. These facts are evidenced in a survey of the Alexa Top 10,000 websites, which reveals that 88.45% of them include at least one third-party JavaScript library [47]. The work also exposes some contents in the trusted libraries of these sites that can be compromised. They later identify four newly discovered vulnerabilities that could be used in attacks. The work also reviews the effectiveness of proposed solutions for protecting web applications that utilize third-party content.

In principle, JavaScript code in a web page is loaded and executed by a browser within a JavaScript engine. JavaScript code can interact with users, modify the web content, and can access (read/write) data stored in a browser such as cookies. Browsers enforce the same-origin policy to ensure that JavaScript code from one origin cannot access the data belongs to other origins.

Same-Origin Policy. The same-origin policy (SOP) is a critical web security standard that prevents JavaScript code from one origin from accessing data of other origins [41]. An origin in the web is identified by the scheme, host, and port of a URL. The SOP does not apply for the code included in a web page but sourced from a third-party server. For example, if a web page from <https://mywebguard-host.github.io> (the host) includes JavaScript code from <https://thirdparty.com>, e.g., using

```
<script src="https://thirdparty.com/script.js"></script>
```

then the script code from <https://thirdparty.com/script.js> is executed in the host web page and has the same origin as the host. This policy means that the external code can access data belongs to that host. This issue is a known limitation of SOP as identified in the literature, e.g., [45,48,54]. Our work addresses this drawback by tracking where the code comes from and enforce different policies for different code origin.

Content Security Policy. Content-Security-Policy (CSP) [60] is a web standard that functions to prevent code injection attacks such as Cross-Site Scripting (XSS). This prevention is done by allowing the creation of a whitelist of trusted content that can be rendered and executed by the browser. The CSP can be used to prevent data leakage to external domains, not in the whitelist. However, it cannot prevent potential privacy leakage to whitelisted domains, as we investigated in this work. Moreover, there has also been a surge in the attempts to study the CSP. In [63], the authors explore how challenges presented by CSP

lead to a low adoption rate among the Alexa Top 1M. The results show that the inclusion of CSP lags other security headers, and CSP policies are often ineffective at preventing content injection attacks like XSS. Additionally, they also suggest an improvement in CSP that could increase its use on the web.

2.4 Related Work

Below, we survey the literature on topics related to our work, including JavaScript security and browser extensions. We also discuss how our work advances the state-of-the-art.

JavaScript Security. In the past decade, there are many research works and proposals focused on the topic of JavaScript security. In 2007, BrowserShield [52] was introduced as a JavaScript security solution by vulnerability-driven filtering dynamic HTML in web pages. The BrowserShield system rewrites web pages to secure vulnerabilities relied on embedded scripts. Caja [21, 39] is a similar approach developed at Google. Google Caja allows passive data to become active content and rewrites scripts into an object-capability language to ensure that their operations are safe on a webpage. In [31], Maffei *et al.* design language-based methods that websites use to filter untrusted JavaScript. They also propose a foundation for language-based filtering that addresses website vulnerabilities. Phung *et al.* [49] introduce a lightweight self-protecting JavaScript approach to controlling the behaviors of JavaScript code to prevent potential attacks. Meyerovich and Livshits in [37] present a new design, termed ConScript, that empowers to host webpages byways of fine-grained security policies. These policies are to constrain executed code to protect against vulnerabilities opened when untrusted JavaScript codes appear. In [48], Phung *et al.* introduce a new approach to enforcing security policies on untrusted third-party ad content, referred to as FlashJaX. Their design, FlashJaX, addresses the problem of vulnerability assessment sites that are mixed by JavaScript with Adobe ActionScript content.

There are also other JavaScript security proposals; however, they focus on advertisement networks. For example, ADSafe [12] is a safe subset of JavaScript for ads. Ad developers must follow this subset so that an ad can be included in a hosting web page; otherwise, it will be rejected. In [50], Politz *et al.* propose a lightweight and efficient technique to verify sandbox sources that utilize ADSafe to demonstrate the effectiveness of the system by securing previously unknown weakness within ADSafe rooted in sandboxing. And, Finifter *et al.* in [18] examine a vulnerability found among one-third of the Alexa Top 100, accessible via ADSafe-verified advertisements. This vulnerability is rooted in third-parties exploiting prototype objects of the hosting page. The authors propose a JavaScript subset that retains static verification while upgrading security. Furthermore, in [24], there is a development of an online advertising system, referred to as Privad, that aims to secure user privacy while still allowing advertisers to serve targeted ads. This design focuses on improving user's browser experience through increased speeds over other online advertising systems, presenting

microbenchmarks and informal analysis of Privad’s privacy properties. In addition to these studies, Fredrikson and Livshits in [19] present RePriv, a tracker implemented as a browser extension to collect user interests and share them with third-parties with the user’s permission. This paper also shows how RePriv can collect user interests to personalize the content on various websites with high quality and keep a low overhead, as well as preserve user privacy.

Information flow-control in JavaScript to prevent potential leakages in web pages, similar to our work, has also been studied in the literature. For instance, in [11], the authors present a web system supported by inlined information flow control monitor as well as an evaluation of that monitor. This application uses “mashups” of JavaScript code that is one of the most common types of codes using on many web apps today. Hedin *et al.* [26] introduce JSFlow as a method for tracking the information flow of JavaScript on sites that utilizes third-party code, deploying it as a browser extension. Through the utilization of JSFlow, they also indicate the different main policies of sharing sensitive user information on websites.

JavaScript security is still an emerging topic in recent years. For example, in 2018, the authors in [51] provide a comprehensive survey of existing client-side web application security solutions that consider desirable features and develop a framework for specifying and enforcing security policies for JavaScript web applications, namely GUARDIA. In [45], Musch *et al.* propose a *ScriptProtect* framework that instruments third-party JavaScript code to prevent the string-to-code conversions to protect the first-party origin from potential XSS attacks.

Although the works mentioned above provide possible solutions to secure JavaScript code in web pages, they aim to be used at the development phase. In contrast, our work aims at a tool that end-users can use to protect their privacy.

Browser Extensions. In addition to the research works on JavaScript security, there are many studies on browser extension design problems in recent years. Although these browser extensions provide a tool for web users to protect themselves from possible attacks, their technical approaches are different from our work, as we discuss in detail below.

In [3], the authors propose a new approach to blocking malicious third-party content achieved through an analysis of inclusion sequences constructed from an in-browser vantage point and implemented with EXCISION in a modified Chromium browser. Many experiments conducted in [4] simulate how advisement and analytics companies bypass ad-block extensions and how a bug with chrome web request help to block the extension from functioning. They base on the evaluation of the bug fix in different states to indicate that some companies still use WebSocket in troubleshooting like serving advertisements, infiltrating the DOM, and fingerprinting.

In addition, Barshir *et al.* [5] develop a methodology for detecting ad exchange information flow (both client- and server-side) by leveraging retargeted ads. Also, in [2], the authors propose a new approach (referred to as ORIGINTRACER) to determining the source of content modifications done

by over-privileged browser extensions or other third-party content. They indicate that this approach statistically improves the ability of users to recognize injected advertisements through the use of visual indicators while incurring a modest overhead. In [23], the authors present a framework for the verification of browser extensions to ensure they are secure and not over-privileged. Ter *et al.* in [58] also provide an approach to synthesizing abstract behavioral models from XPCOM interfaces. Their method invokes sequences of extensions obtained by the runtime interface invoking approach. This approach requires the preparatory implementation of a behavior monitoring system. It also proposes to define the vulnerable behavior sequence patterns. The design in this paper is used to guide the testing process adopted on sequence matching methods for detecting the security and reliability vulnerability of extensions.

In [14], Dhawan and Ganapathy present an in-browser system for tracking information-flow in order to analyze JavaScript-based browser extensions and identify their violations. In [65], they design *Expector*, a system for identifying chrome extensions that serve malicious ads, utilizing this system to detect extensions that inject ads or participate in malvertising. In [27], the authors propose a preliminary extension system design for protecting users' privacy from malicious extensions. Their method is also based on the idea of mandatory access control. Moreover, the authors in [57] study a comprehensive new model for extension security that aims to redress the shortcomings of existing extension mechanisms. Their proposed model works through the use of a logic-based specification language. The language describes fine-grained access control and data flow policies that govern an extension's privilege over web content. This model verifies extensions through analysis and includes a module for converting safe extensions into a form that allows for the execution of safety checks at runtime.

There are also other research studies focusing on the interaction between a browser extension and the security policy. For instance, in [25], Hausknecht *et al.* examine why browser extension's interaction with Content Security Policy is one reason for its slow adoption. They also classify three types of vulnerabilities arising from the interaction between extensions and CSP, proposing a solution, and providing a case study as a proof-of-concept. Besides, the authors in [56] explore browser extension discovery and the invasive nature of both websites and extensions. Roeshner *et al.* [53] and Mayer and Mitchell [34] investigate the web tracking. The former is with a focus on the technical implementation and functions of trackers, and the latter one is with the focusing on the policy debate over trackers. In [15], the authors explore the ability of browsers to be uniquely identified through fingerprinting. They implement a fingerprinting algorithm to estimate future success rates and discuss the privacy threats and implications posed by fingerprinting.

Likewise, in [46], the authors propose a new client-side JavaScript framework in order to check the integrity, origin, authentication, and risks of all JavaScript third-party resources, evaluating the solution through implementation. Moreover, in [36], the authors provide insight into the landscape of tracker blocker tools. Wills and Uzunoglu [64] explore the effectiveness of ad-blocking tools.

In [28], Iqbal *et al.* study a graph-based machine learning approach to blocking ads and tracker. They create a graph representation used to trace relationships between page contents (HTML structure, network requests, JavaScript behavior of the webpages) and (third-party) ads trackers. Besides, the authors in [29] investigate a framework that manages which security mechanism is active to cut down on high overheads created by outdated security solutions.

3 Proposed Approach

3.1 Overview

The objective of our method is to monitor the JavaScript code execution and stop a JavaScript operation if it violates a given policy. To this end, we intercept JavaScript operations, including property access and method calls, to enforce policies. In our enforcement code, we track the actual caller of the operation (we termed it `code origin`) and apply a specific policy for each code origin. Listing 1 illustrates the overview of our technical approach in pseudo-code, where we implement the interception within an anonymous function. Code within an anonymous function is to ensure that any code outside of the scope cannot access our code. In the next subsection, we describe our interception method, how we track the code origin, and how we enforce policies for each code origin.

Listing 1. The overview of our technical approach in pseudo-code.

```
(function(){
  let reference = original;
  let code_origin = getCodeOrigin(..);
  original = wrapper(){
    if (PolicyCheck(code_origin, reference_name, arguments))
      execute(reference);
  }//the wrapping
})();
```

3.2 JavaScript Interception

We review the JavaScript operations and categorize them into three different types: method calls, object creation and access, and property access. For each type of operation, we propose to intercept as follows.

Method Calls. Method calls are functions belonging to a global object. For example, `document.getElementById(..)` is a method call, which invokes the `getElementById` function of the `document` global object. For each method call, we wrap the original reference and its aliases by capturing any available prototype inheritance chain of the reference. We leverage a prior library [32] that implements these types of wrapping for our approach. In Sect. 4, we elaborate in more details of this approach.

Listing 2. Simplified code of mediating the access to document.write

```

var desc= Object.getOwnPropertyDescriptor(Document.prototype,
                                     "cookie");
//assert the desc object and its prototype chain
Object.defineProperty(document, "cookie", {get: function(){
    var code_origin = getCodeOrigin(new Error().stack);
    if (originAllowed(code_origin,"cookie")) {
        setOriginSourceRead(code_origin,"cookie");
        return desc.get.call(document);
    }
    return;
},
set: function(val){ desc.set.call(document, val); },
enumerable : false,
configurable : false
});

```

Property Access. A property is a field of an object that can be accessed (read or write) by JavaScript code. As these properties, for example, `document.cookie` may contain sensitive data of users, we need to mediate the access to them. To this end, we leverage the `Object.defineProperty(..)` API (standardized in ECMAScript 5 and still supported in later versions [16]) to define the handler functions whenever a property is read (get) or write (set). Simplified code (for brevity) in Listing 2 illustrates this mediation process for the `document.cookie` property.

Object Creation and Access. There are several JavaScript operations that are based on object creation. For example, the `"new Image(..)"` operation creates a new image object (`HTMLImageElement`) that can be appended to a web page to display an image. In our observations and experiments, in some cases, although this object creation operation returns the same object type as an equivalent function call (for example, `new Image(..)` (object creation) and `document.createElement("img")` (method call) both return an `HTMLImageElement` object), wrapping the object prototype alone, e.g., `HTMLImageElement.prototype`, does not capture the object creation operation and therefore cannot control the property access to this newly created object. For this reason, we manually create a wrapper class for each original class and mediate the access to the wrapper class using the Proxy object, standardized in ECMAScript 6 [16] and define a specific policy each access operation. Listing 3 demonstrates an implementation example for mediating the `"new Image(..)"` operation.

Listing 3. Simplified code of mediating the access to new Image(..)

```

var imgPolicy = {
  get: function(obj, prop) { /* policies for get */},
  set: function(obj, prop, value) { /* policies for set */}
}

```

```

};
var OriginalImage = Image;
class ImageWrapper {
    constructor(height, width) {
        var imgObject = new OriginalImage(height, width);
        imgObject = new Proxy(imgObject, imgPolicy);
        return imgObject;
    }
}
Image = ImageWrapper;

```

3.3 Tracking the Source (origin) of the Code

A web page nowadays includes typically JavaScript code that can be sourced from different servers [47]. Browsers enforce the same-origin policy to ensure that JavaScript code comes from one origin (specified by the protocol, host, and port of an URL) cannot access the credentials of other origins [41]. However, the same-origin policy treats all code included in a web page as the same origin even though the code comes from an external web site [41, 45, 54]. Several prior works, e.g., [20, 48], recognize this security issue and propose new approaches to enforcing different policies for JavaScript within a web page but is sourced from external servers. For example, FlashJax [48] provides a new JavaScript API to load and execute JavaScript under a principal and enforces principal-based policies for external JavaScript code. However, FlashJax requires web developers to use this new JavaScript API on the web page; thus, it is not applicable for a browser-based tool. In our work, we leverage the call stack of JavaScript language to trace the source of the code, i.e., from which origin (scheme, host, port) the code is included. We term this source of the code as **code origin**. We note that our code origin concept is different from the web origin term usually used in the same-origin policy [41]. To track the code origin at runtime when we want to enforce a policy, we create a new **Error** object (`new Error().stack`) to get its stack and trace the top of the stack to get the code origin (We notice that a similar approach has been introduced in [45]; however, our implementation is independent and concurrently with that work). Our experiments demonstrate that this code origin tracing method can keep track of exactly where a JavaScript operation is invoked. The usage of this approach has already been illustrated in Listing 2.

3.4 Context-Aware and Code Origin-Based Policies

As we can trace the origin of the code, we wish to define and enforce more precise policies that can detect and prevent possible information leakage channels. The ideal method is to encode information flow policies. However, this approach requires new language constructors with new browser implementation [26]. In this work, we propose to control the information flow at the endpoints. This method means that we monitor the operations that read from sensitive data

sources or send information to the outside of a browser. When a sending operation is called, we check whether sensitive information was read. As we can keep track of the code origin, we can enforce policies that depend on the code origin. We can also consult the user and let the user decide to allow or deny a data send operation that might be suspicious.

4 Implementation

We realize our proposed method in a JavaScript library and deploy it into a browser extension to develop a self-protecting tool for web users. In this section, we describe our JavaScript library, the concrete policies we have implemented, and how we develop a browser extension that leverages our library.

4.1 JavaScript Monitoring Library

We implement our method by developing a JavaScript library to intercept JavaScript operations that might cause privacy leakage. Our library must be executed first in a web page to ensure that it can keep the original references to the intercepted operations.

We divide our interception implementation into two types: (i) data source access: operations can get sensitive data; (ii) data sink channels: operations can send data to the outside of a browser. We detail each type below.

Data Source Access. The data sources in a web page include the cookie, HTML local storage, browsing history, location, the values of form elements, and the web page contents [34, 49]. These data can be accessed by method calls or property read. We implement each type of data source read according to the approach presented in Sect. 3.2. For each operation, we check whether the code origin is allowed to read the data. If allowed, we mark that the code origin has read the data by update the corresponding state. The code in Listing 2 (in Sect. 3.2) illustrates this interception implementation method for a property access of a data source (cookie). In Listing 4, we demonstrate the interception implementation of a data source using a method call (`localStorage.getItem(...)`). In this code, `monitorMethod` is a wrapping interface implementing the wrapping approach for method calls presented in Sect. 3. We list the common data source read operations implemented in our library in Table 1 (`document.getElement*` in the first row stand for `getElementById`, `getElementsByTagName`, `getElementsByClassName`).

Listing 4. Interception implementation of a method call accessing a data source

```
function localStorage_getItem_policy(args, proceed, obj) {
  var itemID = args[0];
  var code_origin = getCodeOrigin(new Error().stack);
  if (originAllowed(code_origin, "localStorage",
                    "getItem", itemID)) {
```

```

    setOriginSourceRead(code_origin, "localStorage");
    return proceed(); //execute the method call
}
return; //supress the method call
}
monitorMethod(localStorage, "getItem",
    localStorage_getItem_policy);

```

Table 1. List of operations that can access users' data

Operation	Type	Data source
<code>document.getElementById*</code>	Method call	Page contents
<code>localStorage.getItem</code>	Method call	Local storage
<code>document.cookie</code>	Property access	Cookies
<code>window.history</code>	Property access	Browsing history
<code>navigator.geolocation.getCurrentPosition</code>	Method call	Location

Data Sink Channels. We consider the channels where data can be sent from a browser to outside. We do not consider the attack scenarios that are aware by end-users such as redirection as this is out of the scope of our work. Generally speaking, these data channels are HTTP requests sent from a browser [41]. JavaScript operations that can generate a HTTP request include assigning an URL source to an object such as `Frame`, `IFrame`, `Image`, `Script`, and `Form`, `Ajax`, and `WebSocket`. We note that `WebSocket` is not an actual HTTP request yet allows browsers to open two-way interactive communication to a server to send and receive messages asynchronously [42]. In the past, tracking companies have leverage `WebSocket` to circumvent the blocking mechanism in ad blockers [4]. We intercept these channels using the approach for object creation presented in Sect. 3.2. Also, we intercept method calls, for example, `document.createElement(..)` that create these objects and enforce the same policy as the object creation interception. We stop a data send operation if it violates a policy.

4.2 MyWebGuard - The Privacy Protection Tool in Browsers

In principle, our library can be deployed either at the server or client-side. However, as we aim to develop a tool for web users, we opt to implement a browser extension using our library. As discussed earlier, our method requires that the interception library run first before a web page is loaded. To this end, we put our JavaScript library code within the `innerHTML` property of a script object and append it to the current page. We perform experiments to confirm that our code is executed before any other code in a web page. Listing 5 depicts this inclusion method of our library.

Listing 5. Injection of our code in a browser extension

```

var mywebguard = document.createElement("script");
mywebguard.innerHTML = `
(function() {
  //the full JavaScript interception library here
  //...
})();`;
document.documentElement.appendChild(mywebguard);

```

Implemented as a JavaScript library, our method should be able to be deployed in any browser extension. In this implementation prototype, we develop our MyWebGuard extension and evaluate it in Chromium family, including Google Chrome, Chromium, and Brave. In the future, we explore to deploy our library in other major browsers.

5 Evaluation

In this section, we report the evaluation of our method implemented in a browser extension. We first create a test suite that contains a web page for the first-party and a script file for the third-party. We simulate attack scenarios to evaluate the effectiveness of our method.

To evaluate our method, we installed our MyWebGuard extension in Chromium family browsers. We perform the evaluations on a Dell Inspiron 3521 machine with CPU 2127U @ 1.90 GHzx2, 8 GB memory, Ubuntu 18.04.2 LTS. We test our extension in Google Chrome (Version 76.0.3809.100 64-bit), Chromium (Version 76.0.3809.87 64-bit) and Brave (version 0.65.121) browsers.

5.1 Privacy Leakage Detection and Prevention

We create a test web site and develop first-party code to perform operations that we intercept. We include a third-party JavaScript code that performs similar operations. Our experiments demonstrate that potential information leakage to the first-party or third-party server is captured by our method. Figure 1 shows two test cases that leak data detected by our browser extension.

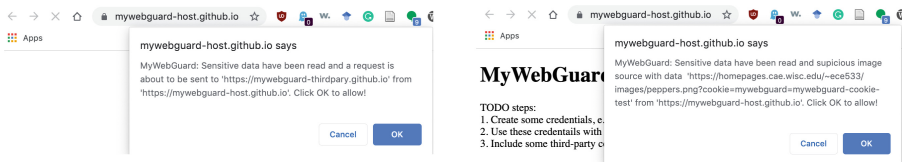


Fig. 1. MyWebGuard can detect and prevent potential information leakage that is ignored by Ghostery and uBlock Origin extensions

Interestingly enough, these simulated data leakage channels are not detected by leading browser extensions such as Ghostery, and uOrigin Block, demonstrated in Fig. 2. This evidence reflects and confirms our observation and motivation discussed in the introduction.

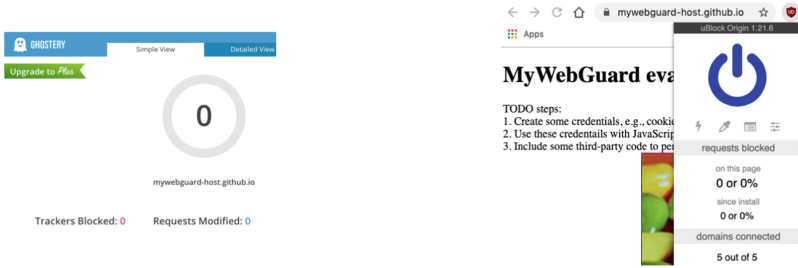


Fig. 2. Ghostery and uBlock Origin extensions do not detect the tracking requests on our test suite site

In an intensive mode with all alert messages turned on, MyWebGuard can detect potential leakage sent to external origins and warn the user in real-world websites. For example, Fig. 3 shows the case that our MyWebGuard extension detects a cross-origin data leakage on Youtube.

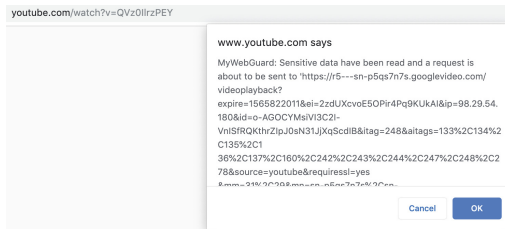


Fig. 3. MyWebGuard extension detects cross-origin data leakage on Youtube

5.2 Compatibility and Load Time Overhead

We turn on our extension with alert messages disabled in the tested browsers (Google Chrome, Chromium, and Brave) and load regular websites to test whether these sites are loaded as usual. We do not notice any issues when loading the websites with our extension. Figure 4 demonstrates that a Youtube video can play when our extension is turned on in Chromium.

We also measure if our extension slowdowns the load time of actual websites. We test this by getting the load time numbers in the Network debug in a Chromium browser for top websites in Alexa and several Vietnamese news, with and without our extension. We disable alert and debug messages and load each website ten times to get the average numbers. Figure 5 illustrates the load

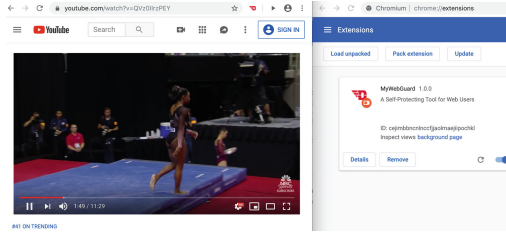


Fig. 4. Youtube video can be played functionally with MyWebGuard extension in Chromium

time overhead of our extension over ten popular websites. As we can see from this graph, our MyWebGuard tool does not pose a great slow down in load time. The average slow down ratio is 1.33, but in some cases, the load time is even faster with our extension. For example, the slow down for <https://ebay.com> and <https://vnexpress.net> is 0.95 and 0.99, respectively. This improvement might be due to the fact that some wrapped operations can be executed faster than the original ones, as reported in the literature [48]. We note that we test <https://facebook.com> and <https://mail.google.com> in logged-in sessions because the content of these web pages are too light to test without logged-in.

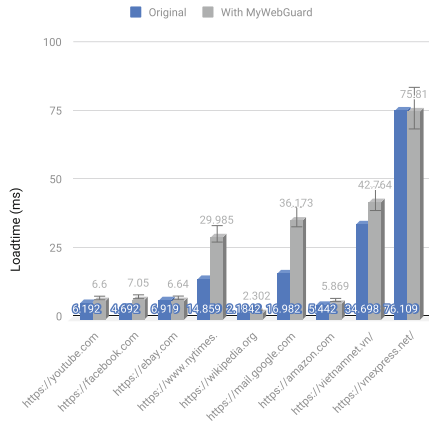


Fig. 5. Load time overhead of MyWebGuard in actual websites

6 Conclusions and Future Work

In this paper, we have presented MyWebGuard, a user-centric tool to protect the privacy of web users. MyWebGuard can enforce context-aware and code origin-based policies to prevent privacy leakage channels that cannot be captured by existing tools in the industry. The evaluations of our prototype implementation evidence that our novel enforcement method can effectively protect the privacy of web users yet pose lightweight overhead on popular real-world websites.

In the future, we plan to extend and refine the security policies as well as the enforcement mechanism to allow end-users can customize their privacy preferences. We will perform large-scale evaluations of our tool on top websites and investigate whether our tool interferes with co-existing browser extensions. We also plan to leverage machine learning to produce practical policies that protect users but do not break legitimate third-party code. Our objective is to provide a robust and mature tool that web users can use in the real-world to protect their privacy. We also want to explore the effectiveness of our approach when built-in into a browser.

Acknowledgment. The authors wish to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Agarwal, L., Shrivastava, N., Jaiswal, S., Panjwani, S.: Do not embarrass: re-examining user concerns for online tracking and advertising. In: Proceedings of the Ninth Symposium on Usable Privacy and Security, SOUPS 2013, pp. 8:1–16. ACM (2013)
2. Arshad, S., Kharraz, A., Robertson, W.: Identifying extension-based ad injection via fine-grained web content provenance. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 415–436. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_19
3. Arshad, S., Kharraz, A., Robertson, W.: Include me out: in-browser detection of malicious third-party content inclusions. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 441–459. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_26
4. Bashir, M.A., Arshad, S., Kirda, E., Robertson, W., Wilson, C.: How tracking companies circumvented ad blockers using Websockets. In: Proceedings of the Internet Measurement Conference 2018, pp. 471–477. ACM (2018)
5. Bashir, M.A., Arshad, S., Robertson, W., Wilson, C.: Tracing information flows between ad exchanges using retargeted ads. In: 25th USENIX Security Symposium, USENIX Security 16, pp. 481–496 (2016)
6. Batt, S.: What is “do not track” and does it protect your privacy?, August 2019 <https://www.makeuseof.com/tag/not-track-actually-work/>
7. Burt, A.: Privacy and cybersecurity are converging. here’s why that matters for people and for companies, January 2019. <https://hbr.org/2019/01/privacy-and-cybersecurity-are-converging-heres-why-that-matters-for-people-and-for-companies>. Accessed 13 Aug 2019
8. Caleb: Ranked: Security and privacy for the most popular web browsers, March 2019. <https://www.expressvpn.com/blog/best-browsers-for-privacy/>
9. Chanchary, F., Chiasson, S.: User perceptions of sharing, advertising, and tracking. In: Proceedings of the Eleventh Symposium On Usable Privacy and Security, SOUPS 2015, pp. 53–67 (2015)
10. Chromium Blog: Improving privacy and security on the web, May 2019. <https://blog.chromium.org/2019/05/improving-privacy-and-security-on-web.html>
11. Chudnov, A., Naumann, D.A.: Inlined information flow monitoring for JavaScript. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 629–643. ACM (2015)

12. Crockford, D.: ADsafe - Making JavaScript Safe for Advertising (2007). <http://www.adsafe.org>. Accessed 11 Aug 2019
13. devlin@chromium.org: Manifest V3, December 2018. <https://docs.google.com/document/d/1nPu6Wy4LWR66EFLeYInl3NzzhHzc-qnk4w4PX-0XMw8/edit#heading=h.xgjl2srtytjt>. Accessed 14 Aug 2019
14. Dhawan, M., Ganapathy, V.: Analyzing information flow in JavaScript-based browser extensions. In: 2009 Annual Computer Security Applications Conference, pp. 382–391. IEEE (2009)
15. Eckersley, P.: How unique is your web browser? In: Atallah, M.J., Hopper, N.J. (eds.) PETS 2010. LNCS, vol. 6205, pp. 1–18. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14527-8_1
16. Ecma International: ECMAScript 2015 Language Specification ECMA-262 6th Edition, June 2015. <https://www.ecma-international.org/ecma-262/6.0/>. Accessed 14 Aug 2019
17. Englehardt, S., Narayanan, A.: Online tracking: a 1-million-site measurement and analysis. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 1388–1401. ACM (2016)
18. Finifter, M., Weinberger, J., Barth, A.: Preventing capability leaks in secure JavaScript subsets. In: NDSS (2010)
19. Fredrikson, M., Livshits, B.: Repriv: re-imagining content personalization and in-browser privacy. In: 2011 IEEE Symposium on Security and Privacy, pp. 131–146. IEEE (2011)
20. Georgiev, M., Jana, S., Shmatikov, V.: Rethinking security of web-based system applications. In: Proceedings of the 24th International Conference on World Wide Web, pp. 366–376. International World Wide Web Conferences Steering Committee (2015)
21. Google Caja: Compiler for making third-party HTML, CSS, and JavaScript safe for embedding (2007). <https://developers.google.com/caja/>. Accessed 5 Aug 2019
22. Google Chrome: chrome.webRequest. <https://developer.chrome.com/extensions/webRequest>. Accessed 14 Aug 2019
23. Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: 2011 IEEE symposium on security and privacy, pp. 115–130. IEEE (2011)
24. Guha, S., Cheng, B., Francis, P.: Privad: practical privacy in online advertising. In: USENIX Conference on Networked Systems Design and Implementation, pp. 169–182 (2011)
25. Hausknecht, D., Magazinius, J., Sabelfeld, A.: May I? - Content security policy endorsement for browser extensions. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) DIMVA 2015. LNCS, vol. 9148, pp. 261–281. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20550-2_14
26. Hedin, D., Bello, L., Sabelfeld, A.: Information-flow security for JavaScript and its APIs. *J. Comput. Secur.* **24**(2), 181–234 (2016)
27. Heule, S., Rifkin, D., Russo, A., Stefan, D.: The most dangerous code in the browser. In: 15th Workshop on Hot Topics in Operating Systems (HotOS XV) (2015)
28. Iqbal, U., Snyder, P., Zhu, S., Livshits, B., Qian, Z., Shafiq, Z.: AdGraph: a graph-based approach to ad and tracker blocking. In: IEEE Symposium on Security and Privacy, May 2020
29. Katz, O., Livshits, B.: Toward an evidence-based design for reactive security policies and mechanisms. arXiv preprint [arXiv:1802.08915](https://arxiv.org/abs/1802.08915) (2018)

30. Leon, P.G., et al.: What matters to users?: factors that affect users' willingness to share information with online advertisers. In: Proceedings of the Ninth Symposium on Usable Privacy and Security, p. 7. ACM (2013)
31. Maffeis, S., Taly, A.: Language-based isolation of untrusted Javascript. In: 2009 22nd IEEE Computer Security Foundations Symposium, pp. 77–91. IEEE (2009)
32. Magazinius, J., Phung, P.H., Sands, D.: Safe wrappers and sane policies for self protecting JavaScript. In: Proceedings of the 15th Nordic Conference in Secure IT Systems NordSec, pp. 239–255, October 2010
33. Mathur, A., Vitak, J., Narayanan, A., Chetty, M.: Characterizing the use of browser-based blocking extensions to prevent online tracking. In: Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, pp. 103–116 (2018)
34. Mayer, J.R., Mitchell, J.C.: Third-party web tracking: policy and technology. In: 2012 IEEE Symposium on Security and Privacy, pp. 413–427. IEEE (2012)
35. McDonald, A.M., Cranor, L.F.: Americans' attitudes about internet behavioral advertising practices. In: Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society, pp. 63–72. ACM (2010)
36. Merzdovnik, G., et al.: Block me if you can: a large-scale study of tracker-blocking tools. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P, pp. 319–333. IEEE (2017)
37. Meyerovich, L.A., Livshits, B.: ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In: 2010 IEEE Symposium on Security and Privacy, pp. 481–496. IEEE (2010)
38. Microsoft Edge: Security and privacy group policies (2018). <https://docs.microsoft.com/en-us/microsoft-edge/deploy/group-policies/security-privacy-management-gp>. Accessed 14 Aug 2019
39. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Safe active content in sanitized JavaScript. Tech. rep. Google Inc. (2008)
40. Mozilla: webRequest. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest>. Accessed 14 Aug 2019
41. Mozilla Developer Network: Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed 14 Aug 2019
42. Mozilla Developer Network: The WebSocket API (WebSockets), April 2019. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
43. Mozilla Developer Network: What are extensions? March 2019. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/What_are_WebExtensions
44. Mozilla Security Blog: Privacy archives, August 2019. <https://blog.mozilla.org/security/category/privacy/>. Accessed 14 Aug 2019
45. Musch, M., Steffens, M., Roth, S., Stock, B., Johns, M.: ScriptProtect: mitigating unsafe third-party javascript practices, pp. 391–402 (2019)
46. Nakhaei, K., Ansari, E., Ansari, F.: JSSignature: eliminating third-party-hosted JavaScript infection threats using digital signatures. arXiv preprint [arXiv:1812.03939](https://arxiv.org/abs/1812.03939) (2018)
47. Nikiforakis, N., et al.: You are what you include: large-scale evaluation of remote JavaScript inclusions. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 736–747. ACM (2012)
48. Phung, P.H., Monshizadeh, M., Sridhar, M., Hamlen, K.W., Venkatakrishnan, V.: Between worlds: securing mixed JavaScript/ActionScript multi-party web content. IEEE Trans. Dependable Secure Comput. TDSC **12**(4), 443–457 (2015). <https://doi.org/10.1109/TDSC.2014.2355847>

49. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting JavaScript. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (AsiaCCS), pp. 47–60, March 2009
50. Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: ADSafety: type-based verification of JavaScript sandboxing. In: Proceedings of the 20th USENIX Conference on Security. SEC 2011, USENIX Association (2011)
51. Pupo, A.L.S., Nicolay, J., Boix, E.G.: GUARDIA: specification and enforcement of Javascript security policies without VM modifications. In: The 15th International Conference on Managed Languages & Runtimes, pp. 17:1–17:10. ACM (2018)
52. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: BrowserShield: vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web (TWEB)* 1(3), 11 (2007)
53. Roesner, F., Kohno, T., Wetherall, D.: Detecting and defending against third-party tracking on the web. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p. 12. USENIX Association (2012)
54. Schwenk, J., Niemietz, M., Mainka, C.: Same-origin policy: evaluation in modern browsers. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 713–727. USENIX Association, Vancouver, August 2017
55. Siddiqui, A.: Google’s Manifest V3 will change how ad blocking Chrome extensions work: is it to cripple them, or is it for security? June 2019. <https://www.xda-developers.com/google-chrome-manifest-v3-ad-blocker-extension-api/>
56. Sjösten, A., Van Acker, S., Sabelfeld, A.: Discovering browser extensions via web accessible resources. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 329–336. ACM (2017)
57. Swamy, N., Livshits, B., Guha, A., Fredrikson, M.J.: Programming, verifying, visualizing, and deploying browser extensions with fine-grained security policies, March 2015, US Patent 8,978,106
58. Ter Louw, M., Lim, J.S., Venkatakrisnan, V.N.: Enhancing web browser security against malware extensions. *J. Comput. Virol.* 4(3), 179–195 (2008)
59. Ur, B., Leon, P.G., Cranor, L.F., Shay, R., Wang, Y.: Smart, useful, scary, creepy: perceptions of online behavioral advertising. In: Proceedings of the Eighth Symposium On Usable Privacy and Security, SOUPS 2012, p. 4. ACM (2012)
60. W3C: Content security policy (2018). <https://www.w3.org/TR/CSP/>
61. W3C: Tracking-Preference Expression (DNT), January 2019. <https://www.w3.org/TR/tracking-dnt/>
62. W3Techs.com: Usage Statistics of JavaScript as Client-side Programming Language on Websites, August 2019. <https://w3techs.com/technologies/details/cp-javascript/all/all>
63. Weissbacher, M., Lauinger, T., Robertson, W.: Why is CSP failing? trends and challenges in CSP adoption. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 212–233. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11379-1_11
64. Wills, C.E., Uzunoglu, D.C.: What ad blockers are (and are not) doing. In: 2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), pp. 72–77. IEEE (2016)
65. Xing, X., et al.: Understanding malvertising through ad-injecting browser extensions. In: Proceedings of the 24th International Conference on World Wide Web, pp. 1286–1295 (2015). International World Wide Web Conferences Steering Committee