Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks

Alexander Sjösten*, Steven Van Acker*, Pablo Picazo-Sanchez and Andrei Sabelfeld
Chalmers University of Technology
{sjosten, acker, pablop, andrei}@chalmers.se

Abstract—Browser extensions enable rich experience for the users of today's web. Being deployed with elevated privileges, extensions are given the power to overrule web pages. As a result, web pages often seek to detect the installed extensions, sometimes for benign adoption of their behavior but sometimes as part of privacy-violating user fingerprinting. Researchers have studied a class of attacks that allow detecting extensions by *probing* for Web Accessible Resources (WARs) via URLs that include public extension IDs. Realizing privacy risks associated with WARs, Firefox has recently moved to randomize a browser extension's ID, prompting the Chrome team to plan for following the same path. However, rather than mitigating the issue, the randomized IDs can in fact exacerbate the extension detection problem, enabling attackers to use a randomized ID as a reliable fingerprint of a user. We study a class of extension *revelation* attacks, where extensions reveal themselves by injecting their code on web pages. We demonstrate how a combination of revelation and probing can uniquely identify 90% out of all extensions injecting content, in spite of a randomization scheme. We perform a series of large-scale studies to estimate possible implications of both classes of attacks. As a countermeasure, we propose a browser-based mechanism that enables control over which extensions are loaded on which web pages and present a proof of concept implementation which blocks both classes of attacks.

I. Introduction

Browser extensions, or simply extensions, enable rich experience for the users of today's web. Since the introduction of browser extensions in Microsoft Internet Explorer 5 in 1999 [42], they have been an important tool to customize the browsing experience for all major browser vendors. Today, the most popular extensions have millions of users, e.g. AdBlock [10] has over 10,000,000 downloads in the Chrome Web Store [24]. All major web browsers now support browser extensions. Mozilla and Chrome provide popular platforms for browser extensions, with Mozilla having over 11.78%, and Chrome over 66.1% of the browser's market share (April 2018) [57].

Power of extensions: Firefox and Chrome provide their extensions with elevated privileges [41]. As such, the extensions have access to a vast amount of information, such as reading and modifying the network traffic, the ability to make arbitrary modifications to the *Document Object Model (DOM)*, or having the possibility to access a user's private information from the browsing history or the cookies. The extension models for both Firefox and Chrome allow extensions to read and modify the DOM of the currently loaded web page [44], [26]. In addition to the aforementioned scenarios, some browser extensions like password managers, have access to sensitive data such as the user's passwords, which can include credentials to email accounts or social networks.

Detecting extensions: Due to the increased power that browser extensions possess, they have been target for detection from web pages. Today, Chrome comes with a built-in ChromeCast extension [31], which has *Web Accessible Resources (WARs)*, public files which exist in the extension and can be accessible from the context of the web page. Web pages, such as video streaming pages, can then probe for the ChromeCast extension, and add a cast button which would allow to cast the video player to the connected ChromeCast. By doing this, the browsing experience of the user is improved. On the other side, a web page might want to prevent DOM modifications (e.g. by detecting ad blockers), prepare for an attack against the user of a browser extension with sensitive information (e.g. by performing a phishing attack [16]), or even to gain access to the elevated APIs the browser extension has access to [3]. With the possibility of detecting browser extensions by web pages, users can be tracked based on their installed browser extensions [22], [55], [53]. This motivates the focus of this paper on the problem of protecting browser extensions from detection attacks.

Probing attack: Previous works [55], [53] have focused on *non-behavioral detection*, based on a browser extension's listed WARs. The WARs are public resources which can be fetched from the context of a web page using a predefined URL, consisting of a public extension ID (or *Universally Unique Identifier (UUID)*) and the path to that resource. With the predefined URL to fetch a WAR from an extension, a web page can mount a *probing attack*, designed to detect an extension by probing for WARs, since a response with the probed WAR indicates the corresponding extension is installed. This attack can be seen in Figure 1a where ① denotes the requests made by the attacker to probe for an installed browser extension. If the browser extension is in the browser context, the attacker will get a response consisting of the requested WAR (denoted by ②). This attack can be magnified by probing for a set of browser extensions' resources, thereby enumerating

*These authors contributed equally.

(a) Probing attack.

(b) Revelation attack.

Fig. 1: Schematic overview of the extension probing attack and extension revelation attacks. In the probing attack, a web page probes for the presence of an extension. In the revelation attack, the extension reveals itself to the attacker by injecting content in the web page.

many or even all installed browser extensions.

Firefox defense against probing: As the probing attack is possible when the URLs of a browser extension's WARs are fixed and known beforehand, Firefox implements a randomization scheme for the WAR URLs in their new browser extension model, *WebExtensions*. To make the probing attack infeasible, each browser extension is given a random UUID, as it *"prevents websites from fingerprinting a browser by examining the extensions it has installed"* [50]. The Chrome developers are considering to implement a similar randomization scheme, when they have *"the opportunity to make a breaking change"* [8].

Revelation attack: Starov and Nikiforakis [56] show that browser extensions can introduce unique DOM modifications, which allows an attacker to determine which extension is active based on the DOM modification. In contrast to probing attacks, these attacks are *behavioral* attacks because they are based on detecting behavior of a browser extension via, e.g., DOM modifications.

This work puts the spotlight on *revelation attacks*, an important subclass of behavioral attacks, first introduced by Sánchez-Rola et al. in the context of Safari extensions [53]. The core of a revelation attack is to trick an extension to inject content via WAR URLs, thereby giving up its random UUID and provide a unique identifier of the victim. This attack is displayed in Figure 1b. When the WAR is injected by the browser extension (①), the URL with the random UUID becomes known to the attacker, who is monitoring changes to the web page through JavaScript. With the random UUID known, an attacker can construct WAR URLs to known resources by initiating a probing attack (② and ③). The probing in this case will be done for known unique resources for browser extensions which have the injected WAR as a resource, a set which can be precomputed by the attacker. Upon finding one of the resources in this precomputed set, the attacker can deduce which browser extension injected the information, allowing derandomization of browser extensions.

Starov and Nikiforakis [56] show that browser extensions can provide unique DOM modifications, allowing an attacker to determine the active extension. However, it is not possible to uniquely identify the victim only based on the browser extensions [33]. This is the crucial part of the revelation attack: as the random UUID becomes known to the attacker, it enables them to uniquely identify the victim, based on that installed extension alone. Furthermore, in most cases these random WAR URLs can easily be used to derandomize an extension, indicating the UUID randomization does not prevent extension fingerprinting. In fact, since a malicious web page in many situations can not only figure out which browser extension has the random UUID, but also uniquely identify the user, the randomization of UUIDs amplifies the effect of a revelation attack rather than mitigating detection possibilities. The problem with randomization of UUIDs is known, and has been a topic of discussions among browser developers [1], as well as presented as an attack against a built-in browser extension which takes screenshots for Firefox [13]. Although this attack requires user interaction, it is important to study how many of the Firefox and Chrome extensions can be exploited without the need for user interaction.

Empirical studies: To see how many extensions are susceptible to the revelation attack without user interaction, and how many web pages probe for extensions, we conduct several empirical studies.

- We download all extensions for Firefox and Chrome and determine that, in theory, 1,301 ($\approx$94.41%) and 10,459 ($\approx$89.91%) of the Firefox and Chrome extensions respectively that might inject content are susceptible to the revelation attack.
- We check how many of the extensions susceptible to the revelation attack actually reveal themselves, where the attacker model is a generic web developer with the ability to host a web page visited by the victim. While the victim is on the attacker web page, the attacker will attempt to make the installed browser extensions inject content to make them reveal themselves, with the hope of determining exactly which browser extensions are being executed based on the injected content. If the randomized token proves stable enough, the attacker may also use it to track the victim on the Web. This attacker model fits a wide range of possible attackers, from small and obscure web pages, to top-ranked web applications. To emulate this, we check how many extensions reveal themselves based on where the extension is defined to inject content, and whether the actual content on the web page matters, showing that 2,906 out of 13,011 ($\approx$22.3%) extensions

reveal themselves on actual pages.

- We visit the most popular 20 web pages for each of the Alexa top 10,000 domains, and find that 2,572 out of those 10,000 domains probe for WARs.

"Latex Gloves" mitigation approach: In popular culture, crime scene investigators frequently use latex gloves to avoid contaminating a crime scene with fingerprints. In this work, our goal is to prevent that extensions leave any "fingerprints" that are detectable by an attacker web page, be it through a probing attack or a revelation attack. For this reason, we named our approach "Latex Gloves" for extensions.

A key feature of our approach is its generality. The mechanism is parametric in how whitelists (or, dually, blacklists) are defined, with possibilities of both web pages and extensions having their say. Extension manifest files can be used for automatic generation of whitelists already. While it might be suitable to let the advanced user affecting the whitelists, the goal is to relieve the average user from understanding the workings and effects of web pages and browser extensions. For the whitelist, which defines which extensions are allowed to reveal themselves to the web page, there are several options, each with its own benefits and drawbacks. For example, a mechanism similar to Google Safe Browsing [28] can be employed, where browser vendors can provide blacklists for our mechanism containing web pages known to perform extension fingerprinting. This would put the burden on the browser vendors to keep the blacklist up to date. Another option would be to allow web pages to specify a whitelist, similar to how a *Content Security Policy (CSP)* [58] is defined. Naturally, there is a big risk web pages would simply try to deny all extensions any access, greatly limiting a user's intentions. Another option is a simple interface that allows users to classify websites into sensitive (e.g., bank) and insensitive (e.g., news portal), so that it is possible to configure whether an extension is triggered on a(n) (in)sensitive website. Yet another option is an all-or-nothing policy: either all extensions are triggered on all insensitive websites or no extensions are triggered on any sensitive websites. This would keep interaction with the user to a minimum. Each option has advantages and disadvantages, and usability studies can help determine the most suitable alternatives.

Our vision is to have direct browser support for Latex Gloves. However, in order to aid evaluation of the general mechanism, we present a proof-of-concept prototype consisting of a Chromium browser modification, a Chrome extension and a web proxy. This prototype allows the whitelisting of those web pages that are allowed to probe for extensions, and the whitelisting of those extensions that are allowed to reveal themselves to web pages.

Contributions: In this work, we present the first large-scale empirical study of browser extensions on both Firefox and Chrome based on the revelation attack, in order to determine how fingerprintable the browser extensions — and the users of browser extensions — are, in the presence of a random WAR URL scheme. Additionally, we propose a countermeasure based on two whitelists, defining which web pages may interact with which extensions and vice versa, thus allowing users to avoid being fingerprinted or tracked by untrusted web sites. We finally give some guidelines to avoid this security issue for browser developers.

The main contributions of this paper are:

Revelation attack on Firefox. We demonstrate how to de-randomize Firefox extensions through revelation attacks (Section IV).

Empirical studies of Firefox and Chrome extensions. We present large-scale empirical studies of Firefox and Chrome extensions regarding revelation attacks (Section IV), where we determine how $\approx 90\%$ out of all extensions injecting content can be uniquely identified in spite of a randomization scheme, as well as evaluating how many extensions can be detected with a revelation attack, based on the attacker model.

Empirical study of the Alexa top 10,000. We report on an empirical study over the Alexa top 10,000 domains, with up to 20 of the most popular pages per domain to determine how widely the probing attack (Section III) is used on the Web.

Resetting Firefox random UUID. We investigate the user actions required to reset the random UUID of a Firefox extension, in order to remove a unique fingerprint accidentally introduced by Mozilla, on the most prominent operating systems: Windows, Mac OSX and Linux.

Design of a mechanism against the two attacks. We give the design for "Latex Gloves" (Section V), a mechanism against both probing and revelation attacks using whitelists to specify which web sites are allowed to interact with which extension's WARs, and which extensions are allowed to interact with which web sites.

Proof of concept prototype. We implement a proof of concept prototype (Section VI) consisting of a modified Chromium browser, a browser extension and a web proxy, all based on the whitelisting mechanism. Our prototype is evaluated (Section VII) against two known attacks (extension enumeration [55] and timing attack [53]).

Recommendations for browser developers. We use key insights from our empirical studies to give recommendations (Section VIII) to browser developers for a browser extension resource URL scheme.

II. BACKGROUND

An extension is a program, typically written in a combination of JavaScript, HTML and CSS. Browser extensions have become a vital piece in the modern browser as they allow users to customize their browsing experience by enriching the browser functionality, e.g. by altering the DOM or executing arbitrary scripts in the context of a web page.

JavaScript code in a browser extension can roughly be classified as *background pages* and *content scripts*. Background pages are executed in the browser context and cannot access the DOM of the web page. Instead, they are allowed to access the same resources as the browser, e.g. cookies, history, web requests, tabs and geolocation. However, in order to make use of these capabilities the user has to explicitly grant most of them.

Content scripts are files that is executed in the context of a web page. Although the content scripts live in isolated worlds, allowing them to make changes to their JavaScript environment without conflicting with the web page or any other content scripts, they have access to the same DOM structure

```json
{
  "manifest_version": 2,
  "name": "Example",
  "version": "1.0",
  "background": {
    "scripts": ["background.js"]
  },
  "content_scripts": [
    {
      "matches": ["*://*.example.com/*"],
      "js": ["content_script.js"]
    }
  ],
  "web_accessible_resources": [
    "images/img.png",
    "scripts/myscript.js"
  ],
  "permissions": ["webRequest"]
}
```

Fig. 2: Example of a *manifest.json* file

as the main web content. As content scripts are executed in the context of the web page, the content scripts can read and modify the DOM of the web page the browser is visiting, as well as inject data such as images and other scripts into the web page [44], [26]. Content scripts can only use a subset of the extension API calls ("extension", "i18n", "runtime" and "storage"), neither of which need approval from the user. In case the content scripts need access to more privileged extension APIs, they can only access them indirectly by communicating with the background pages through message passing. As the access of the privileged API calls goes through the background page via message passing, the user must approve them upon installing the extension.

The structure of an extension is defined in a *manifest file*, called *manifest.json*, which is a mandatory file placed in the extension's root folder [46], [30]. The manifest file contains, among other things, which files belong to the background page, which files belong to the content script, which permissions the extension requires, and which resources can be injected into the web page. An example of a manifest file can be seen in Figure 2, which specifies the background page to be the JavaScript file `background.js` and the content script (`content_scripts`) to use the JavaScript file `content_script.js`, and be executed on all domains that matches the domain `example.com`. It defines two WARs (`web_accessible_resources`), which are resources that can be injected into the web page from the content script. The path for the WARs is the path from the extension's root folder to the resources. The extension also asks for the permission `webRequest`, which indicates the extension's background page want the ability to intercept, block and modify web requests.

Browser extensions scope: In the particular case of content scripts, browser extensions insert their JavaScript files in those web pages explicitly defined by the extension's developers in the manifest file. Concretely, there is a mandatory property named `matches` which indicate the web pages the content script should be injected into. URLs can be defined

following a match pattern syntax, which is reminiscent of regular expressions, operating on a `<scheme>://<host><path>` pattern [18]. Background pages are not affected by the `matches` property. Instead, they remain idle until a JavaScript event such as a network request or message passing coming from an arbitrary content script, triggers their code, after which they return to an idle state.

Web Accessible Resources: If an extension wants to inject a resource, such as an image or a script, into a web page, the recommended way is to make the resource "web accessible". WARs are files that exist in a browser extension but can be used in the context of a web page. A browser extension must explicitly list all WARs through the `web_accessible_resources` property in the manifest file [50], [29].

WAR URLs are different for Firefox and Chrome: `moz-extension://<ext-UUID>/<path>` and `chrome-extension://<ext-UUID>/<path>` in Firefox and Chrome, respectively. In Firefox, `<ext-UUID>` is a randomly generated UUID for each browser instance, and is generated when the extension is installed [50]. However, for Chrome, `<ext-UUID>` is a publicly known 32 character string derived from the RSA public key with which the extension is signed, encoded using the "mpdecimal" scheme. WAR URLs in Chrome have the `<ext-UUID>` hardcoded as the "hostname" part. For both Firefox and Chrome, the recommended way of getting the URL of the resource is to use the built-in API, which is `browser.extension.getURL("path")` in the case of Firefox [45], and `chrome.runtime.getURL("path")` for Chrome [25]. Since Chrome extensions have a publicly known extension UUID, an attacker could enumerate all installed extensions which have WARs (See Section III).

Browser profiles and extension UUIDs: In Chrome and Firefox, data such as bookmarks, passwords and installed extensions is stored in a *browser profile* [49]. A browser installation may have several browser profiles, each with its own data. Because Firefox's extension UUIDs are randomized, the same extension installed in multiple browser profiles will have a different UUID for each profile. In Chrome, which uses fixed extension UUIDs, an extension installed in multiple browser profiles will use the same extension UUID in each profile.

III. PROBING ATTACK

When probing for an extension, JavaScript running in a web page tries to determine the presence of a browser extension in the browser in which the web page has been loaded.

One way of performing the extension probing is by requesting a browser extension's WARs through the publicly known URLs for these resources. This is schematically shown in Figure 1a where ① denotes the request made by the web page to probe for a browser extension's WAR. A successful response to this request (denoted by ②) indicates the presence of the extension to which the WAR belongs.

Probing for an extension in itself does not mean an attack is taking place. It is not an attack if, e.g., Google probes

TABLE I: Alexa top 10,000 domains probing for Chrome extensions. Note that a domain may appear in several rows and/or columns.

	same domain	other domain	YouTube
top frame	185	15	4
sub frame	36	2,399	2,277
Total		2,572	

for the ChromeCast extension on YouTube.com since this is the extension developer who probes for their own extension. However, if it is not the extension developer who is probing for the browser extension, but rather a third party with the intent of discovering installed extensions to, e.g., increase the entropy for browser fingerprinting, the probing becomes a *probing attack*. Attackers may use a probing attack to detect the presence of any of the known browser extensions, thereby enumerating the installed browser extensions in a victim's browser.

Sjösten et al. [55] explore the Alexa top 100,000 domains to examine how many of them probe for WARs on their front page and their reasons for doing so. Their research shows that web developers and their applications may probe for WARs for legitimate reasons. They find only 66 domains, none in the top 10,000, and surmise that this is caused by the technique not being widely known.

We repeat the experiment using a different detection method, in order to study how this problem has developed over time. Instead of the top 100,000, we limit ourselves to the top 10,000, but perform a deeper study by visiting up to twenty of the most popular web pages on each domain. We also gather metrics that indicate whether the probing is due to a third-party web origin, or whether it originates from the domain itself.

Setup: We use a modified version of Chromium 63.0.3239.84, which allows us to monitor requests for WAR URLs from a Chrome extension, as described in Section VI. The entire process is automated using Selenium 3.8.1.

When visiting a web page, we wait for up to 10 minutes for the web page to load. Once loaded, we wait an additional 20 seconds in order for any JavaScript on the web page to execute.

During this time, a custom browser extension monitors any requests made towards `chrome-extension://` URLs and logs them. In addition to the WAR URL itself, we also log whether the request came from the parent frame or a sub frame, as well as the web origin from which the request occurred.

Results: Starting from the list of top 10,000 domains according to Alexa, we queried Bing to retrieve the most popular twenty pages per domain. Bing returned 180,471 URLs for 9,640 domains. We further disregard domains for which Bing did not return any results. Of the 180,471 URLs, we were able to visit 179,952 spread over 9,639 domains.

An overview of the results is shown in Table I. In total, out of the 10,000 domains, 2,572 probed for 45 different extensions from either the top frame or a sub frame. Of the domains that requested a WAR from the top frame, 185 had not redirected the browser to another domain, while 15 did.

In the latter case, 4 redirected to YouTube.com. In the other cases, WARs were requested from a sub frame: 36 domains loaded the sub frame from the same domain, while 2,399 loaded it from a third-party domain. Strikingly, 2,277 of those sub frames originated on YouTube.com where most of these requests were probing for the ChromeCast browser extension.

Our results are different from Sjösten et al. [55], which may be attributed to the different methodology or an increase in extension probing. No matter the reason for the discrepancy, probing is both common and relevant. Although YouTube.com probing for ChromeCast is not a probing attack, most of the remaining extensions being probed for (e.g. popular extensions such as AdBlock [10], AdBlock Plus [2] and Ghostery [6]) constitute probing attacks.

IV. REVELATION ATTACK

In an effort to eliminate the extension probing attack, Mozilla implemented a randomization scheme in its extensions' UUIDs. Since each extension is given a random UUID upon installation, it is impossible to compose the URL of a WAR to launch a probing attack without knowing that random UUID. However, it is possible for an attacker to learn the random UUID of an extension through an extension revelation attack.

In an extension revelation attack, JavaScript running in a web page tries to determine the presence of a browser extension by monitoring the web page for new content which references WARs. Although any introduced DOM modification might uniquely identify an extension [56], an injected WAR URL contain a unique UUID for each profile, which in turn can be used to track users. Also, due to the nature of the WAR URLs, a vast majority of all extensions injecting content with WAR URLs can still be uniquely identifiable, in spite of the randomization scheme, indicating it might make more harm than good.

Figure 1b displays the revelation attack. JavaScript in a web page detects that a browser extension has inserted a reference to a WAR (①), and can now deduce the presence of this extension.

In the case of Firefox, the revelation attack reveals a WAR URL, which consists of a random UUID and a path component. While the random UUID itself is insufficient to derandomize the extension, it can be used as a basis for a probing attack (② and ③).

It is important to realize that a probing attack may not be needed in order to derandomize Firefox's random UUIDs. In Section IV-A, we show that the path component of the WAR URL, which is not randomized in Firefox, contains enough information to derandomize an extension's random UUID in many cases. In addition, because an attacker can retrieve the content of a WAR and compute a hash over it, it is possible to derandomize an extension even if the full WAR URL is randomized.

Furthermore, because the random UUID is unique per "browser instance", it can also be used as a unique fingerprint to deanonymize web users through the revelation attack. As we show in Section IV-B, it is not trivial to remove this unique fingerprint from the browser.

The developers of Google's Chrome browser have expressed interest in implementing a similar randomization scheme [8]. In Section IV-C, we study the impact of adopting this randomization scheme on Chrome extensions. The results of both Section IV-A and Section IV-C are summarized in Table II, where "Path" is the amount of extensions that can be derandomized based on the path, "Hash" based on the sha256 hash digest of the content of the WARs, and "Path ∪ Hash" the union of those sets.

Finally, in Section IV-D we perform an empirical study of all available Firefox and Chrome extensions to determine how many of them are affected by the revelation attack, revealing themselves and their users to attackers simply by visiting an attacker's web page.

A. Derandomizing Firefox extensions

Since Firefox employs random UUIDs, the enumeration techniques presented in [55], [53] cannot be used. Instead, the extension must reveal itself for an attacker to get hold of the random UUID. In order to derandomize a Firefox extension, the extension must meet the following criteria. First, the extension must have at least one defined WAR, indicating it might inject a resource. Second, the extension must make a call to either of the functions `browser.extension.getURL`, `chrome.extension.getURL` or `chrome.runtime.getURL`, which are functions that, given an absolute path from the root of the extension to the WAR, will return the full `moz-extension://<ext-UUID>/<path>` URL. For the rest of this section, we will group those functions together as `getURL()`. Although these API functions are executed in the context of the extension, i.e. they cannot be called directly from the web page, if the extension injects the WAR in this manner, the random UUID will be revealed to the web page as part of the WAR URL. If this happens, and the attacker gets the UUID, then how many extensions can be uniquely identified based on the injected WAR URL?

To determine this, we scraped and downloaded all free Firefox extensions from the Mozilla add-on store [47]. The extensions are valid for Firefox 57 and above, as it is the first Firefox version to only support WebExtensions [51], indicating all will receive a random UUID when installed. The scrape was done on February 23, 2018, giving us 8,646 extensions. All of these extensions were unpacked, and their manifest file examined for the `web_accessible_resources` key, resulting in 1,742 extensions having at least one defined WAR. The mere presence of a WAR in an extension does not mean that this resource will ever be injected. We took the 1,742 extensions with declared WARs, and checked how many of them call a `getURL()` function, as this will construct the WAR URL to be injected to the web page. This resulted in a total of 1,378 extensions, indicating ≈79.10% of all Firefox extensions with declared WARs can reveal their random UUID.

Having access to only the random UUID is not sufficient. The path component present in a WAR URL can give away the identity of the extension, if there is a mapping between a path and the corresponding extension. Out of the 1,378 extensions that call a `getURL()` function, 1,107 extensions provide at least one unique path, i.e. the full path to a resource. Aside from the WAR URL, a potential attacker also has access to

TABLE II: Breakdown of the uniqueness detectability for browser extensions, assuming a randomized schema with the ability to probe.

	Extensions total	Path	Hash	Path ∪ Hash
Firefox	1,378	1,107 (80.33%)	1,292 (93.76%)	1,301 (94.41%)
Chrome	11,633	7,214 (62.01%)	10,355 (89.01%)	10,459 (89.91%)
Total	13,011	8,321 (63.95%)	11,647 (89.52%)	11,760 (90.39%)

the contents of the WAR. We investigated the contents of the extensions' WARs to determine how unique they are by calculating a hash digest over the contents. A total of 1,292 browser extensions have a unique digest when hashing their WARs, where a different hash digest indicate a difference in content between the WARs of the different browser extensions. We then took the union of the two sets of browser extensions with at least one unique path and a unique digest, yielding a total of 1,301 browser extensions to be uniquely identifiable. Although only ≈15.05% of all extensions can be uniquely identified, it is ≈94.41% of all extensions that have the possibility to inject a WAR.

B. Resetting Firefox's random UUID

For Firefox, each UUID is "randomly generated for every browser instance" [50]. However, it is not clear what "browser instance" means in this setting. In order to determine when the random UUID of a browser extension is being reset in Firefox, we tried different approaches on three operating systems: Windows 10, Linux (Debian) and Mac OSX. The approaches were restarting, updating and re-installing the browser, updating and re-installing the extension, switching the browser tab to incognito mode and clearing the cache and cookies of the browser. The result can be found in Table III, and for the rest of this subsection, we will briefly cover the differences between the operating systems.

None of the operating systems change the internal UUIDs upon restarting the browser, indicating "browser instance" from the documentation does not mean "started browser process". When re-installing the browser, the default behavior for the Windows 10 installer is to reset the standard options, which includes removing the old browser extensions. As this would force a user to re-install the browser extensions, each browser extension would get a new random UUID. However, a user has the option of not resetting the standard options, along with not removing the old browser extensions. Hence, uninstalling Firefox on Windows keeps all settings, and it is up to the user to decide to keep or remove them when re-installing the browser. This is not the case for Linux and Mac OSX. For both operating systems, it is up to the user to manually remove the profile folder (default is `.mozilla` in the home folder for Linux, and `Library/Application Support/Firefox` in Mac OSX) in order to remove the old browser extensions upon re-installing the browser, as they are not prompted about a default option of resetting the standard options.

For all operating systems, the UUID was regenerated when reinstalling the extension, given that the browser was restarted between uninstalling and reinstalling the extension. If the browser was not restarted, the profile file containing the data would not change, giving the new installation the same UUID.

TABLE III: Actions which result in UUID regeneration for each of the major operating systems. "Yes" or "No" means that the action did or did not cause UUID regeneration respectively. Notes: (*) Firefox's installer on Windows prompts the user to reset settings and remove extensions, which is enabled by default, whereas for Linux and Mac OSX (+), the default is to keep all settings.

		Linux	Mac OSX	Windows
Restarting browser			No	
Updating browser			No	
Re-installing browser		No+		Yes*
Updating extension			No	
Re-installing extension	w/ browser restart		Yes	
	w/o browser restart		No	
Incognito mode			No	
Clearing cache and cookies			No	
Clearing the profile			Yes	

On all platforms, clearing the profile (i.e. removing the actual profile folders) would force a user to re-install all extensions, which means they would get a new random UUID.

C. Derandomizing Chrome extensions

As Chrome does not employ random UUIDs, the technique presented by Sjösten et al. [55] still works. However, as Chromium developers plan to employ random UUIDs, we performed the same experiment as for Firefox. In total, we scraped 62,994 free extensions from the Chrome Web Store [24]. Out of those, 16,280 defined `web_accessible_resources` with at least one corresponding WAR. The amount of extension that called either `chrome.runtime.getURL` or `chrome.extension.getURL` was 10,764. We also checked the extensions that called `chrome.runtime.id` (728 extensions), which return the extension's UUID, and the ones that hardcoded their extension UUID into a resource URL (141 extensions), with the assumption they will change to call `getURL()` if Chrome adopts random UUIDs. With this, the total amount of detectable extensions would be 11,633 extensions, which corresponds to ≈71.46% of all extensions with at least one WAR declared. Assuming random UUIDs for Chrome, we must check if a path can uniquely identify an extension. We applied the same uniqueness procedure as in Section IV-A, finding 7,214 extensions being unique without the need for any content hashing. When hashing the content of the WARs, we got a total of 10,355 browser extensions, and the union of those two sets yield a total of 10,459 uniquely identifiable browser extensions. While only being ≈16.60% of all extensions, it is ≈89.91% of all browser extensions that have the possibility to inject a WAR.

D. Extensions revealing themselves to web pages

As browser extensions can inject WARs into a web page to allow it access in the domain of the web page, the WARs are visible to JavaScript executed in the origin of this web page. A web page can scan for these WARs in order to reveal installed browser extensions, as well as to deanonymize the visitor: from the WARs, an attacker can infer the installed extension, and from Firefox browser extensions' random UUIDs, the attacker can identify the visitor.

For this experiment, we consider all 8,646 Firefox extensions, but are also interested in the 62,994 Chrome extensions.

As Chrome are considering random UUIDs, the findings are relevant to their future development plans.

Setup: We use Selenium 3.9.1 with Firefox 58.0.1 and Chromium 64.0.3282.167 to automate the process.

For each browser extension, we visit a web page through mitmproxy 2.0.2 [21] with a custom addon script. In order to be able to manipulate web pages served over HTTPS, both Firefox and Chromium were configured to allow untrusted SSL certificates.

The mitmproxy addon script injects a piece of attacker JavaScript code in the web page which walks through the HTML tree and extracts any attributes that contain `chrome-extension://` or `moz-extension://` present in the web page. In addition, because the CSP may prevent the execution of injected JavaScript, the mitmproxy addon script disables CSP if present.

Because browser extensions may inject content only after a while, the attacker script also installs a mutation observer which repeats the scan every time a change to the web page is detected. With this setup, we can detect the injection of WARs at any point in the web page's lifetime. For every page visit, we wait for up to one minute for the page to load before aborting that page visit. When a page is successfully loaded, we wait for five seconds to let any JavaScript on the page run its course.

Dataset extensions: Because of the way Firefox extensions work, we only consider those extensions which seemingly make a call to `getURL()` and which have web accessible resources. After this filtering step, 1,378 out of the 8,646 Firefox extensions remain for our study.

Similarly for Chrome, we retain 11,633 out of the total 62,994 Chrome extensions.

Dataset URLs: These 13,011 extensions (1,378 Firefox + 11,633 Chrome) will only execute on a web page if the URL matches the regular expressions in their manifest file. For instance, an extension which lists `http://example.com/*` in its manifest file, will not execute when visiting, e.g., `http://attacker.invalid/index.html`. Extensions can only reveal themselves when they are executing on a web page they were designed for, e.g by checking for the presence of a certain keyword in the URL. Because of this, it is important to visit the right URLs.

To determine the set of URLs we should visit for a particular extension, we make use of the CommonCrawl dataset [5]. This dataset contains data about ≈4.57 billion URLs from a wide variety of domains. From the 13,011 extensions, we extracted 24,398 unique regular expressions and matched them against the CommonCrawl dataset using the regular expression matching rules specific to the manifest file specification. For each regular expression, we only consider the first 100 matches. For each extension, which can have many regular expressions in its manifest, we combine all matching URLs and take a random subset of maximum 1,000 URLs. In total we obtained 506,215 unique URLs from the CommonCrawl dataset that match the regular expressions from the extensions' manifest files. We call this set of URLs the "real" URLs.

From the "real" URLs, we derive two extra sets of URLs by considering that an attacker can host a copy of a real web page

on a different web host. For instance, the web page at `http://www.example.com/abc` could be hosted on an attacker-controlled `http://www.attacker.invalid/abc`. We call this cloned set of "real" URLs, where the hostname has been replaced by `attacker.invalid`, the "attackerhost" URLs.

Extensions with more fine-grained regular expressions may require the attacker to register a domain in DNS. For instance, a regular expression `http://*.com/abc` does not match the `attacker.invalid` domain which we assume is under attacker control. Therefore, we also consider a URL set where the hostname in each URL has been replaced by a hostname with the same top-level domain, but with an attacker-controlled domain name. For instance, for `http://www.example.com/abc` we also consider `http://www.attacker.com/abc`. Naturally, we chose a domain name of sufficient length and consisting of random letters, to make sure it was not registered yet. We call this cloned set of "real" URLs, the "buydns" URLs.

In addition to the real CommonCrawl URLs which match the regular expressions, we also generate URLs based on those regular expressions by replacing all "*" characters with "anystring". For instance, we generate the URL `http://*.example.com/anystring` for the regular expression `http://*.example.com/*`. We call this set of URLs the "generated" URLs.

Dataset web page content: Aside from expecting a certain URL, an extension may also depend on certain HTML elements, HTML structure or particular text present on a visited web page. To determine whether this is the case, each web page visited through a URL in the "real" URLs set, as well as the derived "attackerhost" and "buydns" sets, is also visited with all content removed. We visit each of these URLs twice: once with the real content, and once serving an empty page instead of the real content. For the "generated" URL set, we only serve empty pages, since there is no way to determine what type of content should be present on such a URL. A known practice from previous work is to use "Honey Pages", empty pages that create the DOM content of a web page dynamically, based on what the extension is querying [56], [35]. While "Honey Pages" can provide useful information to, e.g., find malicious extensions, some extension behavior can be difficult to trigger in an automated way, as it may not be only nested DOM structures, but also events an extension acts on. In this light, "Honey Pages" may not be representative of the operation of actual web pages. As we are interested in whether web pages would be able to employ a revelation attack with their current structure, our experiments are not using "Honey Pages". Instead, we look at the current interaction between web pages and extensions, providing an indication of how many extensions that are currently vulnerable. For the best coverage, it would be interesting to combine our results with "Honey Pages", but we leave that for future work.

Results: The results of the experiment are shown in Tables IV to VI.

Out of 13,011 extensions, 2,906 revealed themselves on actual pages. We suppose this behavior is intentional, but it can be abused by the website owners to track the users. 9,543 did not reveal themselves and 562 could not be used in our experiment because of issues with the third-party software we

TABLE IV: Breakdown of Chrome and Firefox extensions, indicating which how many extensions revealed themselves, how many didn't, and how many we were unable to analyze (broken).

	Revealed	Broken	Not revealed	Total
Chromium	2,684	412	8,537	11,633
Firefox	222	150	1,006	1,378
Total	2,906	562	9,543	13,011

used in our setup (Selenium, browser-specific or addon-specific issues).

The other remaining 9,543 extensions which call `getURL()` and have WARs, seemingly do not inject any WARs into the web page, or probably more accurately: we did not trigger the correct code path in the extension that results in a WAR being injected into a web page. Analyzing these remaining extensions via "Honey Pages" could reveal they also inject WARs under the right circumstances, although none of the web pages we visited would make them inject content. Nevertheless, our analysis of web page and extension interaction succeeded in exposing 2,906 extensions which reveal themselves on web pages.

Of these 2,906 extensions triggered by real URLs, 2,330 depend only on the URL of the web page visited, and do not depend on the content of that page, since they execute even when the presented web page is empty. Moreover, out of the 2,906 extensions that reveal themselves on the right URLs, 1,149 can be tricked into executing on attacker-controlled web pages. Only for 6 Chrome extensions (but none of the Firefox extensions) does the attacker potentially have to register a new domain to host the malicious website on.

Moreover, for 1,149 of the extensions that can be tricked to execute on an attacker URL, 911 do not depend on the page content, further easing the life of the attacker.

The numbers between brackets in Table V denotes the total number of extension users affected by these revealing extensions. Assuming there are no overlaps between the users of the revealing extensions, a total of 38,604,160 web users are vulnerable to the revelation attack through their installed extensions. For the 792,038 affected Firefox users, this means that they are uniquely identifiable through the unique fingerprint exposed by their revealing extensions. The 37,812,122 affected Chrome users do not suffer from this issue at this point in time, but would also be uniquely identifiable if the Google Chrome developers adopt Firefox's UUID randomization scheme.

Furthermore, as seen in Table VI, out of the 2,906 revealing extensions, 2,261 have at least one unique path, and 2,819 have at least one WAR with a unique content. The union of those sets contains 2,822 extensions, indicating that 97.11% of the 2,906 (97.09% of Chrome and 97.30% of Firefox) revealing extensions can be uniquely identified.

V. MITIGATION DESIGN

From the introductory example in Section I, it is clear that there is a legitimate use-case for being able to probe for WARs. Extensions that want to be detectable through their WARs, e.g. ChromeCast, would become dysfunctional if probing for

	Content-dependent						Any content						Total	
	"real" URL		"attackerhost" URL		"buydns" URL		"real" URL		"attackerhost" URL		"buydns" URL			
Chromium	289	(3,227,947)	217	(2,680,324)	2	(110)	1,281	(17,301,512)	891	(14,601,057)	4	(1,172)	2,684	(37,812,122)
Firefox	49	(39,780)	19	(75,940)	0	(0)	138	(649,236)	16	(27,082)	0	(0)	222	(792,038)
Either browser	338	(3,267,727)	236	(2,756,264)	2	(110)	1,419	(17,950,748)	907	(14,628,139)	4	(1,172)	2,906	(38,604,160)

TABLE VI: Breakdown of revealing Chrome and Firefox extensions, indicating how many of the extensions revealing themselves that could be uniquely identified, either through the path, through the content of the WARs, and the union of those sets.

	Revealed	Unique path	Unique hash	Unique path ∪ hash
Chromium	2,684	2,063	2,603	2,606 (97.09%)
Firefox	222	198	216	216 (97.30%)
Total	2,906	2,261	2,819	2,822 (97.11%)

WARs was blocked in general. Therefore, preventing the extension probing attack through a blanket ban on extension probing, is not an option.

In similar vein, preventing extensions from revealing themselves to web pages is also not an option. The data from Section IV-A implies that many extensions may inject content into a web page, and could become dysfunctional if this functionality was no longer available. Extensions ill intent on revealing themselves may be unstoppable, and we consider them out of scope, only focusing on those extensions that accidentally reveal themselves.

Our experiments show the different ways through which extensions reveal themselves by injecting content. From an unrandomized WAR URL injected in a page, as is the case for Chrome extensions, it is trivial to extract the UUID to determine the installed extension. As is shown in Table II, from a WAR URL where just the UUID has been randomized and probing is possible, as is the case for Firefox extensions, we can deduce the installed extension with a 80.33% accuracy by considering only the path of the URL, and the paths tied to each extension. Similarly, we would be able to deduce the installed extension with a 93.76% accuracy by only looking at the contents of the resources tied to the extensions, and combining the two approaches, we can deduce the installed extension with a 94.41% accuracy. Similarly, we detect Chrome extensions with a 62.01% accuracy based on the path, 89.01% accuracy based on the content of the resource, and 89.91% accuracy when we combine the path and the content.

Without breaking the intended functionality provided by existing extensions, we cannot prevent extension probing attacks and extension revelation attacks in general.

Our envisioned solution, which we call "Latex Gloves" since the goal is to prevent extensions from leaving fingerprints, is depicted in Figure 3.

We prevent extension probing attacks (Figure 3a) by allowing a whitelist to specify a set of web pages that may probe for each individual extension.

For instance, YouTube.com may be allowed to probe for the ChromeCast extension, so that the extension's functionality can be used with YouTube videos. In that case, a request for a WAR in the ChromeCast extension will be allowed by the policy. However, when the same WAR is requested by another web page, such as attacker.com, the request is blocked. Similarly, if YouTube.com would request a WAR from another extension, e.g. AdBlock, it would be blocked with this particular policy.

We prevent extension revelation attacks (Figure 3b) by allowing a whitelist to specify a set of web pages on which each extension is allowed to execute.

For instance, the AdBlock extension may be allowed to run on example.com. In that case, when example.com is visited, the AdBlock extension can remove any advertisements from the page. However, the same extension may be disallowed from running on a website which is trusted by the whitelist policy, thereby not interfering with the revenue stream of that website. Similar to the probing defense example, the policy here also blocks other extensions from executing — and thereby potentially revealing themselves — on example.com.

Conceptually, the policies for both defenses can be visualized in a matrix, with extensions and web origins as rows and columns respectively. Each element in this matrix would then indicate whether access is allowed between the extension and the web origin.

However, such a matrix would make the assumption that policies for the probing and revelation defenses cannot conflict, which is not necessarily the case.

For instance, consider a configuration where AdBlock is installed, and a banking website bank.com, which is trusted by the whitelist policy. Because this trust, bank.com should be allowed to probe for AdBlock. However, due to the sensitive nature of the data on bank.com, the whitelist policy does not allow AdBlock to operate on the bank.com web pages, although AdBlock want to execute on every web page.

This conflict between the policies for a particular web origin and extension illustrates the need for separate whitelisting mechanisms for both the probing and revelation defenses.

VI. Proof of concept implementation

Our prototype implements defenses against both the extension probing and extension revelation attacks as a proof of concept. Because changing browser code can quickly get very complicated, we opted to implement only the core functionality in the actual browser code, while the bulk of our prototype is implemented separately as a browser extension and a web proxy. For adoption in the real world, the full implementation should of course be embedded in the web browser's C++ code. However, our proof of concept implementation still allows to test the effectiveness of our solution. For simplicity, the proof of concept is designed to allow a security-aware end user to

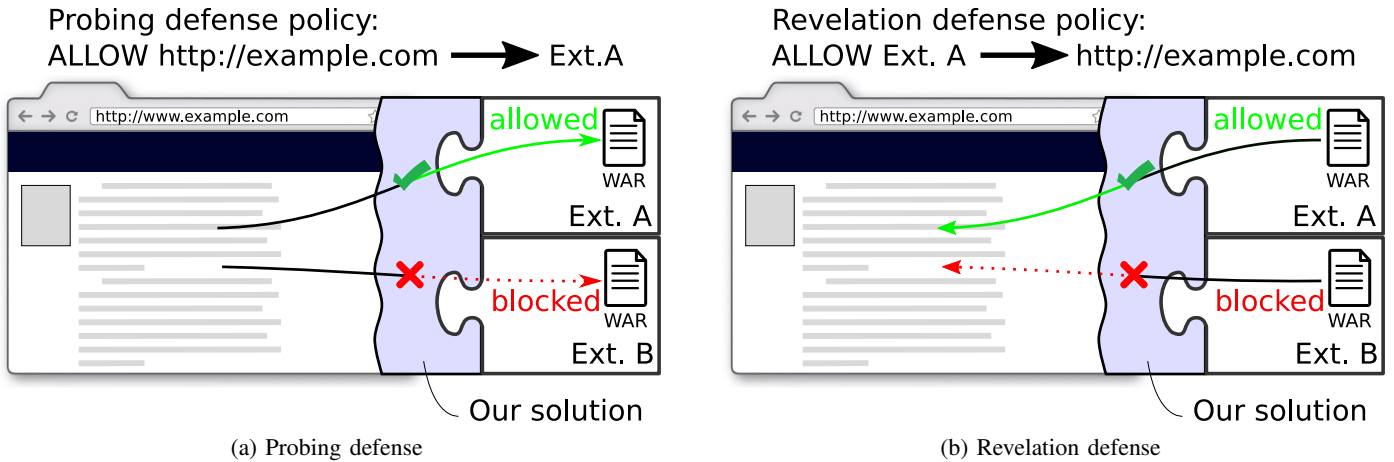(a) Probing defense   (b) Revelation defense

Fig. 3: Concept design of our proposed defenses for the extension probing and revelation attacks. Our solution mediates access from the web page to the extension WARs for the probing defense, and from the extensions to web pages for the revelation defense. In each case, access is mediated based on a specified policy.

arbitrarily modify the whitelists. While this is not something one should assume an arbitrary user would do, we deem it to be good in order to show the functionality of the whitelisting mechanisms. In a full implementation, the end user should be queried as little as possible.

As depicted in Figure 4, our prototype implementation consists of three components: a slightly modified Chromium browser, a browser extension named "Latex Gloves" and a web proxy based on mitmproxy. Our modifications to the Chromium 65.0.3325.181 code consist of nine lines of code spread over four files. The patches to Chromium, as well as binary packages compiled for Ubuntu 16.04, our browser extension and our addon script for mitmproxy 3.0.4 are available upon request to the authors.

A. Preventing the probing attack

Chrome extensions can use the webRequest API to observe, modify and block requests from web pages. The requests that an extension can observe through the webRequest API, include requests with the `chrome-extension://` scheme. However, requests to `chrome-extension://<ext-UUID>` URIs where `<ext-UUID>` is not its own extension ID, will be hidden. Even though requests to non-installed extension resources, or to `chrome-extension://` URIs with an invalid extension ID are hidden from observation with the webRequest API, those URIs are replaced by `chrome-extension:// invalid` internally.

Our prototype needs the ability to monitor requests to all `chrome-extension://` URIs, even for other installed extensions, non-installed extensions or invalid extension IDs. In addition, we also want to avoid that Chromium replaces the URI with `chrome-extension://invalid`, since we are interested in the originally requested URI.

To achieve this, we modified the Chromium source code and changed just two lines of code in two files. First, we disable the check that determines whether the extension ID of the requested URI matches that of the extension observing the request. Second, we disable Chromium's behavior of replacing invalid `chrome-extension://` URIs.

The remainder of this part of the prototype is implemented as a browser extension which uses this modified webRequest API. Requests to all `chrome-extension://` URIs are monitored by the extension and matched against a predefined but customizable whitelist. The whitelist maps a web origin O to a list of allowed extension IDs L. When the browser visits a web page located in the given web origin O, the extension checks any requested `chrome-extension://` URIs and determines whether they target an extension in L. In case of a match, the request is allowed, otherwise it is canceled. In the latter case, it will appear to the web page as if the requested resource is not accessible, whether the extension is installed or not.

B. Preventing the revelation attack

By design, Chrome extensions can specify which URLs they want to operate on, by listing those URLs in the `permissions` and `content_scripts` properties of the *manifest.json* file. Restricting the list of URLs on which an extension is allowed to operate, would help prevent the extension revelation attack on arbitrary attacker pages, since the extension would not execute on those pages, and thus not reveal itself. However, this whitelist of URLs is at the discretion of the extension developer and cannot easily be altered by the whitelist policy provider.

Our implementation, schematically depicted on the right side of Figure 4, exposes the whitelist on which URLs the extension operates to the whitelist policy provider, allowing the restriction of the set of URLs on which the extension operates. Instead of implementing new functionality in the browser to modify this whitelist, and then exposing it to our browser extension, we decided to modify the browser extension CRX [19] files, which are packaged and signed versions of browser extensions, "in flight" when they are installed or updated from the Chrome web store.

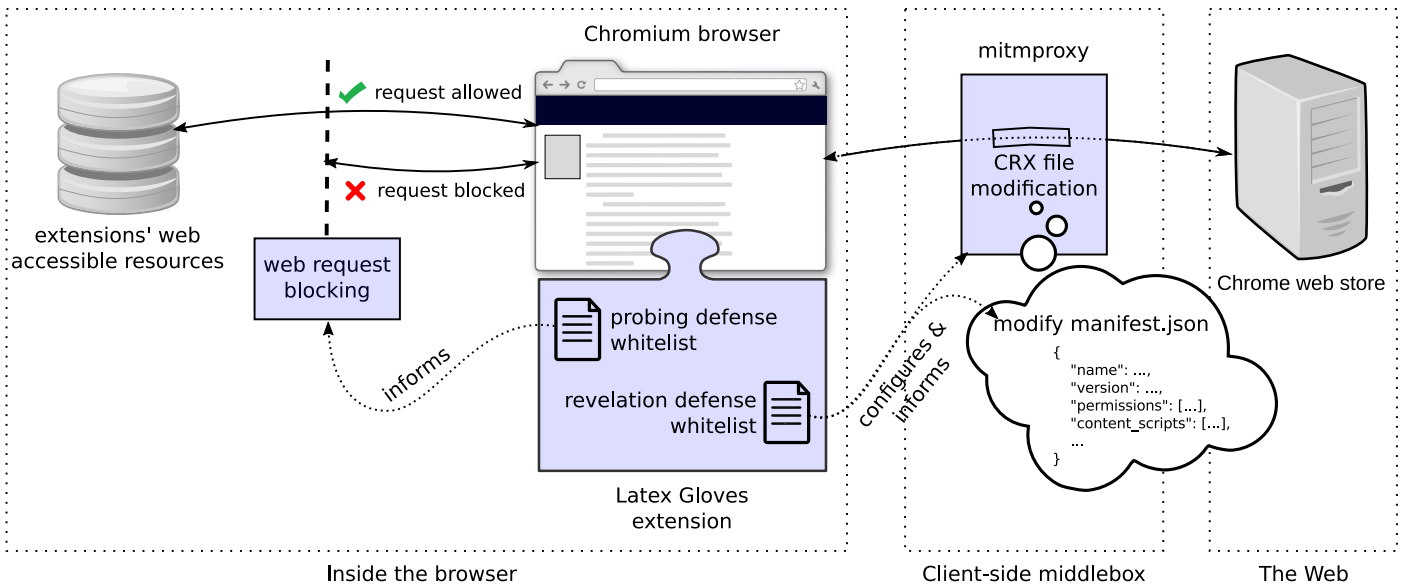Because extensions from the Chrome web store are signed

Fig. 4: Overview of the prototype implementation of our proposed defenses: a modified Chromium browser with the Latex Gloves extension and mitmproxy.

with a private key, which we cannot obtain, we modified the Chromium browser to not strictly verify an extension's signature. This modification consists of six lines of code in a single file, and disables signature verification on both version 2 and 3 of the CRX file format. It is important to note that, for a real-world implementation, this should not be done, but rather have the full mechanism implemented in the browser. We only use this to show and evaluate the core whitelisting mechanism in the proof of concept prototype.

Since the browser no longer verifies CRX signatures, we are free to modify web traffic between the browser and the Chrome web store, and can update the manifest files in extensions' CRX files "in flight" and restrict the `permissions` and `content_scripts` properties according to the wishes of the whitelist. This CRX rewriting process is implemented in a web proxy as a mitmproxy addon script.

When the policy changes the hostname whitelist associated with an extension, the new whitelist is communicated to the proxy. When the auto-update process in the browser queries the Chrome web store whether the extension has been updated, we inform the browser that a new version exists. The browser then downloads the new version of the extension from the Chrome web store, which gets rewritten by our mitmproxy addon script, and includes the new whitelist.

Taking over the extension auto-update process for our proof of concept prototype in this manner, requires us to make more frequent changes to the version number of an extension than the extension's developer would. Because of the way the versioning system works, we need to keep track of a parallel versioning scheme that is only visible between the browser and the proxy. The details of this process are too technical to detail in this paper, but require us to change the `version` property of the manifest file in addition to the `permissions` and `content_scripts` properties.

By default, the Chromium auto-update process can take up to seven days, which we deem too infrequent to be of practical use in our proof of concept. An optional modification of one line of code in one file of the Chromium source code changes this update interval to five seconds, so that updates to the policy whitelist are implemented more promptly.

In addition, it should be noted that the original extension update mechanism will prompt the end user whenever the extension requests additional permissions compared to the previous version. Our proof of concept implementation does not alter this default behavior.

C. Discussion and future work

Our prototype implementation is a proof of concept, showing that it is possible to use whitelisting policies to defend against extension probing and revelation attacks. As mentioned before, an actual production-quality implementation of these defenses would require more changes to browser code and result in better performance and a nicer user experience with regards to e.g. the user interface.

A real-world implementation in the browser would not need to rewrite the extensions on the fly, and would not have to disable security checks. Similar to how the browser checks if, e.g., a WAR should be allowed to be injected, the browser can check if the extension should be allowed to execute on any given domain.

Recently, Google released the plan to allow end users to restrict the host permissions for an extension [7], indicating the core mechanism for modifying browser extension behavior within the browser is possible, and something which can be used to control the extension whitelist. In this case, the browser extension can provide a whitelist which can be modified without the need to re-install the extension.

It is also crucial for a real-world implementation to not have an early-out mechanism, which is what was exploited

in the timing attack presented by Sánchez-Rola et al. [53], and subsequently removed [20]. In the situation an attacker is allowed to probe for an extension, and that extension is present, an early-out from the whitelisting mechanism during a probing attack would allow for the attacker to measure the elapsed time, and deduce whether the request was blocked based on the whitelist. If an attacker knows the time it takes to get a response from an installed extension which they are allowed to probe for, and an extension which is blocked by the whitelist, the attacker can, for each negative probing attempt, deduce which extensions that are not installed, and which that are blocked based on the whitelist.

For our prototype, we made the rather arbitrary choice to limit whitelists to web origins and hostnames in the probing and revelation defense respectively. While these choices serve us well for a proof of concept, it could prove interesting to refine these whitelists to use e.g. regular expressions on URLs instead.

Additionally, for the probing defense, when a web page contains an embedded subframe, we disregard the web origin of the subframe and enforce the whitelist associated with the web origin of the main frame. Our prototype is very well capable of applying a different whitelist for the subframe, in case the end user would wish to do so. However, we regarded this particular refinement of the prototype as out of scope for a proof of concept implementation.

In our proof of concept implementation, only the end-user can specify policy whitelists for both the probing and revelation defenses. In a production implementation, one should consider a system where both web applications and browser extensions can suggest a policy, which the end-user could then refine or even override. Another possibility is to have a system similar to Google Safe Browsing [28], keeping the user interaction to a minimum.

Finally, our prototype implementation displays information to the user about which extensions are being probed for on any visited web page. We do not display similar information regarding revelation attacks. We also consider these visual markers to be out of scope to prove the functionality of the concept.

VII. EVALUATION

We have evaluated the functionality of our proof of concept implementation to ensure that it works as intended. Using the data from Sections III and IV, we randomly selected and visited several dozen web pages that perform probing attacks, and also visited our attacker web page with the top ten (Chrome) extensions that reveal themselves on any web page with any content. As expected, our proof of concept implementation stops both the probing attacks and revelation attacks.

We also perform two evaluations against known old attacks, the enumerating probing attack presented by Sjösten et al. [55] (Section VII-A) and the enumerating timing probing attack presented by Sánchez-Rola et al. [53] (Section VII-B).

A. Enumerating probing attack

We visited two known web pages that employed the enumerating probing attack [54], [32] twice: the first time with an

TABLE VII: Enumeration timing probing attack.

	Chromium 53.0.2785.135	Patch	Patch + Extension
`<realExtUUID>/<realPath>`	8.53ms	9.67ms	8.95ms
`<realExtUUID>/<fakePath>`	12.59ms	9.71ms	9.17ms
`<fakeExtUUID>/<fakePath>`	7.86ms	10.16ms	9.3ms

unmodified Chromium browser, and the second time with the modified Chromium browser and with our browser extension installed. We used browser extensions which we know can be detected both times: AdBlock [10], Avast Online Security [4], Ghostery [6] and LastPass [39]. When visiting with the modified Chromium browser with our browser extension, we set the policy to a "block all" policy, meaning we expect no WARs to be accessible to the web page.

As expected, with our unmodified Chromium browser, the probing attack was successful against all four extensions. Note that although the database was last updated in December 2016 for [54], it could still detect the popular extensions, which might indicate browser extensions do not change internally very often. Using our proof of concept implementation, the probing attacks failed for all extensions. Although the execution time increased significantly, due to the handling of over 11,000 requests for our JavaScript code in the browser extension, we note that this is something that will improve if the mechanism is fully implemented in the source language of the browser. We also set policies to allow for the probing of each extension, one at a time, indicating that the overall idea explained in Section V is sound.

B. Enumerating timing probing attack

To be consistent with prior work, we determined whether our modification of Chromium's core might reintroduce the enumerating timing probing attack — already fixed from versions higher than 61.0.3155.0 — presented by Sánchez-Rola et al. [53]. This timing attack makes a distinction between two types of requests: 1) `chrome-extension://<fakeExtUUID>/<fakePath>`, and; 2) `chrome-extension://<realExt-UUID>/<fakePath>`. The attacker uses the User Timing API [59], which allows to take time measurements with high precision, to check the response times for each of these requests. If the measured times do not differ more than 5%, the attacker can conclude that the requested extension is not installed in the client's browser.

In order to reproduce this timing attack, we downloaded and built Chromium 53.0.2785.135 on a virtual machine with Ubuntu 16.04.

We identified three scenarios: 1) using the original Chromium 53.0.2785.135 source code; 2) Chromium 66.0.3359.117 with our patch applied, but without the Latex Gloves extension, and; 3) Chromium 66.0.3359.117 with our patch applied and the Latex Gloves extension installed.

For each scenario, we had Avast Online Security installed and used it as the `<realExt-UUID>`. When executing with our patch and Latex Gloves installed, we had set the whitelist to allow all requests to extension WARs, apart from to Avast Online Security and AdBlock. Table VII shows the results

TABLE VIII: Breakdown of the amount of Chrome and Firefox extensions that would be uniquely identifiable through the content of a WAR, given that no probing could take place.

	Extensions	Total WARs	Unique WARs	Detection probability
Firefox	1,378	95,920	23,687	24.69%
Chromium	11,633	12,499,335	127,054	1.02%
Revealing	2,906	4,027,046	35,478	0.88%

of our experiment, where the time measurement for each request was averaged over 1,000 runs. From these results, it is clear that Chromium 53.0.2785.135 is vulnerable to the timing attack, since there is more than 5% difference between the time measurement for an existing extension and a non-existing extension. However, with our modification (with or without extension), that difference is no longer present.

VIII. Recommendations

Based on the experiments in Sections III and IV, we recommend several improvements to the browser extension ecosystem, addressed to browser developers and extension developers.

Recommendations for browser developers: Chrome extensions are vulnerable to the extension probing attack because their UUIDs are static and publicly known. Firefox extensions combat this vulnerability by having randomized extension UUIDs. However, Firefox extensions can still be identified through the revelation attack. Worse, because Firefox's random UUIDs are not easily changed after an extension is installed, they can be used to fingerprint the extension user.

Our first recommendation is to re-generate Firefox's random UUIDs more often, either upon starting the browser or for each domain visited. Similarly, if a user enables private browsing mode [48], [23], each active browser extension should be provided with a new random UUID. Although this would not prevent detecting which browser extensions are executed, it would limit the tracking to a specific instance, making it infeasible to use this technique for long-term tracking of users.

Our second recommendation is to randomize the full URL of a WAR, and not just the UUID. With this change, a WAR URL seen by an attacker would be shaped as `moz-extension://<random-UUID>/<random-path>` for Firefox and `chrome-extension://<random-UUID>/<random-path>` for Chrome. Without any recognizable path components, the attacker would be forced to read and fingerprint the contents of the WAR to determine which extension is installed. As depicted in Table VIII, without the ability to probe, this would decrease the probability of detecting Firefox extensions to 24.69% (compared to 93.76%, as shown in Table II), and 1.02% for Chrome (compared to 89.01%) and probability of detecting the extensions we know reveal themselves would drop to 0.88% from 89.52%. The random path approach can be taken one step further by implementing the WAR URLs to be of single use, i.e. the same WAR will have different paths each time it is injected or fetched. Such a change to core extension infrastructure would make it impossible for an attacker to fetch a recently injected resource in order to analyze the content. However, it would also require an overhaul of the browser implementation and possibly most browser extensions, which is very impractical.

Recommendations for browser extension developers: Both Mozilla [43] and Google [27] provide guidelines for browser extension developers, e.g. "never ask for more permission than needed", and "properly secure sensitive or personal data when transmitting over the network". However, neither provide specific guidelines on how to handle WARs in a secure way.

Our only recommendations fall in the "least privilege" category, where no more privileges than needed to perform a certain task should be requested. Firstly, to help prevent the revelation attack, extension developers should not arbitrarily inject content with the random UUID. As seen in Table V, several extensions currently inject content on any arbitrary web page, including blank pages. Secondly, to help prevent the probing attack, extensions should *not expose unused WARs*. A non-existent WAR cannot be used in a probing attack, thus reducing the chances that an extension can be identified through a probing attack.

IX. Related work

User fingerprinting by using web browsers has been widely studied in the literature [12], [9], [11], [38], [15], [34]. As an example, Cao et al. [15] were able to fingerprint 99.24% of web users — being completely web browser agnostic — by using hardware features such as those from GPUs or CPUs. More recently, Gómez-Boix et al. [34] performed a large scale experiment to determine whether fingerprinting is still possible nowadays. They reached the conclusion that in desktop web browsers, both plugins (e.g. Flash, NPAPI, etc) and fonts are the most representative features to fingerprint users. However, none of the aforementioned works have taken browser extensions into consideration.

Nikiforakis et al. [52] showed that implementation differences between browsers can be fingerprinted. There exist several extensions that attempt to erase those fingerprints, but those extensions in turn allow a user to also be fingerprinted. In the same vein, Acar et al. [9] state that browser extensions can be exploited to fingerprint and track users on the Web.

Starov and Nikiforakis [56] presented a method to fingerprint browser extensions using a behavioral attack. They show browser extensions can provide unique, arbitrary DOM modifications, and analyzes the top 10,000 of most downloaded browser extensions, concluding 9.2% to 23% of those extensions are detectable. Contrarily to the experiments they performed — they only analyzed the manifest file of 1,665 browser extensions and they found that more than a 40% of them do make use of WARs, in this work we have scrutinized 62,994 browser extensions and concluded that 16,280 explicitly declare some WARs in their *manifest.json* file ($\approx$26%).

In 2011, Kettle [36] demonstrated that all Chrome extensions could be enumerated by requesting their manifest file, which was explained in 2012 by Kotowicz [37]. Google solved this problem by introducing WARs, but Sjösten et al. [55] showed that all Chrome extensions with WARs can be enumerated without them being active on the attacker page. They demonstrated that approximately 28% of all Chrome extensions and approximately 6.7% of all non-WebExtension Firefox extensions could be enumerated from a web page. Gulyás et al. [33] combine known fingerprinting techniques

with the Chrome extension enumeration attack presented by Sjösten et al. [55], along with a login-leak which determines the web pages that a user is logged in to [40]. They conclude that 54.86% of users which have installed at least one detectable extension and 19.53% of users which have at least one detectable active login, are unique. A combination of at least one detectable extension installed, and at least one detectable active login make the uniqueness number go up to 89.23%, indicating that installed browser extensions can make a good fingerprint, further showing the necessity of a mechanism to prevent extension fingerprinting.

Sánchez-Rola et al. [53] presented a timing attack against Chrome and Firefox by using the fact that the internal branching time for WARs differs between installed and non-installed extensions, thus detecting 100% of all extensions. A temporary solution has been implemented in Chrome [20], and the plan is to implement a randomization scheme similar to Firefox's, when they can make "a breaking change" [8]. In [53], Sánchez-Rola et al. also presented the revelation attack against Safari, which was the first browser to use randomized UUIDs. Based on a static analysis of 718 extensions, they estimated more than 40% of the extensions could leak the random UUID. They manually analyzed 68 security extensions, finding one false negative and 20 out of 29 extensions flagged as suspicious indeed leaked the random UUID. Contrarily to Sánchez-Rola et al, we investigate all Chrome and Firefox extensions to see which leak their UUID on actual web pages.

Chen and Kapravelos [17] developed a taint analysis framework for browser extensions to study their privacy practices. From sources, such as DOM API calls (e.g. `document.location`), and extension API calls (e.g. `chrome.history`), they find 2.13% of Chrome and Opera extensions to potentially be leaking privacy-sensitive information to sinks such as `XMLHttpRequest` and `chrome.storage`. However, they do not seem to consider extension UUIDs as part of the privacy-sensitive information.

Finally, it is worth mentioning that an attacker might use any of the attacks presented in this paper to detect browser extensions and thus, perform more harmful attacks. Buyukkayhan et al. [14] for instance, exploit the lack of non-isolation worlds on the previous version of the Firefox add-ons architecture, allowing legitimate extensions which make use of *Cross Platform Component Object Model (XPCOM)* to access system resources such as the file system and the network. A prerequisite for this attack is that there must be a mechanism to disclose installed extensions in the victim's browser. Thus, the attacks described in our work may be used as a stepping stone to escalate the attacker's privileges in the browser.

X. CONCLUSION

We have investigated the problem of detecting browser extensions by web pages. With the intention to prevent probing for browser extensions by web pages, Mozilla Firefox recently introduced randomized extension UUIDs. A similar move is currently being discussed by the Google Chrome developers. We have demonstrated that the randomized UUIDs can in fact hurt user privacy rather than protect it. To this end, we have studied a class of attacks, which we call revelation attacks, allowing web pages to detect the randomized browser extension UUIDs in the code injected by extensions into the web pages, which, due to the design of the randomization of UUIDs, giving the ability to uniquely track users.

We have conducted an empirical study assessing the feasibility of revelation attacks. Our experiments show that combining revelation and probing attacks, it is possible to uniquely identify 90% out of all extensions injecting content, in spite of a randomization scheme. Furthermore, we have conducted a large-scale study assessing the pervasiveness of probing attacks on the Alexa top 10,000 domains, providing new evidence for probing beyond what was captured by previous work.

As a countermeasure, we have designed a mechanism that controls what extensions are enabled on what pages. As such, our mechanism supports two types of whitelists: specifying which web pages are allowed to probe for which extensions and specifying which extensions are allowed to inject content on which web pages. We have presented a proof of concept prototype that blocks both probing and revelation attacks, unless explicitly allowed in the whitelists.

For future work, it would be interesting to consider XHOUND [56] and Hulk [35] to make a comparison on the different extensions that provide arbitrary DOM modifications (XHOUND), extensions that are deemed malicious (Hulk), and that inject WAR URLs. Unfortunately, the tools are unavailable at present.

Next steps for Firefox and Chrome: We have reported the details of our study and our suggestions for mitigation to both involved browser vendors.

The issue with the randomized UUIDs has been confirmed by Firefox developers [1]. They agree that attacks like the revelation attack defeat anti-fingerprinting measures. While the problem is clear to the developers, the discussion on countermeasures is still ongoing.

As mentioned earlier, Google has recently announced that Chrome will allow users to restrict extensions from accessing websites by a whitelisting mechanism in line with ours [7]. Users will be able to restrict the host permissions for an extension, paving the way for an in-browser mechanism to control the extension whitelist.

REFERENCES

[1] https://bugzilla.mozilla.org/show_bug.cgi?format=default&id=1372288, accessed July-2018.

[2] "AdBlock Plus," https://chrome.google.com/webstore/detail/adblock-plus/cfhdojbkjhnklbpkdaibdccddilifddb, accessed Aug-2018.

[3] "Adobe: Adobe Acrobat Force-Installed Vulnerable Chrome Extension," https://bugs.chromium.org/p/project-zero/issues/detail?id=1088, accessed May-2018.

[4] "Avast Online Security," https://chrome.google.com/webstore/detail/avast-online-security/gomekmidlodglbbmalcneegieacbdmki, accessed May-2018.

[5] "Common Crawl," http://commoncrawl.org/, accessed May-2018.

[6] "Ghostery – Privacy Ad Blocker," https://chrome.google.com/webstore/detail/ghostery---privacy-ad-blo/mlomiejdfkolichcflejclcbmpeaniij, accessed Aug-2018.

[7] "Trustworthy Chrome Extensions, by Default," https://security.googleblog.com/2018/10/trustworthy-chrome-extensions-by-default.html, accessed Nov-2018.

[8] "WebAccessibleResources take too long to make a decision about loading if the extension is installed," https://bugs.chromium.org/p/chromium/issues/detail?id=611420#c19, accessed Feb-2018.

[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, pp. 1129–1140.

[10] "AdBlock," https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom, accessed Aug-2018.

[11] P. Baumann, S. Katzenbeisser, M. Stopczynski, and E. Tews, "Disguised Chromium Browser: Robust Browser, Flash and Canvas Fingerprinting Protection," in *WPES*, 2016, pp. 37–46.

[12] K. Boda, A. M. Földes, G. G. Gulyás, and S. Imre, "User Tracking on the Web via Cross-browser Fingerprinting," in *NordSec*, 2012, pp. 31–46.

[13] M. Brinkmann, "Firefox WebExtensions may be used to identify you on the Internet," https://www.ghacks.net/2017/08/30/firefox-webextensions-may-identify-you-on-the-internet/, 2017.

[14] A. S. Buyukkayhan, K. Onarlioglu, W. K. Robertson, and E. Kirda, "CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities," in *NDSS*, 2016.

[15] Y. Cao, S. Li, and E. Wijmans, "(Cross-)Browser Fingerprinting via OS and Hardware Level Features," in *NDSS*, 2017.

[16] S. Cassidy, "LostPass," https://www.seancassidy.me/lostpass.html, 2018.

[17] Q. Chen and A. Kapravelos, "Mystique: Uncovering Information Leakage from Browser Extensions," in *CCS 2018*, 2018, pp. 1687–1700.

[18] Chrome, "Match Patterns," https://developer.chrome.com/extensions/match_patterns, accessed Apr-2018.

[19] ——, "Webstore Hosting and Updating," https://developer.chrome.com/extensions/hosting, accessed Apr-2018.

[20] Chromium Code Reviews, "Issue 2958343002: [Extensions] Change renderer-side web accessible resource determination (Closed)," accessed Feb-2018. [Online]. Available: https://codereview.chromium.org/2958343002

[21] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A free and open source interactive HTTPS proxy," https://mitmproxy.org/, 2010–, [Version 3.0], accessed May-2018.

[22] U. Fiore, A. Castiglione, A. De Santis, and F. Palmieri, "Countering Browser Fingerprinting Techniques: Constructing a Fake Profile with Google Chrome," in *NBiS*, 2014, pp. 355–360.

[23] Google, "Browse in private," https://support.google.com/chrome/answer/95464, accessed May-2018.

[24] ——, "Chrome Web Store," https://chrome.google.com/webstore/category/extensions?_feature=free, accessed Feb-2018.

[25] ——, "chrome.runtime," https://developer.chrome.com/extensions/runtime#method-getURL, accessed Feb-2018.

[26] ——, "Content Scripts," https://developer.chrome.com/extensions/content_scripts, accessed Feb-2018.

[27] ——, "Developer Program Policies," https://developer.chrome.com/webstore/program_policies, accessed May-2018.

[28] ——, "Google Safe Browsing," https://safebrowsing.google.com/, accessed July-2018.

[29] ——, "Manifest - Web Accessible Resources," https://developer.chrome.com/extensions/manifest/web_accessible_resources, accessed Apr-2018.

[30] ——, "Manifest File Format," https://developer.chrome.com/extensions/manifest, accessed Feb-2018.

[31] ——, "New Cast functionality in Chrome," https://support.google.com/chromecast/answer/6398952, accessed Apr-2018.

[32] G. G. Gulyás, D. F. Somé, N. Bielova, and C. Castelluccia, "Browser Extension and Login-Leak Experiment," https://extensions.inrialpes.fr/, accessed Apr-2018.

[33] ——, "To Extend or not to Extend: On the Uniqueness of Browser Extensions and Web Logins," in *WPES@CCS*, 2018, pp. 14–27.

[34] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale," in *WWW*, 2018.

[35] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting Malicious Behavior in Browser Extensions," in *USENIX Sec.*, 2014, pp. 641–654.

[36] J. Kettle, "Sparse Bruteforce Addon Detection," http://www.skeletonscribe.net/2011/07/sparse-bruteforce-addon-scanner.html, 2011.

[37] K. Kotowicz, "Intro to Chrome addons hacking: fingerprinting," http://blog.kotowicz.net/2012/02/intro-to-chrome-addons-hacking.html, 2012.

[38] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints," in *S&P*, 2016, pp. 878–894.

[39] LastPass, "LastPass: Free Password Manager," https://chrome.google.com/webstore/detail/lastpass-free-password-ma/hdokiejnpimakedhajhdlcegeplioahd, accessed May-2018.

[40] R. Linus, "Your Social Media Fingerprint," https://robinlinus.github.io/socialmedia-leak/, 2016.

[41] L. Liu, X. Zhang, V. Inc, G. Yan, and S. Chen, "Chrome extensions: Threat analysis and countermeasures," in *NDSS*, 2012.

[42] Microsoft, "Internet Explorer Browser Extensions," https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa753587(v%3dvs.85), 2018.

[43] Mozilla, "Add-on Policies," https://developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews, accessed May-2018.

[44] ——, "content_scripts," https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/content_scripts, accessed Feb-2018.

[45] ——, "extension.geturl()," https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/extension/getURL, accessed Feb-2018.

[46] ——, "manifest.json," https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json, accessed Feb-2018.

[47] ——, "Most Popular Extensions," https://addons.mozilla.org/en-US/firefox/search/?sort=updated&type=extension, accessed Feb-2018.

[48] ——, "Private Browsing - Use Firefox without saving history," https://support.mozilla.org/en-US/kb/private-browsing-use-firefox-without-history, accessed May-2018.

[49] ——, "Profiles - Where Firefox stores your bookmarks, passwords and other user data," https://support.mozilla.org/en-US/kb/profiles-where-firefox-stores-user-data/, accessed Mar-2018.

[50] ——, "web_accessible_resoruces," https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/web_accessible_resources, accessed Feb-2018.

[51] Mozilla Add-ons Blog, "WebExtensions in Firefox 57," https://blog.mozilla.org/addons/2017/09/28/webextensions-in-firefox-57/, accessed Feb-2018.

[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, pp. 541–555.

[53] I. Sánchez-Rola, I. Santos, and D. Balzarotti, "Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies," in *USENIX Security Symposium*, 2017, pp. 679–694.

[54] A. Sjösten, S. Van Acker, and A. Sabelfeld, "Non-behavioral extension detector," http://blueberry-cobbler-11673.herokuapp.com, accessed May-2018.

[55] ——, "Discovering Browser Extensions via Web Accessible Resources," in *CODASPY*. ACM, 2017, pp. 329–336.

[56] O. Starov and N. Nikiforakis, "XHOUND: Quantifying the Fingerprint-ability of Browser Extensions," in *S&P*, May 2017, pp. 941–956.

[57] StatCounter, "Desktop Browser Market Share Worldwide," http://gs.statcounter.com/browser-market-share/desktop/worldwide, accessed May-2018.

[58] W3C, "CSP2," https://www.w3.org/TR/CSP2/, accessed Nov-2018.

[59] ——, "User Timing," https://www.w3.org/TR/user-timing, accessed May-2018.