Hardening the Security Analysis of Browser Extensions

Benjamin Eriksson
Chalmers University of Technology
beneri@chalmers.se

Pablo Picazo-Sanchez
Chalmers University of Technology
pablop@chalmers.se

Andrei Sabelfeld
Chalmers University of Technology
andrei@chalmers.se

Abstract

Browser extensions boost the browsing experience by a range of features from automatic translation and grammar correction to password management, ad blocking, and remote desktops. Yet the power of extensions poses significant privacy and security challenges because extensions can be malicious and/or vulnerable. We observe that there are gaps in the previous work on analyzing the security of browser extensions and present a systematic study of attack entry points in the browser extension ecosystem. Our study reveals novel password stealing, traffic stealing, and inter-extension attacks. Based on a combination of static and dynamic analysis we show how to discover extension attacks, both known and novel ones, and study their prevalence in the wild. We show that 1,349 extensions are vulnerable to inter-extension attacks leading to XSS. Our empirical study uncovers a remarkable cluster of "New Tab" extensions where 4,410 extensions perform traffic stealing attacks. We suggest several avenues for the countermeasures against the uncovered attacks, ranging from refining the permission model to mitigating the attacks by declarations in manifest files.

CCS Concepts

• **Security and privacy** → **Browser security**; **Web application security**;

Keywords

Web Security; Browser Extensions

ACM Reference Format:
Benjamin Eriksson, Pablo Picazo-Sanchez, and Andrei Sabelfeld. 2022. Hardening the Security Analysis of Browser Extensions. In *Proceedings of ACM SAC Conference (SAC'22)*. ACM, Brno, Czech Republic, April 25 - April 29, Article 4, 10 pages. https://doi.org/10.1145/3477314.3507098

1 Introduction

Modern web browsers allow users to customize and improve their browsing experience by installing browser extensions. The functionalities of these extensions can range from modifying the aesthetics of websites to blocking advertisements, adding accessibility features, or security and privacy features. Using these functionalities *malicious extensions* routinely steal information from unknowing users [16, 20] and thrive on fake content injection like fake ads [21].

Similar to mobile apps, the extensions are mainly installed from app stores, such as the Chrome Web Store. Google is continuously removing malicious extensions from the Web Store [21]. Yet, new malicious extensions continue emerging [16, 32]. Although extensions submitted to the Chrome Web Store are subject to analysis and vetting, the problem with automatically analyzing extensions is that detecting different threats requires different methods.

Threat Model. The power of extensions poses privacy and security challenges. Extensions can both read sensitive information directly [14, 18] and indirectly by redirecting network traffic. For example, a malicious extension can use JavaScript to read passwords from the DOM or listen to network traffic. There are further underexplored classes of attacks where malicious extensions can also attack other extensions, for example, to steal their internal data, like todo-notes or stored passwords.

The challenge is not only to find malicious extensions but also *vulnerable extensions*. For example, an attacker could trick an extension with access to the user's cookies to send the cookies to the attacker. Previous work has uncovered several classes of attacks and vulnerabilities related to browser extensions (discussed in detail in Section 9). For example, Kapravelos et al. [22] find malicious extensions trying to steal data or modify security headers. Somé [37] discovers code execution vulnerabilities in extensions through static and manual analysis. In this scenario, a malicious website attacks a vulnerable extension. Attacking the implementation in the browser is also possible. For example, Buyukkayhan et al. [7] exploit the lack of isolation mechanisms that Firefox used to implement in its browser extension ecosystem.

Yet there are gaps in the previous work when it comes to analyzing the security of the entire browser extension ecosystem. Our paper is a step towards filling the gaps in the security analysis of extensions. We accomplish this by performing a systematic study of the extension ecosystem, including extensions' assets, attackers, and possible interaction methods. The benefit of our approach is the wider threat model.

Approach. We propose a systematic approach to hardening the security analysis of browser extensions. The main thrust of our systematization is a systematic study of attack entry points in the browser extension ecosystem. This leads us to both discovering novel attacks and analyzing known ones from a wider attacker model perspective. We group all the attacks by the actors involved to define the attacker, victim, attack surface, and target asset. Based on the attack we use a combination of static and dynamic analysis to detect insecure extensions and in some cases synthesize payloads. We download all the 133,365 extensions from the Chrome Web Store and test our detection mechanism on the extensions. We search for characteristics of our novel attacks on the web and confirm their novelty by finding no evidence of attackers using them in the wild so far.

Table 1: Summary of the attacks versus the ecosystem presented in this paper.

Attack	Subattack	Attacker	Victim	In wild	Section
Password	Chrome autofill	Extension	Extension/User/Web page	Novel	Section 5.1.1
	Virtual keyboard	Extension	Extension/User/Web page	Novel	Section 5.1.2
Traffic		Extension	User	4,410	Section 5.2.1
Inter-extension	Collusion	Extension	User	Benign	Section 5.2.2
	History poisoning	Extension/Web page	Extension	1,349	Section 5.3.1
	Code execution	Extension/Web page	Extension	1,349	Section 5.3.2
	Fingerprint	Extension	User	10,785	Section 5.3.3

Attacks. We present novel vectors that extensions can use to attack both the user and other extensions. We divide them into three categories: password stealing, traffic stealing, and inter-extension attacks. We summarize these attacks in Table 1.

Password stealing: We develop a new method for actively stealing passwords, circumventing Chrome's protection. Chrome tries to protect against password stealing by not adding the password to the page DOM before a user interacts with the page. To circumvent this, an extension can change the type of the password field to text and capture a screenshot.

Traffic stealing: By analyzing extensions from the Web Store, we find extensions that are actively stealing search queries by redirecting traffic.

Inter-extension attacks: We create novel methods for detecting potentially vulnerable extensions that can be attacked by other extensions. We detect this by analyzing how inter-extension message passing and poisoning shared resources can lead to Cross-Site Scripting (XSS). In addition, we show that inter-extension message passing can also fingerprint extensions.

Empirical study. Our findings show that 4,410 extensions, totaling over 120,000 downloads, are actively stealing search queries from users. We also detect 1,349 extensions being vulnerable to takeover using XSS from malicious extensions. Furthermore, this group of extensions is also vulnerable to history poisoning attacks leading to XSS while 2,829 are vulnerable to HTML code injection via history poisoning. In the latter case, Content Security Policy (CSP) protects them from XSS. Our new method for fingerprinting extensions also improves the state-of-the-art by adding 162 new extensions.

Countermeasures. We propose countermeasures for each class of the aforementioned attacks. In brief, for the password attacks, we consider that extensions that want to take screenshots should declare a concrete permission (e.g., `captureVisibleTab`) similar to the need for the `desktopCapture` permission. For traffic stealing, Chrome is already in the process of adopting a new version of the manifest file where extensions will have to declare in advance how they will handle users' requests [15]. Finally, for the inter-extension attacks, we propose to either make mandatory the definition of the `externally_connectable` key in the manifest file or if not, change the security-by-default option, and if such a key is not in the manifest, then the extension will not handle any external message.

Contributions. The paper offers the following contributions:

- We present a systematic study of attack entry points in the browser extension ecosystem in Section 3.
- We describe our methodology based on a combination of static and dynamic analysis to discover attacks in Section 4.
- We study the prevalence of attacks in the wild and two novel attacks: password stealers and inter-extension history poisoning in Section 5.
- We perform a detailed case study of the popular "New Tab" extensions in Section 6.
- We present countermeasures to the identified problems in Section 7.

We release our implementation and example extensions[1].

2 Background

While implementation details differ between the main browsers (e.g., Firefox and Chromium-based browsers), the overall architecture for browser extensions is similar.

Chrome isolates the execution of browser extensions in different environments for security reasons [4]. Due to this, *message passing* is used. We can classify message passing, depending on who the sender is, into 1) scripts provided by the web page; 2) content scripts of the extensions, and; 3) background pages.

If the sender is a web page, it can use the `postMessage` method to send a message to any other script also running in the web page context. Also, web pages can send direct messages to background pages of extensions by using one-time requests (`sendMessage(<ext_id>)` from `runtime` and `tabs` APIs). Browser extensions will handle these messages by implementing `onMessageExternal`.

These event listeners are triggered if and only if the extensions define the `externally_connectable` key in the manifest file and explicitly add the web page in the `matches` sub-key. Since extensions cannot establish this external communication by themselves, the only way they have to reply to external messages is in the body of the event listener by using `sendResponse` or `postMessage`, depending on whether the communication is one-time or long-lived respectively. Using the same procedure, Chrome allows cross-extension messaging.

If the sender is the content script of an extension, it can send messages to scripts provided by web pages (or to other content scripts) by using `runtime.postMessage` method. It can also send direct messages to the background of the same extension using

[1] https://www.cse.chalmers.se/research/group/security/hardening-extensions/

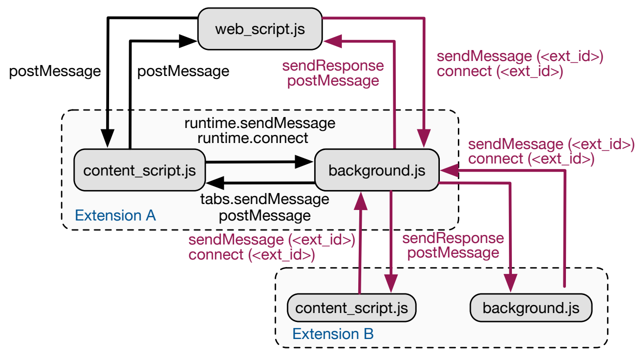

Figure 1: Message passing in browser extensions.

one-time requests. The background of the extension will handle messages coming from the content scripts of the same extension by implementing `onMessage()`. Also, content scripts can send external messages to other extensions using the method explained before, i.e., as if they were scripts coming from web pages.

If the sender is the background, it can use `postMessage()` to send messages to the content scripts of the same extension and `runtime.sendMessage(<ext_id>)` for cross-extension messaging.

In Figure 1, we include a summary of the message passing in browser extensions. In the figure, we represent two extensions (Extension A and B) and simplify the methods to send and receive information between isolated worlds (content scripts, background pages, and web scripts) as explained before.

3 Threat Model

We define a **vulnerable extension** as an extension that can lose control of any of its confidential or integrity-sensitive assets to another actor. For example, another malicious extension can send a message resulting in code execution. On the contrary, we say that an extension is **malicious** when it controls assets from another actor. For example, a malicious extension could steal the user's password, which is an asset. We define both the actors and the assets in the description of the entire ecosystem in Section 4.1.

There are two main **entry points**, which we explore in the following sections, that attackers can use to exploit browser extensions: 1) shared resources, and; 2) message passing.

3.1 Shared Resources

We consider shared resources to be all the elements that extensions have access to, e.g., DOM content, history, Web Accessible Resources (WARs), and bookmarks. The DOM can be used by scripts and extensions to communicate and share data [13].

3.2 Message Passing

As explained in Section 2, in message passing, we can distinguish between messages coming from scripts of the same extension and external ones. For messages coming from external scripts, we propose a novel and methodological way to extract from the manifest of the extensions how vulnerable they are.

External message passing. To know to what extent the extensions are vulnerable due to message passing, we use the optional

Table 2: Each row shows which communications are possible based on the IDs and matches defined in the manifest.

#	∃	externally_connectable IDs	externally_connectable matches	Communication Web pages	Communication Extensions
1	✗	–	–	✗	∀
2	✓	[ID_1,. . .,ID_n]	[URL_1,. . .,URL_n]	[URL_1,. . .,URL_n]	[ID_1,. . .,ID_n]
3	✓	[ID_1,. . .,ID_n]	✗	✗	[ID_1,. . .,ID_n]
4	✓	✗	[URL_1,. . .,URL_n]	[URL_1,. . .,URL_n]	✗
5	✓	✗	✗	✗	✗
6	✓	["*"]	[URL_1,. . .,URL_n]	[URL_1,. . .,URL_n]	∀
7	✓	["*"]	✗	✗	∀

`externally_connectable` key. Such a key can include two optional keys, `ids`, and `matches`. The former indicates the list of extension IDs whose messages are handled, whereas the latter for web pages instead.

If the `externally_connectable` key is not defined in the manifest (1st row in Table 2), all extensions can send messages but webpages cannot. On the contrary, if the key is defined there can be 6 possible options depending on the values of the two lists. We include in Table 2 all the possible cases of (not) defining the `externally_connectable` key in the manifest. Note that the most secure option is when the key is defined but none of the lists are provided (5th row in Table 2). By analyzing the manifest file of all the extensions, we found that 6,417 extensions define `externally_connectable` key whereas 126,948 do not, meaning there are more than 100,000 extensions that accept external messages coming from other extensions.

Content scripts message passing. Even though message passing using content scripts is an attack vector, as recently demonstrated [37], we consider that this is a particular case of the DOM shared resource entry point. Extensions can react to the event fired by either web pages or by other extensions when they send a message.

4 Methodology

In this section, we explain in detail our method for discovering malicious as well as potentially vulnerable extensions.

4.1 Identifying entry points

To harden the analysis we first perform a systematic analysis of the extension ecosystem, including assets, attackers, and interaction methods. We use a top-down approach and start by analyzing the actors, followed by the different assets they possess, the attack surface, and finally the entry points into the extension.

Ecosystem. The browser extension ecosystem allows for rich interaction between three classes of actors: *users*, *web pages*, and *extensions*.

Security Assets. We define security assets as important assets related to sensitive information and program control flow. The main asset of the user in our model is sensitive information, such as history, cookies, passwords, and the user's online activity. Passwords are also custody of web pages, making them an important asset to web pages too. Finally, extensions' assets include both confidential user-generated data, like todo-notes and passwords, and control flow (e.g., code execution). For example, an extension stealing sensitive data like users' history is malicious as it is reading a confidential asset (history).

Attack surface. We model the attack surface by close inspection of the interaction methods available to the actors. For shared resources, we consider DOM content, history, bookmarks, WARs, and cookies. These could potentially be used to carry malicious payloads. In Figure 1 we show the possible message passing interactions between extensions and web pages that make up the second part of the attack surface for the two actors. Note that both web pages and extensions can use message passing but in the inter-extension case the messages can go directly to the high-privileged background scripts making such vulnerabilities a larger threat compared to attacks from web pages. We also further refine the attack surface dynamically by inspecting the manifest of the extension. By comparing the manifests of extensions to Table 2 we can efficiently filter out attack vectors.

Entry points. The final step is to extract all the entry points from the attack surface and analyze them. We use a combination of static and dynamic analysis based on the type of entry points to do this.

4.2 Combining Static and Dynamic Analysis

We crawled the Web Store in Jan 2020 and downloaded all the 133,365 browser extensions. To evaluate our approach on as many extensions as possible we include both old and young extensions. As it can be seen in Figure 2, to detect both malicious and potentially vulnerable extensions, we split our methodology into i) static, and; ii) dynamic analysis. Finally, we installed the potentially vulnerable extensions and confirmed whether they are exploitable or not.

To improve efficiency and harden the analysis we combine both static and dynamic analysis. By combining them we no longer need to dynamically test all extensions, which would require more resources. In many cases, we can do this without losing precision, e.g., by statically analyzing the manifest we know if an extension can manipulate web requests to perform traffic stealing. If the extension lacks the permission we can skip the dynamic analysis.

Static Analysis. During the static analysis, we analyze the manifest key `externally_connectable` according to Table 2. We also parse both permissions and the optional permissions of the extensions to know what sensitive information the extension has access to. This takes about 12 hours on a normal workstation.

We use Esprima[2], a powerful library to perform lexical (tokenization) and syntactic (parsing) analysis of JavaScript, to generate the Abstract Syntax Tree (AST) of the JavaScript files and extract the external message event listener's code. For HTML files, we use regular expressions instead to extract the code.

We find 53,177 potentially malicious extensions containing either `runtime.connect()` or `runtime.sendMessage()` functions. We also find 11,595 extensions that are susceptible to being abused. That is, they implement event listeners for external message passing functions, e.g., `onMessageExternal()`.

After, we compute the SHA-256 hash of the event listener functions and group extensions by their hashes, obtaining 570 different implementations. The use of cryptographic hashes, as opposed to fuzzy hashes like ssdeep [24], ensures that the functions are the same. We discuss the implications of this in Section 8. Then, we chose one extension from each of the 570 groups and dynamically analyzed them. Note that since each extension in the group has the same function it does not matter which one we pick.

[2]https://esprima.org

Dynamic Analysis. Based on the static analysis we extracted extensions that can be further tested for dynamic analysis. The dynamic analysis consists of three major steps: 1) instrumentation; 2) monitoring, and; 3) payload generation. The exact implementation of these differ from attack to attack and is explained in more detail in Section 5.

In general, the extensions are statically instrumented to log the line numbers being executed. This is done by inserting fetch instructions on each line, like the following: `fetch("http://localhost/inst/'+token()+'/line/'+(line_number)+'")`. For each attack, we also hooked API calls, including the arguments, and exfiltrate them by using similar instructions and encodings.

To monitor we use a web server that listens to these fetch instructions. Based on the requests we can determine the trace of the extension's execution and if any security and privacy-relevant APIs were executed.

Finally, the last part is generating payloads or instructions to bring the extension to an interesting state. We used puppeteer [31], which is a tool for controlling Chrome, to load the extension we wanted to test and any other extensions we developed to interact with it. From here the instructions depend on the type of attack. For example, our extension can send inter-extension messages to the other extension. We can load pre-configured chrome profiles to test history poisoning attacks or navigate the browser to test traffic stealing. Dynamically testing one extension takes around 1 minute, but many can be done in parallel and after the static filtering we are only dynamically executing a subset of all extensions.

Manual Verification. When the static or dynamic analysis marks an extension as either potentially vulnerable or malicious we make sure to manually verify it. To manually verify a malicious extension we run the original version of the extension, i.e., without instrumentation code and other modifications, in a normal Chrome browser, not via puppeteer. From there we verify that the same malicious behavior, for example, traffic stealing, is still present. Similarly, for vulnerable extensions, we also use the original extension and load it in Chrome together with a suitable attacker extension we create.

To avoid manually testing every vulnerable or malicious extension, we cluster them using *DeDup.js* [30]. *DeDup.js* allows us to find all extensions that have the same malicious or vulnerable JavaScript files. We also check the manifests to ensure the JavaScript files are loaded and executed. We discuss the risk of possible false positives from this approach in the discussion in Section 8.

5 Discovering Attacks and Vulnerabilities

In this section, we present the novel attacks that we designed as well as both the vulnerable and the malicious extensions we found in the wild. Our novel attacks are presented in Section 5.1, the attacks used by malicious extensions are presented in Section 5.2, while the vulnerable extensions are in Section 5.3. We analyze these

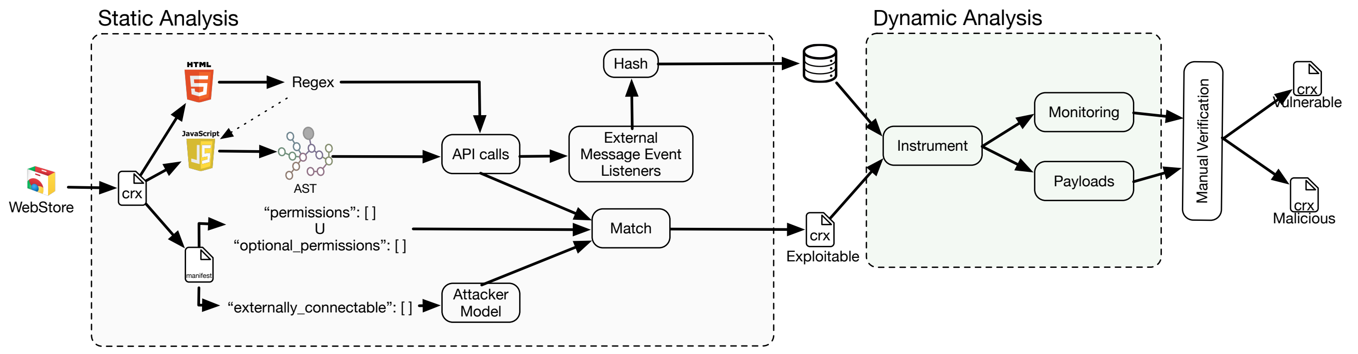

Figure 2: Static and dynamic analysis combination to determine possible attacks and detect malicious and potentially vulnerable extensions.

attacks and vulnerabilities in Chrome but discuss how they apply to other browsers in Section 8.4.

5.1 Novel Attacks by Malicious Extensions

In this section, we detail two novel attacks we designed to steal users' credentials. As these attacks are designed by us and not discussed in prior works, we did not expect to find any of these in the wild. We did not find any extensions using these attacks, further supporting their novelty.

5.1.1 Password stealer A password-stealing extension can attack any actor in the attacker model, i.e., other extensions, web pages, or users. We have seen previous attacks [42] where extensions inspect the DOM to steal passwords.

In this paper, we present a novel active password-stealing approach where the extension actively visits different domains to extract the passwords. A novel component to this is a bypass of Chrome's protection against autofill-scraping [43]. Note that this protection is so far only in Chromium-based browsers, i.e., not in Firefox or Safari yet. The attack still works in Firefox but not in Safari as it requires the user to click a pop-up before the password is added to the field. Chrome protects the user from autofill attacks by waiting for user interaction before adding the password to the DOM. However, we can edit the underlying password element to bypass the protection. Our attack changes the type of the password element to `text` instead of `password`. Although the value is still not added to the DOM, the asterisks are converted into text. Our extension takes a screenshot of the page to exfiltrate the image with the password. Note that taking screenshots of the pages only requires the `<all_urls>` permission, which is used by over 18,000 extensions, including popular extensions like Grammarly with over 10 million downloads. Furthermore, no special screenshot permissions or user interaction are required.

Making the attack stealthy to the user is an orthogonal problem. We use a pop-under attack which creates a new window under the active one. In this new window, we load a page and take a screenshot. This approach makes the attack stealthy on both macOS and Ubuntu. Another approach can be to modify the style of the DOM to make it harder for the user to see the unmasked password. In addition, the attack takes less than five seconds to steal a password in our tests, with possible variation due to network speed, making it hard to stop even if the user notices it.

5.1.2 Virtual keyboard attack To increase the security of the user's credentials, some online services include virtual keyboards on the screen for the users to avoid directly writing the passwords and thus protect them from keylogger attacks. For this attack to work, the attacker has to implement an click event listener using `document.onClick`. Later on, in an offline analysis, the attacker matches both the coordinates and the screenshot to get not only the clicked elements but also the order in which they were clicked. This login system is popular among banks, e.g., ING bank in Spain, France, Australia, BNP Paribas, and La Banque Postale in France, without the option for the users to use an alternative.

5.2 Malicious Extensions in the Wild

In this section, we present extensions attacking users or other extensions.

5.2.1 Traffic stealing Extensions have the power to cancel, redirect, modify request and response headers, and supply authentication. This allows them to implement features like ad-blocking by matching URLs against deny-lists. The permissions needed to modify ongoing network requests are `webRequest` and `host_permissions`, and `webRequestBlocking`. Using these permissions and APIs, an extension can attack the user by intercepting general search queries, for example to Google or Yahoo, redirect these back to their server, and finally have their server redirect the user back to a search engine. Without the user noticing, their search traffic is stolen.

Results. To detect malicious extensions exploiting this attack, we analyzed those that interact with network requests. In particular, extensions that used the `webRequestBlocking` permission and defined a listener for `onBeforeRequest` were dynamically analyzed. We clustered extensions based on the SHA-512 hash of the file responsible for the traffic stealing.

Before dynamically executing the extension, we instrumented it to send the list of filtered URLs to our control server at runtime. These are the potential URLs the extension can interact with. We then compared this to a list of domains, plus query parameters, we wanted to test against. The domains (`www.google.com`, `www.yahoo.com`, and `www.bing.com`) were picked based on a short pilot study of malicious extensions we found, we discuss improvements to this method in Section 8. For each match, we dynamically executed and analyzed the extension. To detect network requests we use `page.setRequestInterception` in Puppeteer. Using this we notice if the extension introduces new requests or redirects.

Example. We found 4,410 extensions abusing this method to steal traffic from users. We discuss these in detail in Section 6.

5.2.2 Collusion Extensions can communicate directly to other extensions by sending messages. The possibility for message passing depends on the `externally_connectable` definition (see Table 2). Malicious extensions can abuse message passing to share permissions, allowing multiple low permission extensions to combine their permissions. This would allow them to stay under the radar.

Results. To detect collusion we looked for message passing between extensions. As can be seen in Figure 1, the `sendMessage` function takes an extension ID as a parameter. Therefore, we scanned each extension for extension IDs in their code.

Extension IDs follow a simple pattern, 32 lower case characters. We first scanned each file using this regular expression in python, `re.findall("[a-z]{32}")`. This returns all the strings that could potentially be other extensions. Next, we compared all the potential extension IDs with the real ones in the dataset we are using. To further filter the list we removed the extensions that do not use message passing APIs. This helps to remove extensions that simply link to other extensions, without directly messaging them. Based on this, we created our "collusion map" which we then manually analyzed to understand why the extensions interact with each other.

Example. In this paper, we did not find any colluding extensions that could be classified as malicious. However, we did find two extensions colluding with each other to share permissions. The first one[3] defines three permissions while the second one[4] requires no permissions. The second one then asks the first one for data.

5.3 Vulnerable Extensions in the Wild

In this section, we present the vulnerable extensions we found during our analysis.

5.3.1 History poisoning History poisoning is a novel attack we present which targets extensions working with users' browser histories. Malicious extensions, or web pages, can poison the browser history with code injection payloads. The vulnerable extensions usually present the user with an overview of their history. This can range from exact history to most visited or recently closed tabs. By adding HTML code to the title of a web page, a web attacker can gain content script code execution if the history titles are not sanitized correctly. The exploitability of this attack depends on the type of history poisoning necessary. Changing the most visited pages is difficult for a web attacker to accomplish. However, adding a single entry to recent history or closed tabs only requires a redirection.

Results. To detect potentially vulnerable extensions due to this attack, we first extract extensions with the `history` permission. Once a vulnerability is found we automatically mark other extensions which share the same file and use the history permission. Later, in the dynamic phase, we loaded the history with visits to pages with HTML and JavaScript payloads in the title. These payloads are explained in Section 6. We then loaded the extension and if a payload is executed we conclude that the extension is vulnerable. Based on the CSP in the manifest, this can either imply full XSS or be limited to HTML injection.

Example. In Section 6 we present a detailed analysis of extensions vulnerable to this attack.

5.3.2 Code execution If an extension uses dynamic code execution functions, such as `eval`, without safety precautions it can potentially be abused by other web pages and extensions.

The impact of this attack depends on the context of the code execution. The highest impact context is the background scripts where, if exploited, an attacker would gain access to all the APIs the extension has permission to use. The impact in the DOM context depends on the connection to the background. By executing code in the DOM context, the attacker gains access to all the APIs available for the content scripts. For instance, the attacker can make use of the `sendMessage` function and impersonate the legitimate content scripts of the extensions, allowing the attacker to send messages to the background as if she were the content scripts, affecting the confidentiality, integrity, and availability of the extension.

Results. To detect code execution vulnerabilities, we extracted the code from the message passing functions in Figure 1. We did this by analyzing and extracting listeners like `onMessageExternal`. From this, we clustered the extensions based on the SHA-512 hash of the code of these functions. For each cluster, we extracted the API calls from the listener functions to check if they either execute code dynamically, e.g., by `eval`, or interact with shared resources, e.g., local storage. We then checked these API calls against the permissions that the extension uses. Finally, we analyzed the clusters with API calls to determine if they can lead to code execution.

To find more extensions we search for others containing the same file, and if the permissions are correct, we mark them as vulnerable. This could lead to false positives if another extension has the same file but does not use it. However, in our manual testing, we found no such cases.

Example. In this paper, a group of 1,349 extensions was vulnerable to code execution attacks. We discuss them in-depth in Section 6

5.3.3 Fingerprinting Knowing which extensions a user has installed can allow both extension and web page attackers to degrade the privacy of a user. Multiple methods of browser extension fingerprinting have been proposed. Sjösten et al. [36] used WARs; Starov et al. [38] identified extensions by the changes they perform to the DOM; [23] et al. do so by sending messages and waiting for their response; recently, [25] et al. used CSS styles. However, extensions with the "management" permission can use the `chrome.management.getAll` function to get a list of all installed extensions. We show how extensions can bypass the "management" permission and get this list. Note that malicious extensions do not require any extra capabilities or permissions to bypass the permission and perform the attack.

Our technique is based on the `externally_connectable` property. Recall this allows extensions and web pages to send messages to another extension using inter-extension messaging. If that property is not in the manifest, by default the target extension will accept

[3]dnclbikcihnpjohihfcmmldgkjnebgnj (version 1.3.6)
[4]bepofoammpdjhfdibmlghoaljkemineg (version 0.1.4)

Table 3: Browser extensions fingerprinting.

	Methodology	Browser Extensions	
		Fingerprintable	Non-Fingerprintable
2019	This paper	10,785 (8.0%)	665
	WARs[36]	10,919 (8.1%)	531
	WARs [36] ∩ Bloat [38]	10,977 (8.2%)	473
	This paper ∩ [36] ∩ [38]	11,147 (8.3%)	303

all messages coming from extensions. However, if the property is defined, then two main properties can be defined: `matches` and `ids`. Both are allowed-lists where only the web pages and extensions defined can send direct messages to the target extension.

Results. To detect how many extensions are fingerprintable due to external messages, we parsed the manifest file of all the 133,365 we downloaded as of January 2020. 126,948 extensions **do not** define the `externally_connectable` key in the manifest. Remember that if this key is not in the manifest file but the extensions implement listeners for external communications, e.g., `onMessageExternal`, web pages cannot send direct messages through message passing functions but other extensions can.

From the 126k extensions, we statically check how many of them implement any of the methods to handle external connections and got a lower bound of 11,595 potentially fingerprintable extensions. Later, we automatically installed them all—146 could not be analyzed because the manifest had syntactic errors and the bowser could not install them—and certified that 10,785 use the `sendResponse()` function to send a response back to the sender.

We then coded two fingerprinting attacks proposed by 1) Sjösten et al. [36], where extensions are fingerprintable due to the WARs they publicly expose, and; 2) Starov et al. [38], where extensions are fingerprintable due to bloat (changes they automatically perform over the DOM). In Table 3 we summarized our findings and the combination with prior work in the field. Despite being aware of a recently published work [23], we could not reproduce the proposed attack being therefore impossible for us to corroborate their findings and include them in the final result. After executing the three fingerprinting attacks (third row of Table 3), we successfully fingerprinted 11,147 extensions (approximately 8.3% of the total extensions) where 303 were not possible to do so with these techniques. The reason for not being possible to be fingerprinted is because either they do not send a message back using the `sendResponse()` method, they do not define WARs, or they do not automatically modify the web content provided by the server.

6 New Tabs Case Study

Our empirical study has uncovered a remarkable cluster of "New Tab" extensions. The main characteristic of these extensions is that they override the new tab functionality in the browser. When the user opens a new tab in the browser, this is replaced by the one the new tab extension created. Such new tabs usually have a search bar with some arbitrary wallpaper backgrounds. Some extensions also add widgets like todo-notes, weather reports, email, a list of the most visited sites, and bookmarks management among others.

Table 4: Distribution of file versions for `search-overwrite.js` and domains used to steal queries.

URL	# Extensions	SHA[0:3]
s.tablovel.com	1740	026
www.explorenewtab.com	1178	1f8
www.newtabprobe.com	560	191
www.newtabexplore.com	539	513
www.lovelychrometab.com	255	381
www.newtabexplore.com	82	6d9
www.themefornewtab.com	40	c49
www.newtabwallpapers.com	8	7e1
www.newtabwallpapers.com	7	5dc
www.newtabprobe.com	1	732

Our analysis has flagged many vulnerable and malicious extensions in this category, justifying further investigation. In the following section, we analyze them in more detail.

Traffic stealing. We found 4,410 extensions, with a combined 176 thousand downloads, that steal search queries from users while posing as new tab extensions. There is a common file in all these extensions called `search-overwrite.js` which implements an event listener that is fired just before a web request is sent (`chrome.webRequest.onBeforeRequest.addListener`). In such a function, all the extensions block the ongoing request and redirect it to different URLs, like s.tablovel.com. We realized that, in addition to stealing queries from popular search engines like Google, a subset of them, containing 2,588 extensions, also steal queries from other new tab extensions, e.g., from `redirect.lovelytab.com`.

Regarding the `search-overwrite.js` file, from all the 4,410 extensions, we computed all the SHA-512 hashes of such a file in different extensions and found that there are 10 unique versions. In Table 4 we show the distribution of the files and the URL used to steal the queries. As can be seen, the two most popular versions of that file are used by a majority of the extensions.

Code execution. In addition to finding malicious new tab extensions, we also found multiple vulnerable ones. In particular, we found that 1,349 extensions were vulnerable to XSS attacks from other extensions. At the time of download, these extensions had over 73 million downloads combined, making them quite popular. However, we do believe many of these downloads were fraudulent and some extensions have lost many of their downloads since then. This attack depends on three factors: 1) the extensions allow external messages; 2) the external messages are stored and reflected in the extension, and; 3) the CSP in the manifest allows for JavaScript execution. The problem in these extensions is that data from the `localStorage` is reflected unsanitized, as is the case for the "Peppa Pig HQ Wallpapers New Tab"[5] with over 3,000 downloads. For the 1,349 extensions, all of the above criteria were met. We also found two other groups of new tab extensions, totaling 2,829 extensions, which met all but the last criterion. This means that XSS is no longer

[5]cikheolhmcgdkmblgmfkgkcgfflddaem (version: 3.2) now removed.

possible but HTML injections still are, allowing attackers to inject ads or meta redirects, effectively taking over the new tab extension.

History poisoning. A popular function with new tab extensions is showing both recently and most visited websites. We set up a local server and installed the extensions in the browser. The browser is running a profile with multiple poisoned shared resources, i.e., before running the extension, history data are added to the profile. We found that if we change the `<title>` tag of a page in the history, we were able to remotely execute code in the user's browser.

As an example, we use "Halloween Backgrounds New Tab"[6] and the title payload `<script>alert(1)</script>`. When the extension reads our web page from the most visited list of the user, which happens when the user opens a new tab of the browser or when the user launches the browser, the payload is executed. We found that the same group of extensions being vulnerable to inter-extension XSS is also vulnerable to XSS from history poisoning. In total, all 1,349 are vulnerable to XSS and 2,829 to HTML injection from history poisoning.

Evasion. We discovered that New Tab extensions actively practice evasion mechanisms, making their detection challenging for purely dynamic mechanisms. It turned out that 53 of them share a file that implements a time bomb logic that waits five days after installation before starting to redirect traffic.

7 Countermeasures

In this section, we briefly discuss possible countermeasures for the attacks we discovered.

Password stealing. Stopping malicious extensions from stealing passwords is complicated. Many legitimate password manager extensions need the capability to both read and write passwords to the DOM. Thus, simply blocking access to password fields is not viable. To stop the more nefarious type of password-stealing, which is active password-stealing where no user input is needed, Chrome decided to hide autofilled passwords from the DOM until the user interacts with the page, which our attack bypasses with screenshots.

To protect against extensions screenshotting passwords there are three avenues. First, warning the users about the screenshot being taken, similarly to what Chrome does for full desktop screenshots already. A similar approach for screenshots using `chrome.tabs.captureVisibleTab`, would clearly show the password being leaked to the user.

The second approach is for the browser to hide the password before taking the screenshot. As the browser knows which field it inserts the saved password into it can either obfuscate or fully remove this field before taking the screenshot. Then after the capture is completed, add back the field again. The third approach is to create a new permission such that extensions that want to use the `captureVisibleTab` function should declare in advance such permissions so the user has to specifically approve such privileges.

Traffic stealing. With the adoption of the new manifest version 3, the `webRequestBlocking` permission will be moved to the `chrome.declarativeNetRequest` API [15]. If extensions would want to block web requests by using the `chrome.webRequest` API, they will have to declare that in advance in a set of rules. With this, traffic stealing will not be solved but extensions will have to explicitly declare the purpose in advance, being easier to detect and block by either Google or users.

Inter-extension attacks. In this paper, we conclude that one of the most effective methods to avoid undesired extensions to send messages to others is using the `externally_connectable` key in the manifest, similar to what Somé [37] suggested. We developed a script that automatically inserts `externally_connectable` into the manifest of those extensions without it. After, we run the inter-extension attacks presented in this paper and certified that the extensions were not exploitable anymore due to external messages.

A common problem in both the extension being vulnerable to inter-extension XSS and history poisoning attacks was that they did not sanitize user-controlled input. These problems could have been avoided by properly sanitizing content before presenting it to the user, whether it is todo-notes or an overview of the browser history. There are many good JavaScript libraries available to sanitize input before adding it to the DOM, e.g., Both jQuery and DOMPurify.

For XSS specifically, strict CSP policies are very effective. Chrome already applies a default strict CSP policy protecting against inline XSS attacks. However, as we found in this paper, many extensions override this with weaker policies. Making it clearer to both the developers and users that weak CSP policies can lead to the extension and user data being compromised might help raise awareness of this problem.

8 Discussion

In this section, we discuss the limitations of our work and how we try to minimize both false positives and false negatives. We also discuss deviating results in the Safari browser.

8.1 Static analysis

There are some inherent limitations in using static code analysis. In the case of JavaScript files, there are two main aspects to consider: dynamic function execution and code obfuscation. Dynamically executed code inside an `eval()` statement would not be parsed and analyzed correctly. Obfuscated JavaScript code is another well-known problem. During our manual analysis, we did find a heavily obfuscated traffic stealing extension that used a base64-encoded string to evade static detection. Thus, even if we use basic regular expression searches in the code, they can be evaded. To improve this future static analysis approaches should focus more on deobfuscation and attempt to decode and decrypt data.

Due to dynamic code execution and obfuscation, our analysis may miss potentially malicious extensions resulting in false negatives. However, for the code in the extension that can be parsed, we ensure that the permissions and APIs we extract are in the extension's code.

8.2 Dynamic analysis

Before we dynamically executed the extensions, we instrumented them to relay useful information about API calls. However, the instrumentation is done statically, which inherits similar limitations as mentioned in the static analysis. In the case of traffic stealing

[6] jckojlfhehjnlkdoiojpmkjjoojbgfjl (version: 0.1.8.4) now removed.

detection, this means that we would not be able to acquire the exact filter list the extensions use. We handled this by dynamically testing the extension on all the URLs in our target list, which means that we tested everything but at the cost of worse performance.

For traffic stealing, we tested extensions against three URLs we had seen being targeted by traffic stealers, namely `www.google.com`, `www.yahoo.com`, and `www.bing.com`. Similarly, by executing the extension multiple times on the same URL we could reduce the noise in terms of requests being made, thus lowering false positives. For example, visiting `www.google.com` might result in different requests between two visits. By making multiple visits both with and without the extension we could determine which requests are introduced by the extensions. Future work includes testing more URLs than the three we picked. This can lower the false negatives by potentially finding more malicious extensions.

8.3 Manual Analysis

To further minimize any false positives from the static and dynamic analysis we test the original extension in a plain Chrome browser. We test one extension from each cluster of extensions, where the cluster is all the extensions that have the same vulnerable or malicious JavaScript file. We also check that the manifests for all the extensions in the cluster contain the same file in the manifest to ensure it is not simply unreachable code.

8.4 Cross-browser

For each attack and vulnerability we recreated them in both Firefox and Safari. In general, they worked similarly in Firefox but failed in Safari. This is because browser extensions in Safari follow a slightly different model than both Chrome and Firefox. In Safari, extensions are composed of three main parts [2]: 1) a macOS app; 2) the browser extension—similar to Chrome and Firefox, and; 3) a native app extension that mediates between both the macOS app and the browser extension. In addition to that, the number of permissions available for extensions is limited in comparison to the other browsers. As a consequence, most of the attacks explained in Section 5 are not exploitable in Safari but the password stealer.

9 Related Work

From the privacy point of view, we differentiate among approaches that focus on how browser extensions handle the (sensitive) information that they have access to [1, 10–12, 17, 41], approaches that demonstrate how extensions can be used to fingerprint users [23, 36, 38–40], and approaches that analyze the permissions of the extensions [4, 6–9, 19, 28]. From the security point of view, extensions can be used to execute malicious code [3, 5, 37] and how extensions can cooperate to execute collusion attacks [33].

Often, these works pursue attacker models and capabilities based on concrete attacks, e.g., fingerprinting and information leaking. To complement the depth, our approach offers the breadth of systematically analyzing the attack surface for the browser extension ecosystem, identifying gaps that lead to discovering new attacks.

Mystique [12] proposes a powerful approach to taint privacy information in browser extensions by modifying the V8 engine of Chromium. This approach is suitable for detecting traffic stealing attacks, assuming the complexity of V8 does not break the soundness of taint tracking [44]. In this paper, we used a complementary technique based on a combination of static and dynamic analysis with clustering of similar extensions. While Mystique is powerful enough to find the traffic stealers, the benefit of our approach is its performance efficiency and independence of the browser engine.

We review the literature on the paper's three main themes: password-stealing, traffic stealing, and inter-extension attacks individually.

Password stealing. The security of password managers has been widely evaluated, resulting in discoveries of security flaws in the autofill functionality, being vulnerable to web pages managed by an attacker [26, 27, 35]. In this paper, we demonstrated how a malicious browser extension can transform the password field of the DOM into text, take a screenshot of the password, and transform it back without the user noticing.

Traffic stealing. In 2014, Hulk [22] was one of the first papers pointing out that extensions are sniffing users' communication traffic. The authors analyzed 48,332 extensions and marked 130 as malicious and 4,712 as suspicious. Seven years later, we realized that this technique is still being used by 4,410 extensions that capture users' searches and forward that information to external servers apart from performing the original request. To our surprise, most of these extensions fall into the "HD Wallpapers New Tab" group where they claim to modify Chrome's new tab page.

Inter-extension. Many authors focused on fingerprinting browser extensions. The common adversarial model is a web page that attempts to track users using the extensions they have running in the browser. Probing for WARs [36], detecting the changes the extensions automatically perform over the DOM [38], using message passing to know which extension replied to the message [40] or executing timing attacks [34] are a few examples of how browser extensions can be fingerprinted. In our paper, apart from considering web pages, we also include browser extensions as attackers and knowing which extensions the user has without declaring the "management" permission.

Regarding collusion, very little effort has been made in this area in browser extensions. As far as we know, Saini et al. [33] were the first authors who demonstrated that legitimate extensions can collude to achieve malicious goals and showed how a malicious extension can misconfigure the browser or other extensions using message passing. Similarly, Buyukkayhan et al. [7] showed how the API that Firefox exposes to extensions, known as Cross Platform Component Object Model (XPCOM), can be exploitable by other extensions because of a lack of isolation mechanisms, being only able to be exploited in previous versions of Firefox. In contrast, even though Chrome isolates the execution of the extensions in different environments, we demonstrate that browser extensions in Chrome not only can modify others' configurations but also cooperate to achieve a common goal.

Pantelaios et al. [29] investigate 922,684 extensions and find 143 malicious extensions where 64 were still online. From these 143 extensions, they detected that 16 changed the search engine of the user. To do so, they use the users' feedback (a combination of rating and comments) and a clustering algorithm to classify and detect malicious extensions. With our systematic study of attack entry points and a combination of both static and dynamic analysis, we discover 4,410 that steal search queries of the user by redirecting them to external servers.

10 Conclusions

As a step toward filling the gap in the security analysis of the browser extension ecosystem, we have presented a systematic study of attack entry points leading us to novel methods for password stealing, traffic stealing, and inter-extension attacks. Because extensions are highly privileged, it pays off for the attacker to target vulnerable extensions, leading to possibilities of exfiltrating secrets and performing unauthorized modification. Combining static and dynamic analysis we have shown how to discover extension attacks and study their prevalence in the wild. Our findings indicate that 1,349 extensions are vulnerable to cross-extension messaging passing attacks leading to XSS. We also discovered a remarkable cluster of "New Tab" extensions where 4,410 extensions in this class perform traffic-stealing attacks. We have suggested countermeasures for the uncovered attacks.

Coordinated disclosure. Our disclosure report to Google includes not only malicious and vulnerable extensions but also recommendations on mitigating inter-extension attacks, as well as our findings on password and traffic stealing, which require browser support for the countermeasures. All of the 4,410 reported traffic stealers have now been deleted. We are in contact with Google about the rest, including the password stealer which is currently being triaged by their security team.

Acknowledgments. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Facebook.

References

[1] A. Aggarwal, B. Viswanath, L. Zhang, S. Kumar, A. Shah, and P. Kumaraguru. 2018. I Spy with My Little Eye: Analysis and Detection of Spying Browser Extensions. In *Euro S&P*. 47–61.
[2] Apple. 2021. Messaging Between the App and JavaScript in a Safari Web Extension. https://developer.apple.com/documentation/safariservices/safari_web_extensions/messaging_between_the_app_and_javascript_in_a_safari_web_extension.
[3] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. 2011. Vetting Browser Extensions for Security Vulnerabilities with VEX. *Commun. ACM* 54, 9 (2011).
[4] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. 2010. Protecting Browsers from Extension Vulnerabilities.. In *NDSS*.
[5] A. Barua, M. Zulkernine, and K. Weldemariam. 2013. Protecting Web Browser Extensions from JavaScript Injection Attacks. In *ICECCS*. 188–197.
[6] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, and Yuan Tian. 2014. Analyzing the dangers posed by Chrome extensions. In *CNS*,. 184–192.
[7] Ahmet Salih Buyukkayhan, Kaan Onarlioglu, William K. Robertson, and Engin Kirda. 2016. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In *NDSS*.
[8] S. Calzavara, M. Bugliesi, S. Crafa, and E. Steffinlongo. 2015. Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions. In *PLAS*.
[9] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An Evaluation of the Google Chrome Extension Security Architecture.. In *USENIX Sec.* 97–111.
[10] Wentao Chang and Songqing Chen. 2013. Defeat Information Leakage from Browser Extensions via Data Obfuscation. In *ICICS*. 33–48.
[11] W. Chang and S. Chen. 2016. ExtensionGuard: Towards runtime browser extension information leakage detection. In *CNS*. 154–162.
[12] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *CCS*. 1687–1700.
[13] Chrome. 2019. Content scripts. https://developer.chrome.com/docs/extensions/mv2/content_scripts/.
[14] Chrome. 2020. Chrome extensions permission model. https://developer.chrome.com/extensions/declare_permissions.
[15] Google Chrome. 2020. Migrating to Manifest V3. https://developer.chrome.com/extensions/migrating_to_manifest_v3.
[16] crytpo-wallet-steal-2020 2020. Google Pulls 49 Cryptocurrency Wallet Browser Extensions Found Stealing Private Keys. https://news.bitcoin.com/google-cryptocurrency-wallet-browser/.
[17] M. Dhawan and V. Ganapathy. 2009. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *ACSAC*. 382–391.
[18] Firefox. 2020. Firefox extensions permission model. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/permissions.
[19] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. 2011. Verified Security for Browser Extensions. In *S&P*. 115–130. https://doi.org/10.1109/SP.2011.36
[20] Sam Jadali. 2019. DataSpii: The catastrophic data leak via browser extensions. https://securitywithsam.com/2019/07/dataspii-leak-via-browser-extensions/.
[21] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. Abu Rajab, and K. Thomas. 2015. Trends and Lessons from Three Years Fighting Malicious Extensions. In *USENIX Sec.* 579–593.
[22] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *USENIX Sec.* 641–654.
[23] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. 2020. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting. In *NDSS*.
[24] Jesse Kornblum. 2021. ssdeep - Fuzzy hashing program. https://ssdeep-project.github.io/ssdeep/.
[25] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *USENIX Sec.*
[26] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. 2014. The Emperor's New Password Manager: Security Analysis of Web-based Password Managers. In *USENIX Sec.*
[27] Xu Lin, Panagiotis Ilia, and Jason Polakis. 2020. Fill in the Blanks: Empirical Analysis of the Privacy Threats of Browser Form Autofill. In *CCS*.
[28] Lei Liu, Xinwen Zhang, Guanhua Yan, Songqing Chen, et al. 2012. Chrome Extensions: Threat Analysis and Countermeasures.. In *NDSS*.
[29] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. 2020. You've Changed: Detecting Malicious Browser Extensions through Their Update Deltas. In *CCS*. 477–491.
[30] Pablo Picazo-Sanchez, Maximilian Algehed, and Andrei Sabelfeld. 2022. DeDup.js: Discovering Malicious and Vulnerable Extensions by Detecting Duplication. In *International Conference on Information Systems Security and Privacy (ICISSP)*.
[31] puppeteer. 2021. puppeteer. https://github.com/puppeteer/puppeteer.
[32] Reuters. 2020. Exclusive: Massive spying on users of Google's Chrome shows new security weakness. https://www.reuters.com/article/us-alphabet-google-chrome-exclusive/exclusive-massive-spying-on-users-of-googles-chrome-shows-new-security-weakness-idUSKBN23P0JO.
[33] A. Saini, M. Singh Gaur, V. Laxmi, and M. Conti. 2016. Colluding browser extension attack on user privacy and its implication for web browsers. *Computers & Security* 63 (2016), 14 – 28.
[34] I. Sánchez-Rola, I. Santos, and D. Balzarotti. 2017. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *USENIX Sec.*
[35] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. 2014. Password Managers: Attacks and Defenses. In *USENIX Sec.* 449–464.
[36] A. Sjösten, S. Van Acker, P. Picazo-Sanchez, and A. Sabelfeld. 2019. Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks. In *NDSS*.
[37] D. F. Somé. 2019. EmPoWeb: Empowering Web Applications with Browser Extensions. In *S&P*. 227–245.
[38] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. 2019. Unnecessarily Identifiable: Quantifying the Fingerprintability of Browser Extensions Due to Bloat. In *WWW*. 3244–3250.
[39] Oleksii Starov and Nick Nikiforakis. 2017. Extended Tracking Powers: Measuring the Privacy Diffusion Enabled by Browser Extensions. In *WWW*. 1481–1490.
[40] O. Starov and N. Nikiforakis. 2017. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *S&P*. 941–956.
[41] Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. 2007. Extensible Web Browser Security. In *DIMVA*. 1–19.
[42] usmedicalit. 2020. Another Chrome extension is stealing passwords. https://www.usmedicalit.com/2018/09/18/another-chrome-extension-is-stealing-passwords/.
[43] vabr@chromium.org. 2016. Issue 636425: Value of Autofilled input[type="password"] Shows in DOM as Empty. https://bugs.chromium.org/p/chromium/issues/detail?id=636425/.
[44] Mengfei Xie, Jianming Fu, Jia He, Chenke Luo, and Guojun Peng. 2020. JTaint: Finding Privacy-Leakage in Chrome Extensions. In *ACISP*. 563–583.