Discovering Browser Extensions via Web Accessible Resources

Alexander Sjösten
sjosten@chalmers.se

Steven Van Acker
Chalmers

Andrei Sabelfeld
Chalmers

ABSTRACT

Browser extensions provide a powerful platform to enrich browsing experience. At the same time, they raise important security questions. From the point of view of a website, some browser extensions are invasive, removing intended features and adding unintended ones, e.g. extensions that hijack Facebook likes. Conversely, from the point of view of extensions, some websites are invasive, e.g. websites that bypass ad blockers. Motivated by security goals at clash, this paper explores browser extension discovery, through a non-behavioral technique, based on detecting extensions' web accessible resources. We report on an empirical study with free Chrome and Firefox extensions, being able to detect over 50% of the top 1,000 free Chrome extensions, including popular security- and privacy-critical extensions such as AdBlock, LastPass, Avast Online Security, and Ghostery. We also conduct an empirical study of non-behavioral extension detection on the Alexa top 100,000 websites. We present the dual measures of making extension detection easier in the interest of websites and making extension detection more difficult in the interest of extensions. Finally, we discuss a browser architecture that allows a user to take control in arbitrating the conflicting security goals.

Keywords

Web security; Browser extensions; Large-scale study

1. INTRODUCTION

Browser extensions provide a powerful platform to enrich browsing experience. The Chrome web store currently contains around 43,000 free extensions, with many of these extensions, such as AdBlock, Adobe Acrobat, and Skype, having more than 10,000,000 users.

From the security point of view, browser extensions are deployed as a "man in the browser" [27], implying that extensions have privileges to arbitrarily alter the behavior of webpages. Naturally, the power of browser extensions creates tension between the security goals of the webpages and those of the extensions themselves. Let us consider some representative scenarios to illustrate the challenges in balancing these goals.

The first and second scenarios present an exclusive point of view of websites, concerned with malicious extensions. The third scenario presents an exclusive view of extensions, concerned with malicious websites. The fourth scenario illustrates legitimate synergies between websites and extensions. Finally, the fifth scenario illustrates the security goals of websites and extensions at outright clash.

Bank scenario Bank webpages manipulate sensitive information whose unauthorized access may lead to financial losses. It is desirable to detect potentially insecure and vulnerable extensions and prevent extensions from injecting third-party scripts into the bank's webpages. The latter technique is in fact a common practice for many extensions [28, 31]. This scenario motivates the goal of discovering browser extensions, as the knowledge of what extensions run on the webpage can be used for tuning the defense.

Facebook scenario With over a billion daily users [15], Facebook is a popular target for attacks. Since the Facebook application itself is relatively well protected from attacks like cross-site scripting, attackers look for attacks elsewhere. A prevalent threat to user integrity and confidentiality is the use of browser extensions to inject scripts into the Facebook application to gain full access to the user's account [12]. Jagpal et al. [31] identify Facebook as the number one target for malicious extensions, reporting on the proliferation of attacks such as fake content (ad or otherwise) injection and information stealing.

This scenario motivates the need for recognizing browser extensions by webpages. Having an extension detection technique available, the webpage can adapt its behavior to the extensions installed. Research by Facebook's anti-abuse team confirms that this is a realistic scenario [12].

LastPass scenario LastPass [34] is a password manager that permits users to only remember one master password while automatically generating, storing, and filling in passwords for the individual services. The LastPass Chrome extension has currently over 4,000,000 users. Being a sensitive extension, LastPass has been subject to attacks. For example, LostPass [35] is a "pixel-perfect phishing" attack that exploits the fact that LastPass displays its notification in the browser viewport. LostPass fakes a message of an expired session and redirects users to a fake login page where it harvests the master password. (LastPass subsequently responded by interface measures and asking for email confirmation for all logins from new IPs [33].)

This scenario motivates the need to protect sensitive extensions. Being able to detect LastPass is a trigger for phishing attacks via a malicious webpage, as in the case of LostPass. It is in the interest of LastPass to stay undetected. Similar scenarios arise with extensions such as Avast Online Security and Ghostery, popular security- and privacy-critical extensions that can be targeted by malicious websites.

Google Cast scenario Google Cast [26] is a popular extension to play content on a Chromecast device from Chrome. Upon detecting the Google Cast extension, websites like Twitch.tv adjust their functionality and offer richer features.

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA
© 2017 ACM. ISBN 978-1-4503-4523-1/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3029806.3029820

This scenario highlights the benefit of browser extension detection, as motivated by enriching functionality rather than by security considerations.

AdBlock scenario With over 40,000,000 users, AdBlock is currently the most popular Chrome extension [10]. It is in the very nature of ad blocking to modify webpages, looking for ads and blocking them. These goals are clearly at odds with the webpages' goals. Consequently, some webpages try to detect ad blockers.

This scenario motivates both the need for extension detection from the point of view of webpages and the need for evading discovery from the ad blockers' point of view. As we detail in Section 2, the state of the art for this scenario is much of a cat-and-mouse game.

Security goals at clash The above scenarios demonstrate that the different stakeholders (websites vs. browser extensions) have different interests, resulting in the clash of the respective security goals. Motivated by these security goals, this paper focuses on discovering browser extensions and pursues the following research questions: (i) How to discover browser extensions from within a webpage, i.e, without modifying the browser? and (ii) How can extensions evade detection?

We emphasize that this paper does not assume the interest of webpages over the interest of extensions or vice versa. Instead, we recognize that these different interests are legitimate, even if conflicting. We seek to better understand these interests, conceptually and empirically, and suggest steps to improve the state of the art on both sides.

Non-behavioral extension discovery We refer as *behavioral* to extension discovery techniques that require analyzing the behavior of browser extensions. Behavioral detection is sometimes desirable, when a particular behavior needs to be detected, regardless of what extension triggers it. On the other hand, *non-behavioral* discovery detects extensions without having to analyze their behavior. Non-behavioral detection is attractive when it can be done with low efforts. This motivates our focus on non-behavioral techniques.

In similar vein, when we consider measures against extension discovery, our goal is to stop non-behavioral detection and force attackers to do behavioral analysis of extensions.

Discovery via web accessible resources We explore a non-behavioral technique for discovering extensions, based on so called *web accessible resources* and implement it for detecting Chrome and Firefox extensions. Web accessible resources are the resources accessible in the context of a webpage. These resources enable interaction of extensions with the user via the underlying webpages.

While there are other, more elaborate, ways to set up this kind of interaction without web accessible resources (see Sections 3.2 and 6.2), web accessible resources provide a straightforward mechanism of direct access via URIs. Indeed, as we will see later, web accessible resources are used by many popular extensions.

Our detection is precise, in the sense of no false positives, and robust, as long as extensions require web accessible resources. While behavioral techniques may mistakenly detect an extension based on a monitored behavior, our technique is based on detecting resources that are bound to unique extension ids, implying that we never report an extension that is not present.

Contributions To the best of our knowledge, this work is the first comprehensive effort on non-behavioral extension detection, putting the spotlight on a largely unexplored area and systematically studying the technique and its applicability at large scale. To this end, the paper offers the following contributions:

Precise non-behavioral extension discovery We investigate a non-behavioral extension detection technique, based on web accessible resources (Section 3). Based on unique extension ids, our detection is precise, in the sense of no false positives, and robust, as long as extensions require web accessible resources.

Empirical studies of Chrome and Firefox extensions We report on a empirical study with Chrome's free extensions where we detect over 50% of the top 1,000 free Chrome extensions, including popular security- and privacy-critical extensions such as AdBlock, LastPass, Avast Online Security, and Ghostery, and 28% of the Chrome extensions in the study overall (Section 4).

We report on a similar study with Firefox's free extensions (Section 4). Due to Firefox's lax architecture, extensions are not prevented from direct modifications to the UI of the browser. This explains the lesser need for web accessible resources in Firefox extensions and, therefore, lower discovery rates.

Demo webpage for Chrome and Firefox We provide a demo webpage [55] to demonstrate discovery of Chrome and Firefox extensions in practice. This proof-of-concept webpage lists detected extensions once a user visits the page with Chrome or Firefox. This page serves as a starting point, providing a core that can be further developed either as a standalone service or a library for inclusion into other webpages. In fact, our code is already used by INRIA's Browser Extension Experiment [30].

Empirical studies of the Alexa top 100,000 websites We conduct an empirical study of non-behavioral extension discovery on the Alexa top 100,000 websites. Our findings suggest that the technique is not widely known, although we do discover several websites that try to find extensions for types that include fun, productivity, news, weather, search tools, developer tools, accessibility, and shopping (Section 5).

Measures We discuss two types of measures that correspond to the interests of webpages and extensions, respectively. For webpages, we discuss a solution based on extension whitelisting. For extensions, we have recommendations to restrict APIs related to web accessible resources and webpage whitelisting (Section 6). We also discuss behavioral techniques and argue that to be effective, they need to be extension-specific.

2. STATE-OF-THE-ART ARMS RACE

The state of the art is best illustrated with the arms race between ad blockers and ad blocker detectors, with its rival spirit captured by the (blatantly explicit) naming of the respective libraries.

Whenever an extension manipulates the webpage's DOM, it can be discovered using behavioral analysis. For instance, a webpage can discover an ad blocker when the latter removes an ad from the webpage. Since ad blockers act as good examples of security goals at clash, the rest of this section will focus on the arms race between webpages and ad blockers. Table 1 summarizes the steps in this arms race.

A straightforward approach to check for ad blockers is to create a fake ad which sets a global variable and then check

```
<script src="showads.js">
<script>
  if(window.canRunAds === undefined)
  {
    // Ad blocking detected
  }
</script>
```

(a) HTML part of fake ad

```
var canRunAds = true;
```

(b) showads.js (fake ad)

Figure 1: Ad-blocking behavioral detection

AdBlock	Remove ads
FAB	Injects bait for AdBlock and analyzes behavior
FFAB	Exploits global property in window object set by FAB
FFFAB	Detects if FFAB has done anything, reverts the changes

Table 1: Ad blocking arms race

for that specific variable. Figure 1 displays a current solution [29] which works in AdBlock, AdBlock Plus and AdBlock Pro for Chrome, as well as AdBlock Plus for Firefox, where the default behavior is to block the execution of the file `showads.js`.

Such a useful behavioral technique is often prepackaged as a JavaScript library marketed for detecting ad blockers, called "anti ad blockers". One such example is F***AdBlock (*FAB*) [11], which helps the users do behavioral analysis during a user-specified time interval. If a certain (user defined) amount of negative results in a row occurs, no ad-blocking tools are deemed to be running. This means the check can be run multiple times, making it harder for ad blockers to hide their presence by delaying their interaction.

Just as there are tools designed to help detect ad blockers, there are also tools that detect anti ad blockers. The library F***F***AdBlock (*FFAB*) [37] is an anti anti ad blocker created as a response to the anti ad blocker FAB. FFAB redefines some JavaScript function objects used during FAB's execution, overriding FAB's ad blocker detection mechanism and claims no ad blockers are detected.

But just as FAB is sensitive to behavioral analysis, so is FFAB. In turn, F***F***F***AdBlock (*FFFAB*) [13], is a response to FFAB. FFAB itself is not careful enough when overriding FAB's code, which gives FFFAB an opportunity to detect when FAB's code has been tampered with. When FFFAB detects this manipulation, it restores the original FAB functionality.

Detection of extensions by webpages is possible if the extension somehow modifies the DOM. In addition, behavioral detection is usually cross-browser, as the same behavior will take place no matter which browser is used.

If webpages are forced into behavioral extension detection, they cannot easily determine which extension is causing the behavior, and the extension detection loses precision. If they instead find extensions using unique ids, the extension name for Firefox extensions or a 32-character textual token for Chrome extensions, the extension can be uniquely determined and the detection is exact.

As this arms race indicates, behavioral extension detection is both error-prone because it is imprecise, and costly because it requires time and effort to keep up with the latest evasion techniques. These reasons motivate the need for a more robust and cheaper technique, bringing us to the study of non-behavioral extension detection in the following sections.

3. FINDING EXTENSIONS VIA WEB ACCESSIBLE RESOURCES

This section provides background on how browser extensions work in Chrome and Firefox, the role of web accessible resources, how they can be used for finding extensions and the attacker models considered in this work.

3.1 Extensions

An *extension* is a program, typically written in a combination of JavaScript, HTML and CSS to extend the browser functionality. Extensions are not to be confused with browser *plugins*, such as Flash and Java, that are compiled and loadable modules that may live outside the browsers' process space. Extensions may alter the content of a webpage (e.g. ad blockers) or add features such as executing personal scripts (e.g. Greasemonkey). Browser extensions are built using architectures defined by the browser vendors. Mozilla is currently working on *WebExtensions* [48], a new API which will have a similar structure as the Chrome extension API.

Chrome extensions Chrome extensions can consist of three different parts [25]: (i) a background page, which is an invisible page containing the main logic of the extension; (ii) UI pages, ordinary HTML pages that display the extension's UI ("browser actions" [19] and "page actions" [20]); and (iii) a content script, JavaScript which executes in the context of the webpage. The content script makes the interaction with the webpage and runs in an isolated world [21]. It has access to some Chrome APIs and can communicate with the background page using message passing [24].

Each Chrome extension must have a manifest file, `manifest.json`, which contains important information about the extension [23]. For this work, the only interesting section in the manifest file is *web_accessible_resources*, which defines which resources are accessible in the context of a webpage [22]. The content of the *web_accessible_resources* section is paths to files. They can be URLs or a path to files relative to the package root and can contain wildcards.

Firefox extensions Firefox extensions written using *WebExtensions* will have the same structure as Chrome extensions. This is because Chrome extensions should be easy to port to Firefox [46], as well as having a more unified cross-browser architecture.

For the rest of this section, we will focus on XUL/XPCOM extensions. As this is how most Firefox extensions currently are written, we will refer to them as "Firefox extensions". These extensions also uses manifest files. The extensions automatically read the file `chrome.manifest` in the extension's root [40, 43]. Differently from Chrome, manifest files in Firefox are not mandatory and one manifest file can refer to other manifest files in sub folders.

Similarly to Chrome, a content script can inject and alter content on the webpage and communicate with the background pages using message passing [42, 41]. In the file `chrome.manifest`, a flag `contentaccessible`, which when set to `yes`, makes the specified content web accessible [40].

Differently from Chrome and WebExtensions, Firefox extensions have powerful features such as `overlay`, to describe extra content to the UI [49] and `override`, to override a chrome file provided by the application [40].

3.2 Web accessible resources

Both Chrome and Firefox require that extension resources that are referenced in a regular webpage, are flagged as web accessible in the manifest files. In Chrome and WebExtensions this is done with the key *”web_accessible_resources”* [22, 47] and in Firefox extensions with *”contentaccessible=yes”* [40].

If a Chrome content script injects resources into a webpage, the resource must be flagged as web accessible. This makes the resource available using the following schema: `chrome-extension://<extensionid>/<pathToFile>`, where `<extensionid>` is a unique identifier for each extension and `<pathToFile>` is the same as the relative URL from the package root [25].

Similarly for Firefox, if resources from the extension are to be referenced by an untrusted part using `<img>` or `<script>` tags, the corresponding registered content package must be flagged with `contentaccessible=yes`. Doing this would allow for the webpage to load resources from the extension, e.g. images to an `<img>` tag [40]. The content can then be accessed using the `chrome://packagename/content/` schema [40], where the `packagename` should be unique for all extensions. For WebExtensions, the content can be accessed with `moz-extension://<extensionid>/<pathToFile>` [47].

Examples of web accessible resources in practice To illustrate web accessible resources and how they differ in Firefox and Chrome, consider two real-world examples: AdBlock and LastPass.

AdBlock for Chrome displays an icon in the browser toolbar which seemingly triggers a popup. This popup is actually an HTML page which loads JavaScript code to interact with the user. Both the HTML and JavaScript files are web accessible resources and must be listed as such [22].

When logging in to a new website with a password, LastPass for Chrome will prompt the user whether this password should be stored. This prompt is actually an “overlay” injected and rendered into the viewport of the visited webpage. The overlay is an HTML resource provided by the extension and marked as web accessible. LastPass for Firefox uses a slightly different approach because Firefox extensions have the ability to modify the browser chrome through *XML User Interface Language (XUL)*. Because this XUL file is only part of the browser chrome it does not need to be accessible from the visited webpage. Therefore, it does not need to be marked as a web accessible resource.

Benefits with web accessible resources While web accessible resources are a convenience, it is possible to do without them. Resources can be represented as strings using data URIs [36], which can be added to the created DOM element before injecting it to the webpage. It is also possible to store the resources on an external server and fetch them from there. However, both of these approaches have disadvantages. Encoding and injecting resources as strings can be difficult to maintain, and storing resources on an external server has potential privacy and security issues.

By using web accessible resources, the resources are stored within the extension. This make them easier to maintain and access with extension APIs.

Finding extensions via web accessible resources Because web accessible resources can be accessed in the context of a given webpage, they can be abused to detect the presence of browser extensions to which the resources belong. As mentioned above, LastPass for Chrome has the overlay file `overlay.html` marked as web accessible, making it possible to make a request for the file using e.g. XMLHttpRequest. If the resource is present, the request will receive a positive answer, indicating that the extension is installed.

In Firefox, the extension Firebug has `contentaccessible=yes` set. Similarly to LastPass in Chrome, this makes Firebug detectable without behavior analysis, as the resource can be loaded to a `script` tag, using `onsuccess` and `onerror` to check if the extension is present or not.

Note that thanks to the uniqueness of the extension ids, we obtain a detection technique without false positives. While there is no guarantee that the behavioral techniques precisely detect a given extension, we never report an extension that is not present. Compared to behavioral techniques that may have both false positives and negatives, finding extensions via web accessible resources may have false negatives but no false positives.

3.3 Two attacker models

Recall that we are interested in two perspectives on extension detection: that of a webpage with the goal to enable extension detection (as in the Bank and Facebook scenarios) and that of an extension with the goal to remain hidden (as in the LastPass scenario). Consequently, this yields two attacker models. The first attacker model corresponds to a malicious extension that has been installed on a user’s browser, e.g., to leak bank data or hijack likes. The challenge is to detect such extensions. The second attacker model corresponds to a malicious webpage that tries to thwart the functionality of a legitimate extension, e.g., by blocking ads or phishing. The challenge here is to prevent detection of such extensions. In this paper, we address both perspectives, even if their goals are by nature conflicting.

4. EMPIRICAL STUDY OF CHROME AND FIREFOX EXTENSIONS

This section reports on an empirical study to analyze how susceptible free extensions are to be found via web accessible resources.

The study was performed by downloading all free extensions from Chrome web store [18] and Mozilla’s add-on store [44], extracting and analyzing their manifest files. The extensions were downloaded in September 2016.

4.1 Chrome

As mentioned in Section 3.1, *web_accessible_resources* in the manifest file can be used to determine extension detection via web accessible resources. If the manifest file does not contain the section *web_accessible_resources*, the extension cannot be detected using this technique. If the only accessible resources of an extension are URLs, we deem the extension non-detectable without behavioral analysis.

A total of 43,429 extensions were downloaded. However, the total amount of extensions where the user statistics were found by the scraper was 43,197 (≈99.5% of all downloaded extensions). The reason for this drop is that some extensions were removed from the Chrome web store before the scraper had the time to retrieve the user statistics, whereas some extensions (like Google Cast) did not display user statistics.

Results Table 2 displays the results of testing all downloaded Chrome extensions for *web_accessible_resources.* The parsing of the manifest files yielded parse errors for 36 ex-

Category	Chrome	Firefox
Empty accessible resources	148	–
Only URLs	54	–
No manifest file	–	7,396
Detectable	12,154	1,003
No accessible resources	31,073	6,497
Total amount of extensions	43,429	14,896

Table 2: Chrome and Firefox extension results

tensions, for which we manually edited the manifest files to remove the errors.

We note that 148 extensions have *web_accessible_resources* set to an empty array in the manifest file, which implies that these extensions have no web accessible resources. Similarly, the 54 extensions which only have URLs as web accessible resources cannot be found with our technique as they do not have resources that should run in the context of the website stored locally in the extension. The "No accessible resources" in Table 2 are all the extensions where the *web_accessible_resources* field was missing in the manifest file, including 146 extensions which had only non-existing resources listed.

In total, 12,154 extensions out of 43,429 could be found using non-behavioral extension detection, which corresponds to ≈28%. Figure 2 shows the amount of detectable extensions sorted by popularity, based on the reported number of users in the Google Chrome web store. For this, we only use the set of extensions for which we could find user statistics, yielding 12,112 extensions detectable out of 43,197. We divide the sorted extensions in groups of 1000, which we call "intervals". We find 70% of the top 10, 62% of the top 100 and 52.7% of the top 1000 extensions with a non-behavioral technique. These extensions include popular security- and privacy-critical extensions such as AdBlock, LastPass, Avast Online Security, Ghostery and Disconnect. The graph also shows a descending trend, indicating that more popular extensions have on average more *web_accessible_resources*.

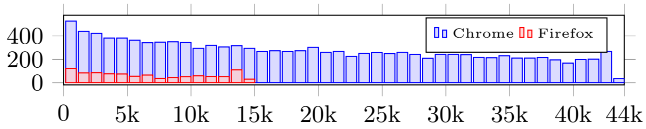

Figure 2: Amount of discoverable browser extensions (y-axis, not stacked) based on extensions' popularity rank (x-axis)

4.2 Firefox

As mentioned in Section 3.1, manifest files for Firefox extensions can be located in several different sub folders of an extension. The manifest files in the sub folders are referenced from `chrome.manifest` in the root directory. For this study, all manifest files were analyzed, including the manifest files in the sub folders.

The `contentaccessible` flag indicates web accessible resources, but we found that a webpage cannot perform a normal `XMLHttpRequest` in order to retrieve the resource. However, it is possible to create a `script` tag with the corresponding `script.src` attribute set to the resource in order to retrieve it. By attaching `onload` and `onerror` event handlers to this `script` element, it is possible to learn whether the resource could be retrieved. In addition, because the absence of a resource is gracefully handled with the `onerror` handler, no error is reported and this method in Firefox is more discrete than the method used with Chrome.

The amount of Firefox extensions was 17,375. However, some extensions were duplicated in the list on Mozilla's add-on page based on the extension name and the extension id. The scraper found a total of 14,925 unique extensions, but was redirected to a dead link for 29 extensions, yielding the total number of analyzed extensions to 14,896.

Results The results of the study can be seen in Table 2. 7,396 did not have a `chrome.manifest` file in the extension's root directory and 6,381 extensions did not have the flag `contentaccessible` in the `chrome.manifest` file in the root directory. 116 out of the 1,119 extensions who had set `contentaccessible` linked it to non-existing files. We also detected a total of 775 extensions who use `WebExtensions`. Out of those 775 extensions, 11 also defined `chrome.manifest`. 221 had `web_accessible_resources` set, indicating ≈ 28,5% of those extensions should be detectable. Unfortunately, *WebExtensions* extension ids are not stored publicly. One could, in theory, manually install all those extensions and see if they have e.g. an options page [45], which when browsed to would give the extension id. Due to this, we do not consider WebExtensions detectable in this experiment.

1,003 out of 14,896 can be found with web accessible resources, which corresponds to 6.73%. The trend for the detectable extensions can be seen in Figure 2. The interval with the most extensions that are detectable was the top 1000 extensions with 121 detectable extensions (i.e. 12.1%). These extensions include Firebug, Easy Screenshot and Web of Trust. However, no ad blockers nor the popular script blocker Ghostery can be found in Firefox without behavioral analysis. As explained in Section 3.2, Firefox extensions have the ability to directly add to the UI using XUL, so that they do not require web accessible resources like Chrome extensions. Therefore, Firefox extensions need less web accessible resources.

4.3 Comparison of results

One major difference between Chrome and Firefox is how `XMLHttpRequest` is handled. In Firefox, it is not allowed to access `chrome://` with `XMLHttpRequest`, whereas it is possible to access `moz-extension://` in Firefox and `chrome-extension://` in Chrome. The use of web accessible resources, and with that the percentage of detectable extensions, is higher for Chrome. As a Chrome extension cannot make much modifications to the UI of the browser compared to Firefox, there is a greater need for using web accessible resources in Chrome. Similarities could be found in the trends of accessible resources, where both browsers had the largest interval of detectable extensions in the top 1000 extensions, but Chrome had a more clear decrease over the following intervals compared to Firefox.

5. BROWSER EXTENSION DETECTION IN THE ALEXA TOP 100,000

We conducted an empirical study of non-behavioral extension detection on the Alexa top 100,000 websites. Our findings suggest that the technique is not widely known, although we do discover several websites that try to find extensions for types that include fun, productivity, news, weather, search tools, developer tools, accessibility and shopping.

This empirical study has been omitted from this version of our work due to space limitations, but is available in the full version [55].

6. MEASURES

Section 6.1 suggests measures in favor of website developers, while Section 6.2 suggests how extensions can prevent being found by webpages. Finally, Section 6.3 concludes with a discussion of how to resolve security goal clashes. The full version [55] elaborates on further details.

6.1 Measures for webpages: whitelisting extensions

To help webpages guarantee a clean web environment for their content, they can be allowed to specify a whitelist of allowed extensions. Such a measure can be implemented as a policy specified by the webpage and enforced by the browser.

For a web application handling sensitive information, an environment known-to-be free from malware would help secure the user's sensitive data. Such a whitelist can, of course, also be used to block any extensions, e.g. an ad blocker, as well.

We envision the webpage suggests the whitelist. One possibility in this design space is to leave the final decision up to the user, endorsing and/or overriding the whitelist, if desirable. We detail this possibility in Section 6.3.

6.2 Measures for extensions

We discuss some measures to reduce the risk for extensions to be detected using non-behavioral analysis.

Prevent direct access to extension resources from webpage Instead of a direct access from webpages to an extension's resources, webpages would need to go through the extension via a message passing API. This would not prevent detection entirely, but it would allow for an extension to be part of the detection process.

No accessible resources One can avoid web accessible resources by hosting the resources on an external server or use data URIs (see Section 3.2).

Using an external server, with or without the browser's caching mechanism, does not fully prevent detectability via a timing attack. Remotely hosted resources also introduce privacy concerns, as all requests can be monitored by an external party.

Data URIs will remove dependencies on web accessible resources, but a disadvantage is that hard-coded data URIs can be difficult to maintain.

Track script provenance One could potentially track who injected a script and only allow access to a given set of principals. Tracking information flow can, however, make the system slower, but it would allow for web accessible resources to be used by scripts on the webpage that originates from the extension, but not by the actual webpage.

Extension ids In order to avoid detection, an extension developer could change the extension id by resubmitting the same extension. This by itself would be of limited effect since the entire userbase needs to be rebuilt for the extension with the updated id.

As an extension has other means to fetch its resources than via web accessible resources, one can allow the extension to generate a random token and pass it along to the webpage. A webpage which possesses this token can use it to gain access to the extension's resources.

Whitelisting webpages Instead of being active on all webpages a browser visits, extensions could be activated on a case-by-case basis. If an extension is not active on a webpage, and its resources not available to this webpage, then it can not be detected through the presence of web accessible resources. This can be implemented through a user-modifiable whitelist in the browser.

6.3 User to resolve conflicting security goals

Because of the conflicting security goals, it is important to strike a balance of the interests of the different parties by combining webpage measures with extension measures. For example, allowing webpages to whitelist extensions which can be active in their domain, whereas allowing extensions to whitelist webpages which are allowed to communicate with the extensions would help both webpages and extensions reach their goals.

But who should resolve the conflicting security goals? We resort to the ”users > developers > browser” principle, as common in the web community folklore. This principle gives users precedence over developers and browsers in the web setting. Driven by this principle, we designate the user as an arbiter to endorse and/or overwrite whitelists provided by webpages and extensions, respectively.

7. RELATED WORK

Non-behavioral extension detection has so far received only scarce attention, primarily in the form of scattered blog posts [6, 3, 2, 4, 1, 5], some referring to outdated browser features and some only traceable in Internet archives [6, 3].

To the best of our knowledge, we are the first to systematically study non-behavioral extension discovery at large in both Chrome and Firefox's extension web stores, as well as the Alexa top 100,000 webpages.

There is a large body of work on detection of maliciously behaving browser extensions. The state of the art is well summarized by Jagpal et al. [31]. The rest of this section focuses on detecting extensions and fingerprinting browsers.

7.1 Detecting extensions

Prior work in detecting extensions has focused on behavioral techniques. For instance, Nikiforakis et al. [52] analyze eleven popular browser extensions that hide the real user agent string from visited websites in order to obfuscate a browser's fingerprint, but observe that the these extensions neglect to remove the same information from the JavaScript environment, making the extension detectable by a visited website through its behavior. This detection mechanism is fragile since, as explained in Section 2, extensions may modify their behavior in order to avoid detection, forcing websites to alter their detection method, triggering an arms race. Using another approach, Thomas et al. [56] detect the in-flight alteration of a webpage, by comparing the DOM of the rendered webpage against the expected DOM. This catch-all method detects all DOM modifying extensions as well as proxies and compromised browsers. Such an approach is more robust, since it will detect all extensions that modify the DOM even when they attempt to evade detection. However, since it does not focus on an extension's specific behavior, it is less precise. Non-behavioral extension detection on the other hand, like the technique presented in this paper, uses simple and cheap checks to determine the presence of a specific extension, without false positives. In addition, an extension can not evade detection by altering its behavior. Instead, the only way for an extension to avoid detection is by removing its web accessible resources, which is not always practical as explained in Section 6.2.

Non-behavioral extension discovery via web accessible resources has only received scarce attention in the form of scattered observations, primarily in blog posts [6, 3, 2, 4, 1, 5], some referring to outdated browser features and some only traceable in Internet archives [6, 3].

We go beyond these observations by systematically studying the entire class of extension discovery via web accessible resources, performing an empirical study with discoverability of all free extensions of the two major browsers, preforming a large scale study of discovery by the top 100,000 Alexa webpages, and proposing measures.

7.2 Fingerprinting browsers

There has been much work on browser fingerprinting. INRIA's Browser Extension Experiment [30] is based on our technique and code to enhance browser fingerprinting by detecting extensions. We overview the work on fingerprinting below, noting that the rest of the approaches are less related because they do not address extension detection.

Panopticlick [54] uses such browsers properties as screen resolution, user agent string, timezone, system fonts, and browser plugins to uniquely identify browsers. Browsers can also be fingerprinted through browser quirks [7], canvas fingerprinting [39, 8], dimensions of rendered font glyphs [16], browser histories [53], ECMAScript compliance [50], performance of the JavaScript engine and whitelisted domains in the NoScript extension [38], and more [52, 58].

Nikiforakis et al. [52] detect font probing and flash-based proxy evasion as fingerprinting mechanisms provided by three commercial fingerprinting companies, and find 40 websites in the Alexa top 10,000 make use of them. Acar et al. build FPDetective [9] and find 404 websites in the Alexa top million that use JavaScript-based font probing, as well as 145 websites in the Alexa top 10,000 that use Flash-based font probing to fingerprint visitors. Acar et al. [8] study the Alexa top 100,000 and find that canvas fingerprinting is the most commonly used fingerprinting technique, with 5% of the studied websites using it.

Defending against fingerprinting is difficult, if even possible. There appears to be no one-size-fits-all solution. Several strategies have been suggested. One crude way to address the problem is by simply blocking certain forms of third-party content, such as JavaScript or Flash known to contain fingerprinting code [8, 14, 52, 53, 58]. Similarly crude would be to disable certain functionality in the browser, such as the ability to query pixel-values from a canvas [39].

Instead of blocking third-party content or functionality, a browser could ask for user permission whenever a fingerprintable characteristic of the browser is queried, e.g. reading those pixel-values from a canvas [8, 39, 58].

Yet another approach adds (smart) noise to fingerprintable browser characteristics, thereby randomizing the fingerprint [8, 39, 14, 16, 17, 32, 51, 57, 58]. The reverse approach is to decrease the randomness of the reported browser characteristics by standardizing the set of possible values for fingerprintable resources, such as the list of system fonts, so that all browsers report the same values [16, 39, 52, 58].

Conceding that fingerprinting cannot be stopped, recent work has investigated preventing the exfiltration of the fingerprint itself by monitoring network traffic [57, 16, 50], or even by rewriting a detected fingerprint through a network proxy [59].

8. CONCLUSION

To the best of our knowledge, we have presented the first comprehensive study of non-behavioral browser extension discovery. We have systematically studied the technique and its applicability at large scale. At the core of our technique is detection of web accessible resources that are associated with extensions via unique extension ids. This yields an effective detection technique with no false positives, which we have instantiated for both Chrome and Firefox. We report on an empirical study with free Chrome and Firefox extensions, detecting over 50% of the top 1,000 free Chrome extensions (including such sensitive extensions as AdBlock and LastPass) and over 28% of the Chrome extensions in the study overall. We have conducted an empirical study of non-behavioral extension detection on the Alexa top 100,000 websites. This study confirms that detecting extensions via web accessible resources is not widely known. Nevertheless, we identify websites that perform extension detection for types of extensions that include fun, productivity, news, weather, search tools, developer tools, accessibility, and shopping. We have presented measures for and against browser extension discovery, catering to the needs of website owners and extension developers, respectively. Finally, we have discussed a browser architecture that allows a user to take control in arbitrating the conflicting security goals.

Our code for discovering browser extensions is already used by INRIA's Browser Extension Experiment [30].

Future work focuses on the measures outlined in Section 6. In particular, our short-term goal is to study whether disallowing GET requests from webpages to extension schemas (Firefox disallows `XMLHttpRequest` apart from for WebExtensions, but not GET from HTML elements such as `script` and `img`, whereas Chrome allows all three) will result in breaking functionality of common extensions. Such a study may provide useful input for the future handling of extensions in Chrome and Firefox. We are also experimenting with a prototype based on Chromium to support fine-grained whitelisting policies that give the user the power to temporarily enable and disable extensions depending on what webpages are being visited.

Acknowledgments Thanks are due to Ioannis Papagiannis for the inspirations and helpful feedback. This work was partly funded by Andrei Sabelfeld's Google Faculty Research Award, Facebook Research and Academic Relations Program Gift, the European Community under the ProSecuToR project, and the Swedish research agency VR.

9. REFERENCES

[1] Detecting Chrome Extensions in 2013. http://gcattani.github.io/201303/detecting-chrome-extensions-in-2013/.

[2] Detecting Firefox Extensions Without Javascript. http://kuza55.blogspot.co.uk/2007/10/detecting-firefox-extension-without.html.

[3] Detecting FireFox Extentions. http://ha.ckers.org/blog/20060823/detecting-firefox-extentions/.

[4] Sparse Bruteforce Addon Detection. http://www.skeletonscribe.net/2011/07/sparse-bruteforce-addon-scanner.html.

[5] The Evolution of Chrome Extensions Detection. http://blog.beefproject.com/2013/04/the-evolution-of-chrome-extensions.html.

[6] Yet Another Way to Detect Internet Explorer. http://ha.ckers.org/blog/20060821/yet-another-way-to-detect-internet-explorer/.

[7] E. Abgrall, Y. Traon, M. Monperrus, S. Gombault, M. Heiderich, and A. Ribault. XSS-FP: Browser fingerprinting using HTML parser quirks. Technical report, 2012. arXiv:1211.4812 [cs].

[8] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS*, 2014.

[9] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the web for fingerprinters. In *CCS*, 2013.

[10] AdBlock. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom.

[11] V. Allaire. FuckAdBlock. https://github.com/sitexw/FuckAdBlock.

[12] Q. Cao, X. Yang, J. Yu, and C. Palow. Uncovering large groups of active malicious accounts in online social networks. In *CCS*, 2014.

[13] clsr. FuckFuckFuckAdBlock. https://gist.github.com/clsr/3f5ca796463a0e6fc8af.

[14] A. FaizKhademi, M. Zulkernine, and K. Weldemariam. FPGuard: Detection and prevention of browser fingerprinting. In *Data and Applications Security and Privacy*, 2015.

[15] http://newsroom.fb.com/company-info/#statistics.

[16] D. Fifield and S. Egelman. Fingerprinting web users through font metrics. In *Financial Cryptography and Data Security*, 2015.

[17] U. Fiore, A. Castiglione, A. De Santis, and F. Palmieri. Countering browser fingerprinting techniques: Constructing a fake profile with google chrome. In *NBiS*, 2014.

[18] Google. Chrome web store. https://chrome.google.com/webstore/category/extensions?hl=en-GB&_feature=free.

[19] Google. chrome.browserAction. https://developer.chrome.com/extensions/browserAction.

[20] Google. chrome.pageAction. https://developer.chrome.com/extensions/pageAction.

[21] Google. Content Scripts. https://developer.chrome.com/extensions/content_scripts.

[22] Google. Manifest - Web Accessible Resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[23] Google. Manifest File Format. https://developer.chrome.com/extensions/manifest.

[24] Google. Message Passing. https://developer.chrome.com/extensions/messaging.

[25] Google. Overview. https://developer.chrome.com/extensions/overview.

[26] Google Cast. https://chrome.google.com/webstore/detail/google-cast/boadgeojelhgndaghljhdicfkmllpafd.

[27] P. Gühring. Concepts against man-in-the-browser attacks. http://www.cacert.at/svn/sourcerer/CAcert/SecureClient.pdf, 2006.

[28] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May I? - Content Security Policy Endorsement for Browser Extensions. In *DIMVA*, 2015.

[29] How to detect Adblock on my website? http://stackoverflow.com/questions/4869154/how-to-detect-adblock-on-my-website.

[30] INRIA. Browser Extension Experiment. https://extensions.pet-portal.eu.

[31] N. Jagpal, E. Dingle, J. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX Sec.*, 2015.

[32] P. Laperdrix, W. Rudametkin, and B. Baudry. Mitigating browser fingerprint tracking: Multi-level reconfiguration and diversification. In *SEAMS*, 2015.

[33] I read that LastPass is vulnerable to phishing attacks - should I be concerned? https://lastpass.com/support.php?cmd=showfaq&id=10072.

[34] LastPass. https://lastpass.com/.

[35] LostPass. https://www.seancassidy.me/lostpass.html.

[36] L. Masinter. The ”data” URL scheme. http://tools.ietf.org/html/rfc2397.

[37] Mechazawa. FuckFuckAdBlock. https://github.com/Mechazawa/FuckFuckAdblock.

[38] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in JavaScript implementations. In *W2SP*, 2011.

[39] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *W2SP*, 2012.

[40] Mozilla. Chrome registration. https://developer.mozilla.org/en-US/docs/Chrome_Registration.

[41] Mozilla. Communicating using ”port”. https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Content_Scripts/using_port.

[42] Mozilla. Communicating using ”postmessage”. https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Content_Scripts/using_postMessage.

[43] Mozilla. Manifest Files. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Tutorial/Manifest_Files.

[44] Mozilla. Most Popular Extensions. https://addons.mozilla.org/en-US/firefox/extensions/?sort=users.

[45] Mozilla. options_ui. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/options_ui.

[46] Mozilla. Porting a Google Chrome extension. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Porting_a_Google_Chrome_extension.

[47] Mozilla. web_accessible_resources. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/web_accessible_resources.

[48] Mozilla. WebExtensions. https://developer.mozilla.org/en-US/Add-ons/WebExtensions.

[49] Mozilla. XUL Overlays. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Overlays.

[50] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien. Fast and reliable browser identification with JavaScript engine fingerprinting. In *W2SP*, 2013.

[51] N. Nikiforakis, W. Joosen, and B. Livshits. PriVaricator: Deceiving fingerprinters with little white lies. In *WWW*, 2015.

[52] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *S&P*, 2013.

[53] L. Olejnik, C. Castelluccia, and A. Janc. Why johnny can’t browse in peace: On the uniqueness of web browsing history patterns. In *HotPETs*, 2012.

[54] Panopticlick. https://panopticlick.eff.org/.

[55] A. Sjösten, S. Van Acker, and A. Sabelfeld. Discovering Browser Extensions via Web Accessible Resources. Full version and code. http://www.cse.chalmers.se/research/group/security/extensions.

[56] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *S&P*, 2015.

[57] C. F. Torres, H. Jonker, and S. Mauw. FP-block: Usable web privacy by controlling browser fingerprinting. In *ESORICS*, 2015.

[58] R. Upathilake, Y. Li, and A. Matrawy. A classification of web browser fingerprinting techniques. In *NTMS*, 2015.

[59] S. Yokoyama and R. Uda. A proposal of preventive measure of pursuit using a browser fingerprint. In *IMCOM*, 2015.