Towards Improving Browser Extension Permission Management and User Awareness

Said Marouf and Mohamed Shehab
Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC, USA
{smarouf, mshehab}@uncc.edu

***Abstract*—Browsers have become the de-facto platform for users and their online presence. They have also become a rich environment for 3rd party extensions that enrich the user browsing experience by extending upon the browser's functionalities. Protecting user privacy against malicious or vulnerable extensions is an important task performed by modern browser platforms such as Google Chrome and Safari. To do so, these platforms adopt a per-extension permission model, where each extension is given a set of permissions based on its requirements. These models suffer from coarse-grained access controls and insufficient user awareness. In this paper we implement a runtime framework as a browser extension called REM. REM monitors the accesses made by 3rd party Chrome extensions, informs users of the accesses, and allows them to customize the permissions given to extensions. The custom permission settings are enforced by the framework at runtime. We evaluated our framework on popular Chrome extensions & were successful in monitoring and controlling their accesses with little overhead. We also conducted a user study to evaluate the effectiveness of REM compared to current standard methods.**

***Index Terms*—browser security, browser extensions, security**

I. Introduction

Today's online activities such as social networking, and banking, have increased the user online presence and made the browser their main portal. Users are increasingly enriching their browsing experience with 3rd party applications that provide new functionalities and improve upon existing ones. 3rd party browser extensions are popularly used by millions of users [1], [2], especially with their wide availability on portals such as Google's Chrome Web Store. Regardless of the popularity and benefits of 3rd party extensions, they could potentially threat the privacy of their users. This could be due to malicious extension developers, or vulnerable extensions written by developers who lack secure coding skills. This lead platforms such as Google Chrome to introducing permission models that control the accesses by 3rd party extensions, especially those regarding sensitive user data. These models allow developers to declare permissions their extensions require. Extension users on the other hand are responsible for making their own access control decisions on the requested permissions.

Existing browser permission models suffer from limitations when it comes to protecting user privacy against 3rd party extensions. Such limitations involve insufficient access control techniques, & limited user awareness. Some browsers provide an "Incognito" mode that disables extensions by default. In this paper, we investigate user privacy under Google Chrome's permission model, in addition to potential privacy threats. We also propose a runtime framework that improves upon the existing Chrome model and incorporates the following: *1) A Runtime Monitoring API*, *2) Fine-grained Runtime Access Control*, *3) REM*: a Chrome extension that implements our proposed framework. Finally, we conduct a user study that evaluates our Chrome extension "REM" and focuses on measuring REM's effect on user awareness towards extension permissions.

II. Background

Users are asked to respond to various permission requests on a daily basis. Such requests can be at the time they install new applications, or while trying to access certain features of an application they already use. For example, users are prompted with permission requests whenever they install a new Facebook application, or whenever an iPhone application tries to use a device's GPS for the first time. Overall, permission requests can happen at install-time or runtime. In this paper we focus on the permission model for Google Chrome extensions, which can be considered as a hybrid model, that is, extensions request permissions at install time, but also have the ability to request optional permissions after installation.

A. Chrome Extensions

Third party browser extensions are widely used within major browsers such as Firefox, Chrome, and Safari [1], [2]. Users can change their browsing experience by adding new functionalities or modifying the core browser functionalities. Chrome extensions are built using a mix of required and optional components. Specifically, a required `manifest.json`, at least one `html` file (`background.html` or `popup.html`), and other additional resources such as JavaScript files, images, and other HTML files.

Manifest: The `manifest.json` file is a required component for each extension, and provides information on an extension's properties, requested permissions, and other attributes. In this paper, we focus on the `permissions`, `plugins`, and `content_scripts` properties within the

COLLABORATECOM 2012, October 14-17, Pittsburgh, United States

DOI 10.4108/icst.collaboratecom.2012.250642

manifest. These are properties related to the privacy of the user when using third party extensions.

Background Page: An optional HTML page that many extensions use for managing background activities. This is used by extensions that need to stay active at all times or be able to perform continuous tasks. Our proposed framework targets background pages when adapting third party extensions to our model.

Content Scripts: These are scripts that run within the context of a webpage that extensions want to interact with. That is, the content script can read and modify a webpage and pass messages back to its parent extension. An example extension that uses content scripts is the Google Dictionary extension which shows a popup with the description of a selected word within a webpage. Extensions declare the hosts targeted by their content scripts within the `manifest.json`. Note that extensions are also able to programmatically inject custom scripts into webpages using the `chrome.tabs.executeScript` API.

III. Chrome Extension Permissions

Third party Chrome extension developers are able to declare permissions needed by their extensions to fulfill certain functionalities, and to access certain Chrome APIs. Such permissions can be declared as required or optional using the `permissions` and `optional_permissions` manifest properties respectively. For example, an extension might request access to browser cookies, or a user's browsing history in order to interact with their associated Chrome APIs. The set of such possible permissions are defined by Google within the Chrome extension API documentation. Developers can also declare permissions as optional, which is ideal for permissions not required immediately by extensions. Additional permissions can also be requested by an extension when updated.

A. Permissions and Chrome APIs

Once an extension acquires its requested permissions, it can access the Chrome APIs associated with each permission, i.e, certain Chrome APIs require certain permissions to execute successfully. For example, the `chrome.cookies.get` API call requires the `cookies` permission. We look at each requested permission, and find all the reachable API calls an extension can perform, which allows us to precisely monitor all potential extension accesses, as explained in our proposed framework in Section V. The full permission to API mappings were generated by scanning the Chrome extension documentation, specifically the manifest permissions and their associated `chrome` modules. By mapping each permission to a set of associated API calls, we can control and monitor an extension's specific accesses. The exception to this rule is any extension using an NPAPI plugin, which allows for native code execution outside of the context of the Chrome browser. That is, NPAPI accesses do not occur through the Chrome APIs.

B. User Awareness

Users are warned about some of the permissions that are requested at installation time, and have the option to either continue installing an extension with the requested permissions, or cancel the installation process. Warnings are also shown to users if a certain extension is updated and requests additional permissions, or if an optional permission is being requested. Note that not all permissions trigger a warning message. Such permissions will be granted to an extension without the user's explicit approval. An example of such permissions is `cookies`. We think the rationale behind this is that these permissions rely on other requested permissions that do trigger warnings. For example, an extension that requests the `cookies` permission can only access cookies for the hosts it has access to. The list of hosts that can be accessed by a certain extension are listed within its manifest file as part of the permissions attribute, and are shown to the user at install time. The caveat here is that not all users will presume giving access to a certain host could also lead to granting access to its cookies. For example, if a user grants access to `<all_urls>` (all urls), this could potentially mean access to all cookies in the user's browser. Another issue involves warnings that

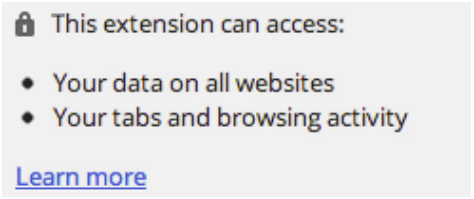


Fig. 1. Permission Details in the Standard method

do not reflect a precise description of what is being granted to an extension. For example, an extension that requests the `history` permission will trigger a warning that says "It can access: Your browsing history", which could potentially be misinterpreted as the list of all URLs a user has visited. But the matter of fact is that the `history` permission also provides an extension with information regarding a user's browsing behavior, e.g. how the user reached a certain website (by typing the url, clicking a link, via a bookmark, etc.), the time they visited a website, and the number of visits too. Such information can be valuable to third parties and could potentially be used for undesired purposes from a user's point of view. In our proposed framework, we provide users with detailed information and feedback on the permissions and accesses granted to an extension as seen in Figure 5. Currently, the "Standard" method for discovering an extension's permissions is to visit its page on the Chrome Web Store and looking at the details tab as seen in Figure 1. From there, users have the option to discover more about the permissions requested by visiting yet another webpage. In Section VII we show that our proposed extension REM performs better in increasing user awareness and understanding of an extension's permissions.

Extension permissions sometimes rely on other permissions, i.e. it is not sufficient for an extension to request one

permission without the other. Hence, certain functionalities within an extension will require a chain of permissions to execute successfully. A popular permission requested by extensions is the `host` permission, which is declared within the manifest as a match pattern. The pattern dictates the hosts that are accessible by extensions. Example patterns include: `http://*/*` (all hosts using the http scheme), `http://example.com/foo.html` which matches that specific url, and `<all_urls>` which matches all urls. The importance of the `host` permission emerges when extensions use other permissions such as the `cookies` or `tabs`. For example, an extension may request `cookies` permission and assume it can read all cookies using the `chrome.cookies.getAll` API. This isn't true, unless the extension requested a `host` permission that covers all URLs associated with the desired cookies.

IV. User Privacy and Threats

Users have widely adopted browser extensions and have become acclimated to using them on a regular basis. With this wide spread of extensions, especially ones developed by third parties, the threats to user privacy have increased [3], [4], [5], [6]. The permission model adopted by Google Chrome does provide some means for controlling the permissions given to extensions, but there are still areas that can be improved to provide for better privacy and protection against potential threats.

A. Threats

Extensions with excessive permissions represent a higher threat to user privacy, especially those that are poorly written and include security vulnerabilities. Excessive permissions are those that are deemed inappropriate or unnecessary in certain privacy related scenarios. For example, granting a `host` permission of `<all_urls>` to a Twitter client extension could be deemed excessive, as it most likely would only require access to `http://*.twitter.com/*`. In the following, we discuss some potential threats when extensions gain excessive Chrome permissions.

Host Pattern	Occurrences (100)
`<all_urls>`	5
`*://*/*`	4
`https://*/*`	38
`http://*/*`	46
Wild Card Subdomain	18
Specific Host	12

Fig. 2. Host permission patterns requested by the top 100 rated extensions

Host Permissions The `host` permission is a popular permission requested by third party extensions and is declared as a match pattern within the extension's manifest. The match pattern represents the webpages extensions would like to access, which could range from a specific webpage (by specifying a specific URL) to all webpages with a schema of `http`, `https`, `file`, or `ftp` (Using the `<all_urls>`). Figure 2 shows the requested host permission patterns requested by the top hundred rated extensions on the Chrome Web Store. The most popular patterns requested were the `http://*/*` and `https://*/*` patterns. Note that the occurrences of match patterns do not sum up to 100, that is because extensions can declare multiple patterns. Extensions with excessive host permissions could potentially succeed in performing attacks on user privacy, especially when combined with other permissions such as the `tabs` or `cookies` permission. With the `tabs` permission, extensions are able to programmatically execute their own custom JavaScript using the `chrome.tabs.executeScript` API. Such scripts are allowed to run on webpages that satisfy the extension's `host` permission. Hence, with an excessive `host` permission, custom scripts are executed on a wider range of webpages. The threats on user privacy arise when custom scripts are vulnerable to attacks such as Cross Site Scripting, that is, a script could potentially execute malicious code embedded within webpages visited by the user. Such a scenario would allow the malicious code to perform with the privileges of the compromised extension. For example, malicious code could access all cookies accessible to a compromised extension that has `cookies` permission. Limiting the `host` permission to a smaller subset of webpages would decrease the attack surface.

The `cookies` permission combined with excessive `host` permissions could also introduce threats to user privacy. Access to cookies is based on the `host` permission an extension has, that is, access is allowed to any cookie that belongs to a host within the match pattern declared by the `host` permission. Hence, a match pattern of `<all_urls>` potentially means access to all user cookies. Extensions could abuse their `host` permission and access user cookies for malicious reasons such as hijacking a user's online session. Another threat scenario involves vulnerable extensions that have the `cookie` permission. Such extensions, if attacked, could elevate the privileges of malicious code and allow it access to user cookies and other reachable resources.

The dependencies between the `host` and both `tabs` and `cookie` permissions makes it important to monitor and control the specific accesses made by extensions, especially when dealing with excessive `host` permissions such as `http://*/*` or `<all_urls>`. The rationale is that extensions may need different host permissions for different types of accesses. For example, executing a script using the `tabs.executeScript` API may require certain `host` permissions, whereas reading cookies via the `chrome.cookies.get` API may require different ones. Currently, the same `host` permission is used for both purposes, which leads to unnecessary privileges and potentially unwanted accesses.

Tabs Permission The `tabs` permission gives extensions access to the browser's windows and tabs within each open window. Extensions are able to access `Tab` objects, which contain information on the tab returned such as the associated

URL. Hence, extensions with `tabs` permission have access to all URLs a user visits. Note that the `tabs` permission is not dependent on the `host` permission with the exception of content script execution, hence, Chrome does not prevent access to tab URLs that are not within the `host` match pattern. With access to all URLs, a malicious extension can directly analyze any URL and its query attributes, and potentially extract important information such as session IDs and OAuth request tokens. Such information can be used in compromising the user's privacy [7].

Another drawback of not bounding the `tabs` permission, is that it undermines the `history` permissions defined by Chrome. That is, extensions can generate their own history repository by keeping track of all URLs users visit. Note that the `history` permission provides additional accesses such as the methodology of reaching a certain webpage (e.g. was a URL typed, clicked, etc.), hence we only consider this a partial undermining. We improve upon the `tabs` permission within our proposed framework by allowing users to customize the URLs accessible by APIs associated to the `tabs` permission.

Other Permissions Other Chrome permissions such as the `history` & `bookmarks` permission could also be used to gain access to URL data, hence potentially executing malicious attacks using extracted session IDs or OAuth request tokens. Such attacks may frequently fail given history and bookmark URLs are potentially old, hence contain outdated information regarding a user's session or request token. Note that both these permissions are not bounded by the `host` permissions. We also improve upon this within our framework.

B. Intrusiveness

Third party extensions that request excessive permissions can be quite intrusive. This is mainly due to the relatively course-grain nature of Chrome permissions. For example, extensions with the `tabs` permission are able to track all URLs a user visits, which in many cases is undesirable, especially in scenarios where users visit webpages of highly confidential matter, such as financial or health related webpages. The `tabs` permission also gives extensions access to the DOM, which gives it the ability to read and write data within the DOM. Such data may be highly confidential. For example, an extension with `tabs` permission can easily detect if a user has visited *https://online.wellsfargo.com/* and extract the user's balance. With additional permissions, the extension could even pass it back to a remote server. Such scenarios show the importance of giving the user the necessary controls over which webpages certain extensions have access to. Other permissions such as `history` and `bookmarks` could also reveal the browsing behaviors of users. We believe users should have the option to control the accesses associated with these permissions. With the potential threats and lack of sufficient user awareness within the Chrome extension permission model, we propose a runtime framework that monitors and informs users of extension accesses, in addition to providing them the means for controlling and customizing the permissions granted to their installed extensions.

V. Proposed Permission Framework

We propose and implement a runtime permission framework that allows for fine grain chrome permission monitoring and access control enforcement. The framework monitors Chrome API calls made by third party extensions and collects the data processed by these calls. For example, when the API `chrome.windows.getAll` is called, an allocated monitor within our framework collects the information relevant to the returned browser windows, such as the set of all Tabs within each of the browser windows. Given the runtime nature

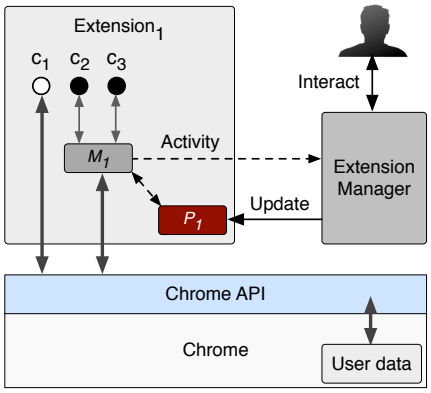


Fig. 3. Framework Architecture

of the framework, it can inform users in realtime of the specific accesses made by extensions (e.g. which specific URLs or cookies it has accessed), it can also enforce fine-grain access control onto attempted accesses. Additionally, the proposed framework allows for users to customize extension permissions, i.e. grant/deny permissions from the original set requested by an extension. The framework consists of two main components, the extension Manager, and extension Monitor. A single Monitor is allocated for each third party extension installed on a user's Chrome browser, and has its own associated access control Policy. All Monitors report back and are managed by the framework's extension Manager. Figure 3 illustrates the overall architecture of our framework.

A. Extension Manager

Our extension Manager is the main component within our framework that allows for monitoring third party extensions. The extension manager itself is a Chrome extension with NPAPI capabilities. NPAPI access allows us to adapt the behavior of third party extensions and allow the extension manager to listen to Chrome API calls made by these extensions, in addition to enforcing fine-grain access controls on requested accesses. In the following we discuss the tasks covered by the Manager.

Adapting Third Party Extensions To monitor API calls made by third party extensions, the manager modifies their default behavior by injecting a proposed Monitor component

that reports back to the manager. Figure 3 shows the Monitor M_1 that is assigned to $Extension_1$. This is achieved by including a custom built `monitor.js` script file into the extension's bundle, then linking to it from within the extension's HTML pages such as `background.html` and `popup.html`. When building the Monitor for a specific extension, the manager can selectively choose which API calls the Monitor should monitor. This allows for optimizing the monitoring process and avoiding unnecessary checks. For example, in Figure 3, only API calls c_2 and c_3 are monitored for $Extension_1$. We explain the details of our Monitor component in Section V-B.

API Notifications and Logging Once a Monitor is built and injected into an extension's bundle, the Manager starts listening to incoming message calls sent by the Monitor. These messages hold information on the Chrome API calls made by third party extensions. Using this information, the manager is able to keep users aware of the extension activities by notifying them in real time of the API calls made. The Manager also logs all accesses for future reference and are accessible via the Manager's UI.

Fine-Grain Permission Customization The Manager allows for users to customize the access control policy for each installed extension. Users are given fine-grain controls over the permissions granted to extensions and are provided with a simple user interface to do so as seen in Figure 4. There are mainly two types of permission controls provided:

1) **Permission-based:** These controls allow users to deny or allow a certain permission as a whole. Doing so prevents any API associated with the permission from executing. For example, users can choose to deny the permission `cookies` for an extension which will block all cookie associated Chrome APIs from executing.
2) **Host-based:** These controls allow/deny extensions from accessing certain hosts via the APIs of a certain permission. That is, we keep track of a *permission-to - host* dictionary that has all the hosts blocked for each permission of an extension. For example, a user could prevent an extension with `tabs` permission from accessing a Tab that is associated with a certain host such as `online.wellsfargo.com`. We provide host-based controls for the `tabs`, `cookies`,`history`, and `bookmarks` permissions. Host-based controls allow for decreasing the effect of excessive `host` permissions and the potential threats discussed in Section IV.

Users are also given the option to fully enable/disable certain extensions.

Extension Policy: Each third party extension is allocated a `policy.js` file which represents its access control Policy. The policy contains the fine-grain decisions made by users via the *Permission-based* and *Host-based* controls. That is, it contains a set of denied Chrome permissions in addition to a set of denied *permission-to-host* values. This Policy is used by an extension's Monitor to make the proper access control decisions whenever a certain API call is detected. Any customizations made by the user are immediately registered by

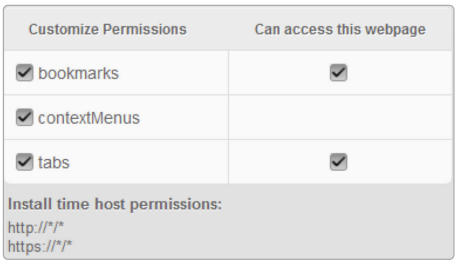


Fig. 4. Permission Customization

the Manager and written into the extension's Policy. Note that the Policy represents a negative access control list (ACL^-), hence if a Chrome permission or *permission-to-host* value does not exist within the Policy, it is considered allowed, otherwise it is denied. Also note that the `policy.js` is embedded within an extension's bundle. Figure 3 shows the Policy P_1 that is assigned to $Extension_1$.

Permission Details The Manager finally provides users with a

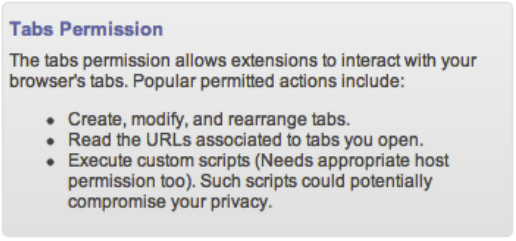


Fig. 5. `tabs` permission details

detailed description on each of the requested permissions. The detailed description for a specific permission also contains a set of examples on popular accesses that map to the Chrome APIs associated to a permission. We manually prepared the descriptions and examples. We evaluate the effectiveness of these detailed descriptions in our user study as explained in Section VII. Figure 5 shows the detailed description for the `tabs` permission.

B. Extension Monitor

An extension Monitor is a custom built JavaScript file (`monitor.js`) that we use to monitor the activities of third party extensions. When a Monitor is created for a specific extension, it is assigned a set of API methods to monitor. These APIs are assigned by our extension Manager to suit the permissions requested by extensions. For example, if an extension requests the `cookies` permission, the monitor would be asked to monitor the corresponding cookie API methods:`chrome.cookies.`[`getAllCookieStores, get, getAll, remove, set, onChanged`]. Note that the Manager could select a subset of these APIs, but we monitor all associated APIs by default.

The Monitor is also assigned a Policy (`policy.js`) which it uses in making access control decisions on the API calls it detects. When relevant API calls are detected by a Monitor, the following steps occur:

1) The Monitor intercepts the API call, i.e. the execution of the API runs through the Monitor. It then informs the Manager of this call.
2) An access control decision is made on the API call. This is decided based on two factors. First, the Chrome permission the API is associated to. If this permission is in the ACL^{-} of the Monitor's Policy, the decision is rendered as Deny. Second, the host used within the API (if applicable). If a *permission-to-host* value is found for the associated permission of the API, the decision is rendered as Deny. If either factors render a decision of Deny, then the final decision is Deny, otherwise it is Allow.
3) If the Policy decision retrieved is Allow, the Monitor executes the API call and returns the relevant results. Otherwise, if the decision is Deny, then the API is blocked and if appropriate returns an empty result (Some extension required an empty result to not break).

Note that in cases of APIs that do not specify a specific host value such as `chrome.windows.getAll`, the Monitor will filter the return values to not include any results associated with a *permission-to-host* value. For example, if the user has denied the host `online.wellsfargo.com` and the `chrome.windows.getAll` API results includes Tab objects associated with this host, then the results returned will exclude these Tabs.

VI. Evaluation

The framework was evaluated on a Windows 7 machine with a 2.4GHz i3 CPU, 4GB of RAM, and was running the Chrome browser version *16.0.912.75*. In our evaluation, we studied the 100 "top rated" Chrome extensions as listed on the official Chrome Web Store at the time of evaluation. The extensions covered all categories on the Chrome Web Store.

A. Implementation

To evaluate the proposed framework, we implemented the framework as a Chrome extension with NPAPI capabilities and used the FireBreath NPAPI Framework to develop the `dll` plugin used for the extension. Our Monitor component of the framework was implemented using the `FunMon2.js` function monitor [8], which allowed us to monitor API calls from within an extension's `monitor.js` file. We used Chrome's message passing APIs to establish the connections between the Manager and Monitor components, specifically the `onRequestExternal. addListener` and `sendRequest` APIs.

When users install our implemented extension, they are required to restart their browser to initiate the adaptation process on their installed extensions. At this point, the framework starts the monitoring process and access control enforcement. The main user interface was implemented via the extension's browser action and its `popup.html`. The browser action button shows the user the number of recent API notifications. The `popup.html` will display the recent notifications and the permission customization controls as seen in Figure 4. Users can also see a detailed activity log when clicking the Activity button of an extension, and can choose to enable/disable the extension. Finally, `popup.html` shows users the list of originally requested host permissions.

B. Permission Requests

When analyzing the 100 top extensions, we found the `tabs` permission to be most popular after the `host` permission. `tabs` was requested 77 times, followed by `contextMenus` 22 times, `cookies` 11, `notifications` 10, and the least requested was the `proxy` and `clipboardWrite` each requested once. From the 100 extensions, we analyzed permission requests that combine both the `tabs` and `host` permission. As discussed in Section IV, with both these permissions, extensions could represent a potential threat on user privacy. We found that 6.5% of extensions with the `tabs` permission have requested a `<all_urls>` `host` permission, 5% with `*://*/*`, 49% with `https://*/*`, and 60% with `http://*/*`. Whereas, 12% have either requested a specific host or ones with wild card subdomains. We also found that 11% have no `host` permissions. Note that the percentages do not add up to 100% because of extensions that use multiple host match patterns.

VII. User Study

To evaluate our proposed browser extension we conducted a user study that compares the Standard permission discovery method (By visiting an extension's detail page on the Chrome Web Store as in Figure 1) with our own browser extension REM. Participants in the study performed a number of tasks related to third party Chrome extensions and answered a number of questions on these tasks.

A. Methodology

The study participants were recruited from UNC-Charlotte. Each participant was supplied with a $10 Amazon gift card. We recruited a total of 20 participants to start the study, of which 18 successfully completed the study and 2 dropped out. Of the 18 participants, 11 were females and 7 were males. 88.2% of the participants are at least familiar with Chrome extensions and how to manage them within Chrome. Participants were given a brief introduction to REM's and to the existing Standard methods, and were also given a few minutes to familiarize themselves with both techniques. We then performed a within-subjects study comparison in which participants use either the Standard method or REM for performing the study tasks at first, then use the other method for performing the same tasks once again. Assigning a method (REM or Standard) to users was random, and the order of the methods assigned was counter balanced.

Study Tasks: Participants were given 8 different tasks & were asked to determine whether performing a certain action was permitted by a third party Chrome extension. For these tasks, participants could answer with: Yes, No, or Uncertain. Note that for each task a participant had to answer in regards to four different third party extension. The tasks were categorized into

Category	Task
Social Networking	Do the installed browser extensions have permission to read your private posts on social sites you visit?
Online Shopping	Do the installed browser extensions have permission to read your history of visited product pages?

Fig. 6. Example Tasks

Social Networking related tasks and Online Shopping related ones. For each category participants performed 4 different tasks. Examples of such tasks are illustrated in Figure 6.

Study Results: To evaluate the performance of participants on tasks, we considered two measures: 1)Response correctness, and 2)The time to finish a task measured in seconds. In Figure 8 we summarize the different time intervals for finishing correctly answered tasks. Notice that we consider only the correctly answered tasks as we are interested in the time it takes to correctly determine permitted actions among third party Chrome extensions. One can notice an overall higher

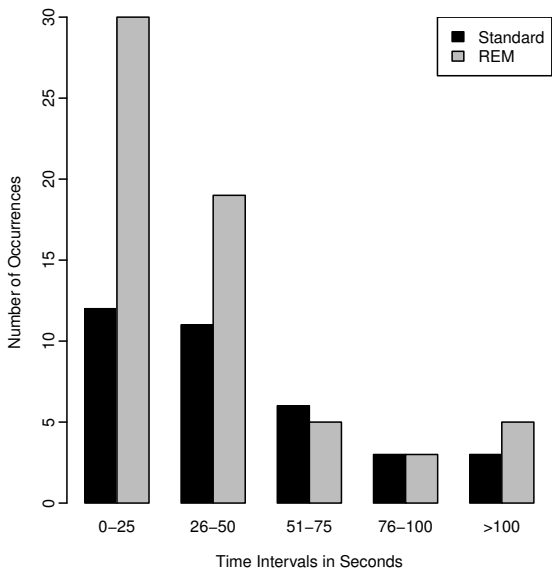


Fig. 8. Time distributions for correctly answered tasks

Task	Standard (μ, σ)	REM (μ, σ)	p-value
Social_1	(0.0, 0.0)	(0.294, 0.469)	**0.01003**
Social_2	(0.47, 0.51)	(0.64, 0.49)	0.13469
Social_3	(0.70, 0.469)	(0.70, 0.469)	0.5
Social_4	(0.11, 0.33)	(0.41, 0.50)	**0.02787**
Shopping_1	(0.0, 0.0)	(0.41, 0.50)	**0.002048**
Shopping_2	(0.235, 0.437)	(0.352, 0.492)	0.16609
Shopping_3	(0.176, 0.392)	(0.235, 0.437)	0.33417
Shopping_4	(0.235, 0.437)	(0.58, 0.50)	**0.014459**

Fig. 7. T-test Task Accuracy.

correctness when using REM, in addition to an overall lower time-to-task intervals. For example, participants were able to answer 30 tasks correctly within a time interval of 0-25 seconds using REM, whereas with the Standard method they were able to answer 12. Surprisingly, even when REM was the first tool option used by participants, it was still able to perform better than the Standard method.

To measure the significance of these results, we performed a t-test on the accuracy rate of participants. In Figure 7 we report the mean accuracy with standard deviation for all 8 tasks when using the Standard method vs. REM. Note that the accuracy rate was significant in tasks Social_1, Social_2, Social_4, Shopping_1, and Shopping_2 with a p-value $p < 0.05$.

In a post survey, participants were asked to assess our proposed browser extension REM and the Standard method using three Likert-scale questions. Participants responded to each of the following statements on a scale from one (strongly disagree) to seven (strongly agree).

S1: I am satisfied with the tool

S2: I was able to easily identify the permissions requested by each third party Chrome extension.

S3: I was confident in determining the permitted actions for installed third part Chrome extensions.

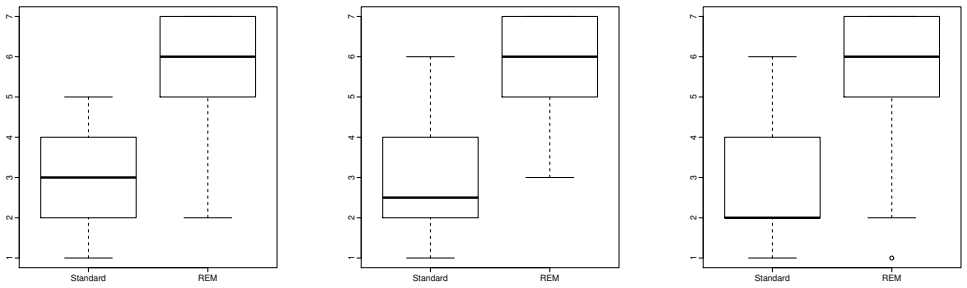
Figure 9 illustrates the user responses using boxplots. The black band in the middle of a box indicates the median. From the responses we observed that REM was rated significantly higher ($p < 0.05$) for all three statements.

VIII. Related Work

In the last few years several extension vulnerabilities have been discovered, which include stealing cookies, key logging, expose confidential information, and hijack the local operating system [3], [4], [5], [6]. In a white paper, Freeman et al. [9] investigated the possible security attacks on Firefox extensions.

Bandhakavi et al. [10], proposed applying static information-flow analysis to the JavaScript code used in the third party applications. They described a set of unsafe flow patterns that may lead to security vulnerabilities. This approach provides a mechanism to query the extension code for the defined unsafe flows and does not provide a mechanism to enable the user to monitor application behavior and control its access. Similarly static analysis [11] has been proposed to address security of web applications such as identifying SQL injection [12], and cross-site scripting [13], [14]. These techniques are complementary to ours, since our runtime monitoring and access control model could benefit from the discovered unsafe flows to recommend to the user fine-grain permissions to eliminate these flows.

Dynamic analysis techniques have also been used to trace information flow properties of JavaScript as it is being executed by the browser [15], [4]. Dhawan et al. [16] proposed a memory tainting approach to trace propagation of tainted objects during JavaScript execution and to raise alerts if an object containing sensitive information is accessed in an unsafe way. These approaches are effective in tracing dynamic program flow, however usually require users to install a modified or recompiled browser or JavaScript engine. Our proposed



(a) Q1: Overall Satisfaction (b) Q2: Identifying Permissions (c) Q3: Confidence

Fig. 9. Summary of Likert-Scale user responses

approach is easily integrated in current legacy browsers by simply installing a browser extension. In addition, our approach can easily be integrated with dynamic analysis tools.

IX. Conclusion and Future Work

This paper proposes a runtime framework that monitors 3rd party extension API calls, and is able to enforce a user's custom access control policy at runtime. Users are provided with fine-grained permission controls that allow for denying/allowing specific permissions. They were also able to set `Host-based` permissions that prevent APIs of certain permissions from accessing certain hosts. The framework was implemented as a Chrome extension "REM". Our future work includes the ability to monitor a larger set of Chrome APIs, and the ability to monitor network connections made by extensions.

References

[1] Mozilla Add-Ons Blog, "How many Firefox users have add-ons installed? 85%!" http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/.

[2] The Chromium Blog, "A Year of Extensions," http://blog.chromium.org/2010/12/year-of-extensions.html.

[3] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proceedings of the 2nd USENIX conference on Web application development*, ser. WebApps'11, Berkeley, CA, USA.

[4] Y. Zhou and D. Evans, "Protecting private web content from embedded scripts," in *Proceedings of the 16th European conference on Research in computer security*, ser. ESORICS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 60–79. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041225.2041231

[5] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," *17th Network and Distributed System Security Symposium*, 2010.

[6] M. Ter Louw, J. Lim, and V. Venkatakrishnan, "Enhancing web browser security against malware extensions," *Journal in Computer Virology*, vol. 4, pp. 179–195, 2008.

[7] OAuth, "Security Advisory:2009.1," http://oauth.net/advisories/2009-1/.

[8] Stephen W. Cote, "FunMon2.js," http://www.imnmotion.com/documents/html/technical/dhtml/funmon.html.

[9] R. S. Liverani and N. Freeman, "Abusing Firefox Extensions," in *Defcon*, July 2009.

[10] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "Vex: vetting browser extensions for security vulnerabilities," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929850

[11] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*. Berkeley, CA, USA: USENIX Association, 2005, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251416

[12] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267349

[13] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in web applications," in *Proceedings of the Web Site Evolution, Sixth IEEE International Workshop*. Washington, DC: IEEE Computer Society, 2004, pp. 71–80. [Online]. Available: http://dl.acm.org/citation.cfm?id=1025133.1026460

[14] V. N. Venkatakrishnan, P. Bisht, M. T. Louw, M. Zhou, K. Gondi, and K. T. Ganesh, "Webapparmor: a framework for robust prevention of attacks on web applications," in *Proceedings of the 6th international conference on Information systems security*, ser. ICISS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 3–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=1940366.1940369

[15] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "Javascript instrumentation in practice," in *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, ser. APLAS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 326–341. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89330-1_23

[16] M. Dhawan and V. Ganapathy, "Analyzing information flow in javascript-based browser extensions," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 382–391. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2009.43