First, Do No Harm: Studying the manipulation of security headers in browser extensions

Shubham Agarwal
Saarland University
s8shagar@stud.uni-saarland.de

Ben Stock
CISPA Helmholtz Center for Information Security
stock@cispa.de

Abstract—Browser extensions are add-ons that aim to enhance the functionality of native Web applications on the client side. They intend to provide a rich end-user experience by leveraging feature-rich privileged JavaScript APIs, otherwise inaccessible for native applications. However, numerous large-scale investigations have also reported that extensions often indulge in malicious activities by exploiting access to these privileged APIs such as ad injection, stealing privacy-sensitive data, user fingerprinting, spying user activities on the Web, and malware distribution.

In this work, we instead focus on tampering with security headers. To that end, we analyze over 186K Chrome extensions, publicly available on the Chrome Web Store, to detect extensions that actively intercept requests and responses and tamper with their security headers by either injecting, dropping, or modifying them, thereby undermining the security guarantees that these headers typically provide. We propose an automated framework to detect such extensions by leveraging a combination of static and dynamic analysis techniques. We evaluate our proposed methodology by investigating the extensions' behavior against Tranco Top 100 domains and domains targeted explicitly by the extensions under test and report our findings. We observe that over 2.4K extensions actively tamper with at least one security header, undermining the purpose of the server-delivered, client-enforced security headers.

I. INTRODUCTION

The Web has become increasingly popular over time to provide a wide range of services to their end-users over the Internet. At the same time, the complexity of the client-side code among Web applications has also increased to provide a better user experience at their disposal. Browser extensions, also known as plug-ins or add-ons in different browsing environments, are an integral part of modern browsing architecture. Third-party entities usually develop them to provide additional client-side functionalities and enhance the user's browsing experience, for example, by improving the appearance of Web sites, integration with popular third-party services, password management, and access to users' resources. They are more powerful than native Web applications, owing to the feature-rich JavaScript APIs that they have at their disposal. Thus, they often mediate privacy-sensitive information and perform security-critical operations for the users.

Web applications are regularly the targets of different attacks from Cross-Site Scripting through framing-based attacks to TLS downgrading. Researchers and practitioners have develop different mitigations for these attacks, which are typically delivered through HTTP headers from the server and subsequently enforced by the client. For instance, the server may define a Content-Security-Policy header to control script inclusion and framing from third-party pages. However, given the aforementioned capabilities, extensions may modify or drop such headers, effectively disabling well-configured security mechanisms and thus undermining the applications' security which is supposed to be enforced by the client.

Numerous studies by the research community have continually reported the abuse of these extensions as vectors to carry out malicious operations, leading to potential privacy and security implications. Previous large-scale studies by various researchers indicate that they often spy on user browsing history, steals privacy-sensitive user information, or illegitimately modifies the content of Web pages [2, 14, 25, 31]. Recent studies further assert that these *man-in-the-browser* entities are persistently abused in the wild and may have grave implications [26, 27, 32]. While these studies indicate that the extensions often intercept and modify the security headers at runtime, we emphasize that a systematic investigation is essential to determine, categorize and quantify the threats associated with them and propose effective countermeasures to tackle these vulnerabilities.

In this paper, we specifically analyze extensions that intercept, inject, drop or modify HTTP security headers within the response. The Web server specifies HTTP security headers and sends them along with the response so that the browser enforces the corresponding security protocol at the client. Tampering with these headers may hamper the client-side security of the applications and expose them to vulnerabilities that the headers aim to mitigate. To study this behavior at scale, we propose an automated framework to analyze and detect such extensions using a combination of static and dynamic analysis strategies. We also review the risks associated with these headers' alteration and their impact on the application security and user privacy. To this end, we investigate over 186K publicly available Chrome extensions. The proposed framework statically analyzes the codebase and crawls on the Tranco Top 100 domains and other target hosts to identify potentially harmful extensions. We observe from our findings that most of these extensions that tamper with the security headers also disable at least one of the corresponding client-enforced security mechanisms.

To summarize, the key contributions in this study are as follows:

- We present our automated approach to identify browser extensions that manipulate security headers, ultimately undermining the security configurations desired by the Web application server.

- Based on the proposed methodology, we conduct a large-scale study with over 186K Google Chrome extensions and show that 2.4k actively undermine the clients' security.

- We conduct a deep dive into the modifications of CSP, highlighting that extensions even take the seemingly non-security relevant decision to add directives to the delivered policies.

- We outline the implications of other security headers being manipulated and propose expansion of our approach as part of future work.

II. TECHNICAL BACKGROUND

In this section, we briefly discuss the extension architecture and its permission model. We also describe the security headers we investigate in our work and outline their use cases.

A. Extension Architecture

Chrome extensions are lightweight add-ons that enhance the user experience on the client side by utilizing the Chrome APIs exposed by the browser. The extension architecture provides component isolation and privilege separation as part of its security mechanism. The browser isolates the installed extensions from each other and their core components from the native Web applications. Each extension provides a *manifest* where the extension developer defines the metadata of the application such as the API permissions required to perform privileged operations, the target hosts on which it should operate, and other operational configurations. It also consists of different script components that contain its core logic. If appropriately configured, their cross-origin capabilities are not governed by the Same-Origin Policy (SOP). Instead, the extensions can make authenticated fetch requests to allowed hosts and can execute JavaScript within Web applications.

The background script, often referred to as the extension core, runs in a single isolated process as a separate component. It has access to the highly privileged, feature-rich Chrome Extension APIs through which it can perform a multitude of functionalities. Some of these functionalities include accessing and deleting browser history, sending notifications to the user, periodic script execution in the background, updating browser configurations, managing the list of applications and extensions installed, and launching or uninstall them if and when desired. It can also passively intercept or actively drop, issue, or modify requests and responses for any given Web site. Hence, extensions are more powerful than regular Web applications, given the wide range of operations they can perform on the client, such as make cross-origin requests.

The content script runs as a separate instance and within the context of any given Web site. It can directly interact with the page as well as modify its contents at runtime. This makes the content script very powerful as the JavaScript executed from within the content script is indistinguishable from the original Web page's JavaScript. Thus, to prevent exploitation of Web applications that may result from abusing content scripts by malicious entities, it only has access to low-privilege and less critical browser APIs such as `i18n`, `XMLHTTPRequest`, or `storage` APIs. It can also directly communicate with the background script using the Messaging API.

To access any Chrome Extension API, browser API, or access the Web site's content that they intend to use in their components, the extension developers must declare its corresponding permissions in the manifest. Upon an extension's installation, the user has to grant all the required permission to enable the extension. While it is necessary to declare permission to use privileged APIs, it is also required to declare the host permissions on which the extension core should operate. For instance, if an extension needs to intercept requests originating from the client on `example.com`, the developer needs to declare this host in the manifest as part of *webRequest* and `example.com` as API and host permission, respectively.

Although it is recommended for the developers to request minimal permissions for their target functionality, as per the *Principle of Least Privilege*, prior research works have shown that only a few extensions adhere to this policy (e.g., [5, 23, 24]). Moreover, once the user grants the required privileges to these extensions at install time, the user does not have any control or knowledge on how and when the extension utilizes the granted privileges to carry out their functions in the background and within what context. Thus, an extension with elevated privileges can pose a severe threat to the security of the application as well as to the privacy-sensitive data processed at the client side.

B. HTTP Security Headers

HTTP security headers are a subset of standard HTTP headers that mediate security-specific information between the server and the client. The Web server sends necessary security configurations along with response headers to the browser when a user accesses any Web site. Upon receiving the response, the browser enforces the desired security features for the given site on the client. Tampering with these headers in-flight may potentially disable the desired defense mechanism, e.g., framing control, and expose an application to the associated types of attacks, e.g., click-jacking. Several header-based mechanisms have been introduced over time which intend to defend against different threat models. We focus on four widely-used, security-critical headers deployed by popular Web applications for this study, based on our observations from recent academic and non-academic studies over adoptions of various HTTP security headers such as by Buchanan et al. [6] and other following works.

1) Content-Security Policy (CSP): Although initially introduced to mitigate cross-site scripting (XSS), this header has undergone several revisions over the years to control framing and enforce secure communication channel on the client [22]. Through various CSP directives, the server controls how the browser should handle different contents for a given Web site, such as scripts, style-sheets, images, and forms. These

directives contain the allowed origins for each type of content and instruct the browser to load only those resources. For example, through `script-src https://*.foo.com`, a site allows only scripts originating from any HTTPS-enabled subdomain of `foo.com` to provide scripts to it and excludes all other sources (including inline scripts), mitigating a potential injection flaw. This primary use case was often the topic of modifications in CSP; e.g., to get rid of the dreaded `'unsafe-inline'` keyword, CSP's Level 2 introduced nonces and hashes to selectively allow the inline scripts from the developer. In Level 3, CSP added the `'strict-dynamic'` keyword to enable scripts trusted through nonces or hashes to programmatically add additional script resources, while at the same time disabling a host-based allowlist. Notably, CSP is designed in a backward compatible fashion, i.e., combining nonces with `'unsafe-inline'` will enable modern browsers to rely on nonces (i.e., they ignore `'unsafe-inline'`), whereas legacy browsers still execute the inline scripts even without supporting nonces.

Orthogonally, other directives such as `frame-ancestors` and `frame-src` limit the domain which can render the current Web page inside an *iframe* and those which the given Web page can load within an *iframe*, respectively, thus mitigating click-jacking and frame-injection attacks. CSP also contains directives to enforce TLS over an insecure connection and prevent any man-in-the-middle attack. For instance, the `block-all-mixed-content` directive explicitly instructs the browser to block all mixed content included within the Web page while declaring `upgrade-insecure-requests` directive forces the browser to upgrade all HTTP resources and links to HTTPS.

2) HTTP Strict-Transport-Security (HSTS): The Web server defines an HSTS header to ensure a secure communication channel between the client and the server and prevent any man-in-the-middle attack such as protocol downgrade attack and cookie hijacking [20]. When the browser receives this header, it blocks any redirection over HTTP and enforces communication over HTTPS connection for the given host. The browser can cache this setting for a given period, as specified by `max-age` directive (at least one year as per the recommended standards) while `includeSubDomains` directive instructs the browser to load all the subdomains for the given host over HTTPS as well.

3) X-Frame-Options: The X-Frame-Options header intends to defend against click-jacking attacks on the client side. It enables the server to restrict their Web site to be framed inside another Web site, within `<frame>`, `<iframe>`, `<embed>` or `<object>` elements, over same or different origin, as desired [21]. For instance, one can completely disallow any framing for their Web site using `DENY` attribute, irrespective of the origin of the parent page, whereas the parent page with the same origin may frame the target Web site with `SAMEORIGIN` attribute. While CSP's `frame-ancestors` is the desired way of stopping framing-based attacks, sites still deliver the X-Frame-Options header much more frequently than the better CSP-based alternative [30]. Furthermore, in the presence of `frame-ancestors`, the X-Frame-Options header is ignored by modern browsers; however, Web applications often use both, often with conflicting security guarantees [8].

4) X-Content-Type-Options: This HTTP header intends to thwart MIME-type sniffing vulnerability on the client by instructing the browser to not determine the MIME type of the content within the response and obey the values specified by the `Content-Type` header. For instance, whenever a user accesses any uploaded file for which either the server sends no `Content-Type` header or sends with inappropriate values, the browser "sniffs" the sent resources to determine its content type. This can lead to dangerous situations, e.g., when the client detects the file to be HTML or JavaScript. By setting the `nosniff` attribute for this header, the server instructs the browser to not sniff the MIME type of any resource.

III. Research Methodology

In this section, we describe our automated framework to identify potentially suspicious Chrome extensions that prevails in the Web ecosystem by utilizing a combination of static and dynamic analysis techniques. We outline the goal of this framework before discussing its technical specifications. The proposed methodology intends to answers the following fundamental questions in line with the discussed threat as follows:

- How many extensions hold the privilege to intercept or modify Web responses before the browser renders them?

- How many of the above extensions perform operations on specific hosts and how many of them target all the domains to modify the response headers?

- How many of the above extensions actively inject, drop, or overwrite HTTP security headers on respective target domains?

- What are the security headers most targeted by these extensions? Do these modifications degrade the client-side security of Web applications in the wild?

In the following parts of this section, we first explain the filtering mechanism by which we select the target extensions that hold required privileges and build our dataset for further analysis. We then outline the process by which we collect the target security headers in the presence of each of the Chrome extensions and the ground truth headers. In the end, we then discuss the steps to analyze and determine those extensions that are suspicious and may prove to be a danger to the security of the Web application. Figure 1 depicts an overview of the proposed framework. In (i), we demonstrate the steps followed during static analysis of the codebase and initial screening. While (ii) and (iii) describe the process of header interception and comparative analysis at runtime.

A. Dataset Construction

The framework leverages a static flow-analysis approach to detect those extensions that intercept responses before the client renders them. An extension is required to hold the *webRequest* and optionally the *webRequestBlocking* permission in order to intercept and/or modify requests and responses at runtime. While *webRequest* allows the extension to intercept requests and responses for the target hosts asynchronously, *webRequestBlocking* allows them to handle
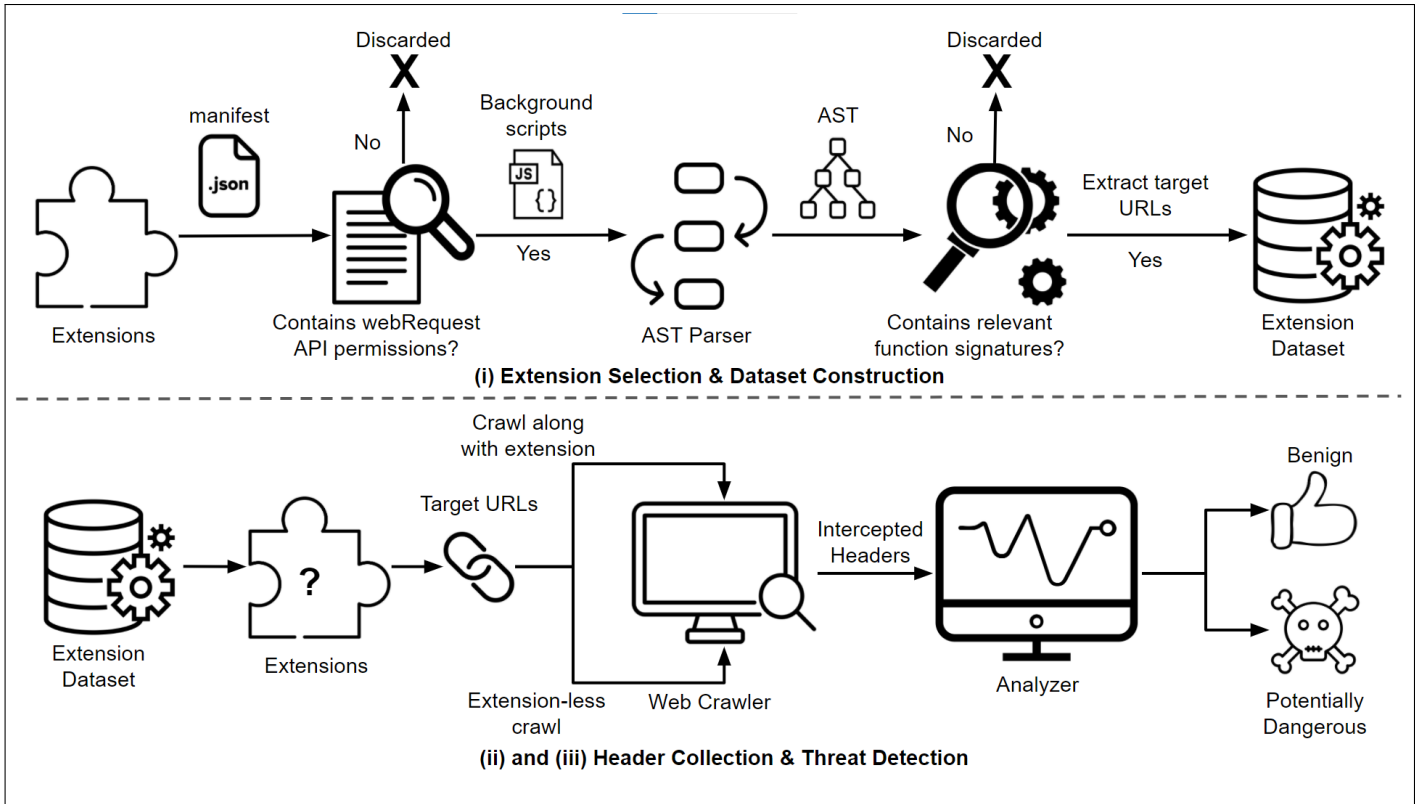
Fig. 1: An overview of the automated framework for analysis

```
var oneMoreDomain = "*://*.bar.com/";

chrome.webRequest.onBeforeSendHeaders.addListener(
    function(details) {
        //core logic
        return {requestHeaders: details.requestHeaders};
    },
    {urls: ["*://*.foo.com", "https://*/*",
    ↪    oneMoreDomain]},
    ["blocking", "requestHeaders"]
);
```

Listing 1: Host Permissions to intercept request passed along with the event listener.

the requests synchronously, and the control-flow returns only after the execution of the callback method [18]. Thus, we proceed ahead with our framework as follows: (i) We first shortlist all the extensions from the downloaded 186K extensions which request the above-stated permission(s), declared in their *manifest*. (ii) We then combine all background scripts into a single file and pass it to our AST parser. (iii) We then check for the existence of `*.webRequest.*` (e.g., `chrome.webRequest.onCompleted.*`) methods within the parsed AST and select the extensions for the next step. (iv) In the last step, the framework again iterates over the ASTs of the shortlisted extensions and, in particular, looks for the method signatures used to intercept the responses right before the browser renders them, i.e., `*.webRequest.responseReceived.*` [18]. In the last step above, the framework follows the function-call chain and

return values and traces back to all the function initialization and definitions to locate the desired function signature effectively. To implement the above-mentioned steps, the framework utilizes the error-tolerant version of Acorn, i.e., *acorn-loose*, an open-source NodeJS library [1], to parse the ASTs of the background script(s), detect the function signature and extract URL literals. The shortlisted extensions obtained from the last step constitute our candidates for further analysis.

The framework now extracts the target domains for each of these shortlisted extensions at this stage. There are multiple ways by which an extension can declare its target domain. *(i)* The first obvious way is to list all the target domains in the *manifest* for host permissions. These domains may also contain wildcards to allow operations on different schemes, paths, and their sub-paths or subdomains of the given domain, for instance. *(ii)* Many extensions are domain-agnostic and operate on all the domains and thus, declare *all_urls* [16] as host permission. *(iii)* Another way to achieve either of the above two steps is by utilizing the *activeTab* privilege when the user manually invokes the extensions' functionality on any target domain, loaded on the currently active tab [15]. It allows intercepting requests from the active tab until it terminates. *(iv)* Many extensions declare *activeTab* or *all_urls* in their manifest yet operate only on specific domains, by sending hosts permissions along with the Chrome API to intercept to requests/responses or perform domain checks in their core logic to carry out operations, as demonstrated in Listing 1.

The framework considers the above four ways of host permissions and extracts the domains for further analysis as

4

follows. First, it extracts the hosts and wildcard domains from the manifest, if there exists any. Depending upon the nature of the wildcard, it either extracts the top-level domain from the wildcard or terms it as *all_urls*. For example, it interprets `http://*/*/*` or `*://*/*` as wildcards for top-level domains and thus, consider them in *all_urls*. Whereas, it extracts `example.com` from `https://*.example.com` as specific target host. Besides, it also checks for the permissions as in (ii) and (iii) above to accurately determine the Chrome extension's operational space. Though the framework also extracts the literals from the AST of the extension background script and perform contextual checks for all URL-like strings to determine hosts, it does not prove to be full-proof effective due to client-side obfuscations, use of hidden or cached values, and data fetched at runtime. We further segregate the filtered extensions into two pools based on their operational space, i.e., extensions that operate on all URLs while those that operate only on specific URLs.

B. Header Collection

We saw in Section III-A that the framework filters out unwanted extensions while it considers and stores those for the next stage of analysis, which show any signs of header interception and manipulation on the client side, using static analysis of the codebase. To accurately determine the extensions that tamper with the security headers and recognize the target security headers, we now leverage dynamic flow-analysis and incorporate it into our framework. For each extension, the framework extracts its target URLs and visit each of them twice: The first crawl provides the ground truth for further analysis by intercepting all the security headers without loading the extensions on the browser. While in the second run, the crawler loads the Chrome extension and allows the framework to collect the security headers, which may or may not be manipulated by the target extension. For those extensions that target all the URLs (using *all_urls* or wildcards) or has access to any active tabs, as discussed before, the crawler visits the Tranco Top 100 domains [29] as their target hosts and collect the headers, along with any other specified or extracted URLs from its data. The framework gathers all these headers and stores them for further comparative analysis.

To incorporate the above crawling feature, we use puppeteer, an open-source NodeJS library, to load browsing extensions on-the-fly and launch headless Chromium instances [17]. In addition to puppeteer, the crawling component also utilizes the Chrome DevTools Protocol to intercept all the response headers after the target extension makes all the modifications at the client side [13]. In particular, we use `Network.responseReceived` method to intercept the responses. The crawler launches the browser for every extension individually, with and without loading them, visiting each of the URLs successively. It stays on the page for five seconds after the page load event triggers to allow and record the client-server communication. It records the request initiator, the request and response URL, the resource type, the frame of the originating request, and other important metadata along with the actual headers to distinguish between the requests and responses, and subsequently, their respective headers. After every page visit and its subsequent timeout, the crawler clears the cache before the next URL is visited. It then dispatches the data to the remote server for analysis after collecting the

```
var targetHeader = "content-security-policy:
↪   script-src 'self', *.xx.fbcdn.com,
↪   *.googleapis.com, *.attacker.com;";

var groundTruth = "content-security-policy:
↪   script-src *.fbcdn.com, *.googleapis.com;";

var difference = headerDiff(targetHeader,
↪   groundTruth);

//difference = ["self", "attacker.com"]
```

Listing 2: An example of header comparison based on their domains.

headers for each of the given URL, where it further stores the data into the database.

C. Identifying Suspicious Extensions

At this stage, we have the target security headers modified by the extensions on respective domains along with the corresponding ground truth headers to compare against, available for analysis. In the next and final step, the framework retrieves all the headers: both ground-truth and the modified security headers in the presence of extension, for each of them gradually. It then performs a comprehensive comparative analysis on the two sets to determine the nature of modifications and whether it exhibits benign or suspicious behavior, based on its subsequent impact on the target application.

More specifically, the framework retrieves the two set of headers for an extension and separately groups them based on a tuple of their {*requestInitiator*, *requestURL*, *responseURL*, *resourceType*, *responseStatus*}. By doing this, it ensures that the requests, as well as the headers collected from them, are distinguishable and comparable against their ground-truth candidates. It then parses the header values for all the grouped-data from both sets into JSON serialized format and further compares one set of headers with its counterpart to record the differences. In case the server sends multiple instances of the same header within a response, the framework groups all the distinct values for the given header in a serialized structure and compare accordingly. For instance, in the case of X-Frame-Options header, when an extension tries to modify the header value to enforce a relaxed version of the policy desired by the Web server, e.g., replace `DENY` with `SAMEORIGIN` or the empty string, the framework identifies the change and report both the original as well as the extension-desired policy accordingly. In case there is any security header for a given request, intercepted along with the presence of an extension and there is no counterpart in the ground-truth dataset, the analyzer labels the header as "injected" by the extension. On the contrary, if any ground-truth security header is missing in the header set intercepted along with the extension, it marks them as "stripped" by the extension.

The analyzer performs an additional check between the injected and stripped headers to rule out any correlation. Moreover, it implements an extended top-level domain-based comparison for the security headers that allow URLs with a wildcard set as their values. For instance, the `script-src` attribute allows wildcards as well as specific URLs. As demonstrated in Listing 2, the framework compares the top-level

Category	# Extensions
Extensions analyzed in this phase	186,434
Extensions with target API privileges	17,434
Extensions with relevant function signatures	3,286
Extensions targeting <all_urls> & wildcards	2,694
Extensions targeting specific hosts	592
No. of distinct domains extracted	4,550

TABLE I: Summary of the findings from static analysis

Category	# Extensions
Extensions analyzed in this phase	3,286
No. of distinct domains crawled on	4,637
Extensions that modifies at least one security header	2,485
Extensions that modifies all four security headers	553
Extensions that targets <all_urls> & wildcards	2,245
Extensions that targets specific hosts	240

TABLE II: Summary of the findings from runtime analysis

domain of all the URLs and wildcards and report those that do not exist in the ground truth headers. This is helpful in particular for CSP as the framework avoids comparing nonces and only identifies the differences among allowed domains and other associated attributes. This way, it intends to reduce the risk of false-positives and accurately determine the nature of modification by the extensions. Once the framework identifies all the modified, injected, and dropped headers, it then reports whether an extension exhibits suspicious activity or is benign if it performs any of the above operations on the target security headers.

IV. EVALUATION

In this section, we evaluate the proposed methodology by analyzing over 186K Chrome extensions that were publicly available on the Chrome Web Store between January and June 2020. The framework crawls the Tranco Top 100 domains, as on 15th September 2020 and other target hosts, depending on their corresponding host permissions, to scrutinize the behavior of individual extensions. It intercepts all the target domains' responses and from all the frames embedded in the parent domain. We report our findings from each of the steps during analysis in the following subsections.

A. Permission & Source Code Analysis

The framework begins by extracting the *crx* packages for a total of 186,434 downloaded extensions and then checks for the API and host permissions declared in the manifest. It identifies a total of 17,434 extensions that requests for either *webRequest* or *webRequestBlocking* permission, necessary to successfully intercept and modify header at the client and filters out the rest in the first round. It then considers these selected extensions for the next stage of analysis.

For the above extensions, the static analyzer component parses the manifest to identify the background script(s) and subsequently builds a single unified AST using them. Now, the analyzer traverses through the AST twice: At first run, it collects all possible forms of function declarations in JavaScript, which could be used as a callback argument to the target APIs, and in the second run, it detects the APIs as well their callback method along with their target hosts, if they are passed as an argument, by utilizing the information obtained from the first run. Moreover, it also extracts the host permissions and URLs for those extensions from the manifest where the relevant function signatures are detected. Thus, it identifies a total of 3,286 extensions, which show definite evidence of header interception and manipulation at the code level. It also extracts 4,550 distinct URLs from these extensions' host permissions

other than the Tranco top 100 domains. It further categorizes the extensions based on their target hosts, i.e., whether they operate on all hosts or a particular set of the host(s). We summarize the findings from this phase of analysis in Table I. As listed in the table, we observe that most extensions operate on a large number of domains. We also manually verify a few of these extensions before storing them for the next phase of scrutiny.

B. Runtime Analysis

Table II outlines the statistics reported from the second and final stage of checking. The crawling framework visits the corresponding target hosts twice for each of the extensions identified in the initial phase simultaneously and intercepts the headers to detect the extension's modifications. It crawls a total of 4,637 distinct URLs that constitute the Tranco Top 100 domains and 4,550 URLs extracted in the preceding phase. It records the target header and their respective values with and without the extension loaded into the browser for all the frames within a domain. We observe that approximately 76% of all the extensions (i.e., 2,485) analyzed at this stage tamper with at least one security header considered for this study. Moreover, approx. 17% (i.e., 553) among the above 2,485 extensions alter the values of all the four headers. Further, when classifying them based on their target hosts, we find that approx. 90% (2,245) among the above-reported findings operate on all URLs or wildcards while the rest of them only target specific hosts. We realize that many extensions that operate on specific hosts may execute their target functionality on a particular path or require user interaction at runtime, thus possibly resulting in a lower rate of detection. On the other hand, when we scrutinize those extensions which target all URLs, a majority of them operate on a small number of domains, or rather URLs, among the top 100 domains, that provide auxiliary services such as ad contents, analytics services, third-party tools, and other resources, even though it requests for *all_urls* or wildcards as host permissions.

C. Results & Observations

As mentioned in Sections IV-A and IV-B, we report that a total of 2,485 extensions could potentially undermine the security of applications based on their privileges and suspicious nature of security-critical header manipulation at the client side. We further investigate the overall frequency of manipulation among different security headers among the above-selected extensions from the collected crawling data. It helps us assess the risks associated with each header and modification of their values and identify the manipulation pattern by these extensions in the wild. We report our findings for each of these target headers

Target Security Headers	Extensions Involved in		
	Alteration	Injection	Dropping
Content-Security-Policy	1,412	-	509
Content-Security-Policy-Report-Only	1,436	-	1,436
Strict-Transport-Security	1,338	844	679
X-Frame-Options	1,345	811	676
X-Content-Type-Options	1,115	707	553

TABLE III: Different security headers & their instances of manipulation by extensions

in Table III. While the alterations represent overall figures for all possible ways of modification by an extension for each header, injections and dropping represent specific instances of these actions, respectively. For instance, among 1,412 extensions that alter CSP headers in any fashion, 509 drop them from the header altogether. We conclude from the statistics in Table II and III that a significant number of extensions conveniently drop all the security headers sent by the server to execute their seemingly benign functionalities at the client side. We observe that the `Content-Security-Policy` (CSP) and the `X-Frame-Options` (X-FO) headers are most likely to be modified by the extensions and more often together. For the sake of completeness, we also monitor the `Content-Security-Policy-Report-Only` header (CSP-RO) along with the standard CSP header, which is deployed as a more relaxed and preliminary policy when compared to the standard CSP to report the violations to the server rather than enforcing them at runtime. Specifically, for CSP-RO, we see that all the reported alterations correspond to dropped instances. We observe a similar trend among CSP headers containing the `'report-sample'` directive. This property instructs the browser to send the first 40 characters of the object that violates the stated policy to the server within a violation report. While over 591 domains sent this property, we find that 490 extensions among 2,709 of them strip this property from the original value and over ten distinct domains.

We observe a mixed trend while analyzing the changes made by the reported extensions over X-FO headers. While many extensions arbitrarily set the header values to `SAMEORIGIN` or even `DENY` at times that may cause side-effects due to a more restrictive policy, other extensions set it to an empty strings. This leads to a relaxed version of security enforcement at the client, undesired by the server, as Chrome does not accept an empty string as a valid value. We see a similar pattern among `X-Content-Type-Options` (X-CTO) headers where half of the reported extensions inject the header with `nosniff` value while the rest of them drop this header. For HSTS headers, we see that there exist inconsistencies between their values originally sent by the server and their modified counterparts. Many extensions often, while modifying the header value, miss out on `includeSubDomains` originally sent by the server and only specify the `max-age` value. Few extensions also assign the `max-age` to zero even though the original value is non-zero. While many extensions continue to drop these headers, an equivalent number of them also injects them to potentially strengthen the application's security. However, at the same time, this may often result in

security- or compatibility-based side-effects.

There exist some discrepancies from the server side as well, particularly for the HSTS and X-FO headers, that cause conflict during our analysis. The Web servers often send the same security header but multiple times and with distinct values to the client, which may be interpreted differently by different browser vendors, as described in detail by Calzavara et al. [8, 9]. We observe similar conflicting cases when the extensions encounter such situations, even though it intends to modify values for more vigorous security enforcement. The security headers and their values differ even with different user-agents, often specified by the seemingly privacy-aware extensions while sending requests. Given the fact that these extensions do tamper with the security headers having the privilege to do so, we still find them suspicious and report them in this study.

D. Deep Dive into CSP

Based on the discovery from previous work by Hausknecht et al. [19] and our findings in Table III, we infer that extensions often modify the CSP headers to bypass the security policies that they dictate at the client. We find that among all the 2,051 Chrome extensions that alter any of the two variants of the CSP (i.e., CSP and CSP-RO) and on 104 distinct domains, 1,604 of them drop these headers altogether on 87 unique domains. These figures highlight that extensions often find CSP as the biggest roadblock to perform their core functionalities at the client and ultimately resort to dropping them from the response headers. We now perform an in-depth investigation over the modifications, injections, and drops among individual directives and policies within those CSP headers, which are only modified by the extensions.

We categorize the alterations among these directives and their values within a modified CSP header based on the following definitions: An instance is deemed to be modified when an extension modifies the existing header such that it only alters the values associated with that directive by selectively adding or dropping any allowed sources whereas the directive remains intact within the existing header. A dropped instance is one that is selectively removed along with its values (or entirely replaced as the empty string) by the extension without affecting any other directives within the header. Lastly, an injected instance refers to a situation when an extension specifically adds a particular directive along with its associated values in an existing CSP header. While it may seem counter-intuitive that extensions could threaten a CSP's security by *adding* a directive, it must be noted that many directives in CSP fall back to `default-src`. That is, a strict `default-src` may be undermined by injecting a lax `script-src`, as resources are always governed by the most specific directive, which does not inherit the values from the fallback directive.

For our analysis, we parse the CSP headers into JSON documents and consider these directives as keys and their specified origins as a list of values. While analyzing injections and drops, we only focus on the existence of keys, and the change in values indicates a modified directive.

Table IV summarizes our findings for various CSP directives that the reported extensions alter, injects, or drops within the modified CSP (and CSP-RO) headers. We identify

CSP Directives	Domains that send them	Modified Instances	Dropped Instances	Injected Instances	Extensions Involved
default-src	1,123	40	9	-	1,110
script-src	1,656	35	23	-	1,123
style-src	1,027	27	19	-	1,114
connect-src	971	24	18	-	223
img-src	334	7	11	-	981
media-src	230	7	9	-	979
worker-src	541	5	11	-	18
frame-src	251	2	13	5	981
frame-ancestors	880	8	13	5	980
font-src	262	12	11	-	13
child-src	101	1	8	-	13
object-src	1,322	5	18	6	981
form-action	58	1	5	-	3
manifest-src	60	-	10	4	974
report-uri	1,938	-	25	-	676
report-to	1,026	-	2	-	2
base-uri	660	-	12	-	10
upgrade-insecure-requests	1,676	-	11	-	3
block-all-mixed-content	802	-	9	-	4

TABLE IV: Different CSP directives targeted by extensions

the distinct domains that send these directives among the crawling dataset and then report those domains as instances where we observe any alterations as defined above among the modified CSP headers. Since the domains and extensions are representative of the alteration among individual directives, we understand that there is significant overlap among the number of extensions that alter more than one directive at a time. For instance, an extension that drops `frame-src` and `frame-ancestors` for a given domain, is listed separately for each directive. It follows in the case of domains as well.

Extensions often alter these directives and their values to bypass the browser's content restriction policies, as evident from the number of modified and dropped instances by 1.1K unique Chrome extensions and over 40 distinct domains. We see that they often inject scripts, images, style-sheets, fonts, and other third-party resources that may come from illegitimate sources and pose a significant risk to the application and expose them to dangerous client-side vulnerabilities, such as cross-site scripting (XSS). For instance, we observe that they add wildcards (*), `unsafe-inline` and other origins such as `google-analytics.com` within the existing `script-src`. While the latter might not seem like a security issue, any Google Tag Manager scripts (which can be arbitrary code uploaded by an attacker) are accessible through https://www.google-analytics.com/gtm/js?id=XXX, making any CSP protection entirely useless.

Such behavior is often accompanied by the manipulations within the value of `default-src` attribute, which serves as a fallback directive for some directives such as `script-src`

and `frame-src` in case the server does not define them, or the extension drops them from the header. Another popular directive among extensions is `connect-src` that controls and whitelists sources for fetch requests, XMLHttpRequests and web-socket connections that originate from the client. We observe that extensions often inject third-party domains and extension identifiers as whitelisted sources among these attribute values to communicate with these domains in the context of the target webpage.

Since CSP also allows the server to control framing at the client side, extensions often interfere with the `frame-ancestors` directives, as evident in Table IV, by either entirely dropping them or injecting unwanted third-party domains as whitelist sources, along with the existing values. While at the same time, they also tamper with the `frame-src` (and `child-src`) directives to enable loading any target third-party domain in an *iframe* from within the given Web page. A similar trend follows in the case of `object-src` and `font-src` that allows loading plugins and fonts from allowed sources, respectively. We observe that the extensions often resort to altering these attributes to inject frames with analytics and ad-serving domains. Extensions may potentially exploit this behavior in collusion with attacker-controlled hosts to not only fingerprint the user activity on the Web but further allows stealing privacy-sensitive information from the client. A specific case where we find such modifications is on `github.com` where an extension injects third-party origins such as `www.youtube.com`, `player.vimeo.com`, `checkout.paypal.com` and `https://*.addtoany.com` within the `frame-src` attribute originally not sent by the server. It allows an extension or any other client-side adversary to inject frames pointing to the above allowed origins, undesired by the server (i.e. `github.com`).

The `report-uri` attribute, used to report the policy violations that occur at the client side to the specified server endpoints, is also manipulated by several extensions. It appears obvious given that these extensions attempt to inject some code of their own (which is why they manipulate the CSP in the first place) and do not want to have possible CSP violations reported accidentally. However, the newly proposed substitute `report-to` attribute is still unpopular and is contrasting to the findings as for its predecessor. While the later revisions in the CSP standards introduced directives, such as `upgrade-insecure-requests`, and enabled the server to enforce TLS over communication channels that originate from the client, we see that only 4 extensions alter these configurations. We find that while almost all the analyzed extensions encounter these attributes over 1,676 distinct domains, only 3 extensions drop it and over 11 discrete domains. We observe similar trends for the `block-all-mixed-content` attribute, though it is recently deprecated.

E. Risks Associated Manipulation of Other Security Headers

As also highlighted in Section IV-C, many extensions misconfigure the security policies of multi-valued headers, such as HSTS and X-FO, while injecting or redefining values for them. In particular, for HSTS, we see that extensions define arbitrary `max-age` values irrespective of the original value sent by the server, often setting it to 0. This modification has an interesting side-effect. For HSTS, once a browser has observed

an HSTS header, it will refuse to connect to the given domain via HTTP until the max-age value has expired. This condition also holds if any other visited pages on the same origin do not send the header. However, by explicitly setting max-age to 0, the browser is instructed to drop the HSTS pin, effectively disabling the protection against SSL stripping attacks.

Extensions also drop other pertinent attributes, such as `includeSubDomains`, specified by the server that allows an attacker to exploit the communication channel between the client and the server upon the users' first visit or by controlling sub-domains for a given parent domain, respectively. It is crucial to correctly configure the policies for HSTS as disabling them, incorrectly defining them, or dropping them would expose the application to vulnerabilities such as protocol-downgrade attacks exploited by the *man-in-the-middle* entities in the wild. In the case of CSP, any form of alteration among values for different directives, as we saw in Section IV-D, may lead to potential degradation of the client-side security required by the server.

In the X-FO header case, many extensions specify an empty string, which often leads to the enforcement of browser-defined fail-safe defaults at the client. Chromium considers this value as invalid and fails *insecurely*, i.e., allows framing from any origin. Thus, it exposes the application to click-jacking vulnerabilities at the client. This behavior is often in conjunction with modifying or dropping the `frame-ancestors` directive within CSP, which is the replacement for the long-since deprecated X-FO header [8]. We also observe "security-aware" extensions arbitrarily defining more-restrictive policies (`"DENY"`) for framing-control that may lead to unwanted side-effects such as inconsistencies and incompatibility among legacy applications. Extensions often drop the X-CTO header from the responses that allow the browsers to perform MIME-type sniffing on the sent resources. An active web attacker may exploit this behavior by injecting malicious JavaScript within the benign content and perform nefarious operations, such as XSS and XSSI, although other mitigation strategies, such as CSP, are well in place.

V. IMPROVEMENTS AND FUTURE WORK

Although the proposed automated framework leverages an amalgam of static and runtime monitoring to precisely identify potentially harmful extensions, we recognize cases where it fails to detect many extensions during analysis relevant to our study due to several limitations in respective phases. While statically scrutinizing the scripts, the analyzer track-backs to all the function initialization, declarations, and definitions along with the target URLs to identify the target event listeners, their callback methods, and the target hosts on which they operate and execute `webRequest` APIs. We observe that many extensions fetch their core logic at runtime from the remote server and either only execute them or, further, store them at the client-side storage for reuse. In either of these cases, our static analyzer fails to identify the use of target function signatures within them. It also fails to analyze over obfuscated scripts and do not extract target URLs if they are defined in variables. Thus, we emphasize that the findings from our framework, as discussed in Section IV, represent an underestimate of the suspicious behavior prevalent among these extensions. While we understand that there are generic

limitations to static analysis, we intend to use runtime monitoring for extensions that describe the above-stated behavior within the codebase in our future work. Moreover, the static analyzer could be made more robust to handle the minified script and extract pertinent literals.

On the other hand, many extensions operate on subdomains, specific paths, and their sub-path to execute their potentially devious operations, which may be behind the login or would require users' intervention at runtime. For instance, the execution of core extension logic by utilizing *activeTab* permission would require users' consent for each target host. Since the runtime analyzer only visits the top-level domains and any other specific URL, if there exists any, it would fail to replicate required user behavior at the client and thus, cannot trigger all the desired target functionalities. Moreover, for extensions with *all_urls* permission or wildcards URLs, the framework only crawls on the Tranco top 100 domains, which severely limits its coverage and detection rate. When visiting target domains with and without extensions at runtime, we observe that the servers often send different security headers on consequent visits or with varying user-agents. This behavior severely impacts our findings as it may lead to a high rate of false-positives and label benign extensions as suspicious. While replicating user behavior at runtime is an apparent limitation to dynamic analysis, we believe that collecting a range of ground truth headers over a small time-interval and with different user-agent configurations may help tackle the randomization among security headers and help overcome the above-stated challenges during analysis.

While we analyze the impact of modifications among the four most popularly deployed security headers, it is imperative to quantify the risks of other pertinent headers as well, such as `Referrer-Policy` that permit Web servers to control referrer information sent along with the request, or `CORS` headers which regulate resource-sharing across different origins. If injected by an extension, this could potentially allow for access to cross-domain resources, which would usually be protected through the Same-Origin Policy [11]. Hence, we aim to extend our framework to cover the rest of the security headers in our future line of work. Lastly, the framework analyzes the extensions individually to identify any suspicious behavior. As much as it helps to discover the characteristics of every extension separately, it does not provide a view on the side-effects of communication-channel and potential collusion that may exist between multiple extensions installed at the client. Picazo-Sanchez et al. [28] further showed that the browser decides the execution order of each extension based on their registered event listeners and installation timestamp, which may cause unintended security issues at the client. Since it is infeasible to analyze the collusion pattern among every combination of extensions, we aim to identify those that are *externally-connectable* and clusters based on their functionalities and installations to detect any potential collusion pattern arising from the union of their respective privileges.

VI. RELATED WORK

This section briefly outlines the previous research contributions related to our work and highlights the significant differences between this work and other closely related studies.

A. Malicious Browser Extensions

The scientific community has ventured significant time and effort in the past to protect against the threats that these browser extensions may pose to the security of the Web applications and the privacy of the user information that it mediates. Thomas et al. [31] and Xing et al. [33] individually conducted a longitudinal study and observed that a large number of extensions inject unwanted ads from illegitimate sources into the webpage for monetary purposes. Perrotta and Hao [27] and DeKoven et al. [14] further affirmed in their respective works that these extensions are often abused to install malware at the client and may often serve as an integral component of the botnet framework over the Internet, owing to the over-privileged capabilities that they enjoy on the client side. Another line of work by the authors in [2, 25, 32] report that the extensions often observe the users' activity on the Web and may steal sensitive information and send it to a third-party domain to create unique user profiles. While these studies demonstrate that the adversaries abuse extensions as an attack vector on the Web for various nefarious purposes, we explicitly focus on a specific threat in this work, i.e., extensions that manipulate security headers on the client side.

B. Extensions & Security Headers

Bauer et al. [5] statically examined the privileges held by the top 1000 Chrome extensions and highlighted that they could be abused as a potential attack vector to carry out a multitude of attacks at the client. They emphasized that extensions often manipulate security headers within the response to carry out intended functionality and may potentially collude with other extensions installed at the client to escalate their effective privilege and perform undesired operations. At the same time, Kapravelos et al. [24], conducted a large-scale study over 48K Chrome extensions to identify malicious extensions in the wild using dynamic analysis and *HoneyPages* for maximum functionality coverage. They also observe that extensions are involved in security header alterations at the client. We identify these two studies as the most closely related ones with our work. Chen and Kapravelos [12] used the taint-propagation technique to detect those extensions which leak privacy-sensitive data of the user over 180K Chrome extensions. Further, Buyukkayhan et al. [7] deployed a multi-stage static analysis technique to determine extension-reuse vulnerability, a specific class of attack among Firefox extensions.

Although we observe that the above works provide sufficient evidence that these *man-in-the-browser* entities manipulate security headers before the client renders the response, they do not provide an in-depth analysis of this behavior due to the scope of their work. To close this research gap, we take inspiration from these studies while designing our framework and further focus on the following key points: *(i)* Large-scale analysis of over 186K Chrome extensions, available on the Chrome Web Store, to date. *(ii)* Determining and quantifying the target security headers that extensions target most often. *(iii)* The risk associated with manipulating these headers and their respective attributes. *(iv)* Different classes of extensions and their interception and modification pattern, based on their functionality.

C. Security Architecture of Extensions

In the early days of Chrome extensions, seminal works by Barth et al. [4] and Carlini et al. [10], respectively, argued for fine-grained privilege separation, a permission system based on the principle of least privilege and component isolation between the extensions and the native Web applications within the browser ecosystem. However, the proposed mechanisms have proved inadequate since the researchers have continually reported that these browsers often indulge in malicious activities on the Web. In their work, Bandhakavi et al. [3] proposed an automated vetting tool to statically analyze the extensions' source code to identify potential vulnerabilities or dangers that it may pose before making them available to the public. Jagpal et al. [23] further proposed a fine-grained screening process that combines both static and dynamic analysis techniques to distinguish rogue extensions from benign ones. However, we observe that it is still possible for an attacker to hide malicious operations of an extension behind an innocuous functionality, trigger them at certain events or dynamically load the code at runtime and thus, could be successfully uploaded on the store, bypassing the security measures in place. Recently, Pantelaios et al. [26] also showed that an extension could initially constitute benign functionalities and can later turn malicious by receiving updates, thus bypassing the initial screening. Hence, it is imperative to have an added line-of-defense at runtime to monitor and defend against such a specific class of attacks.

VII. Conclusion

In this paper, we conducted a large-scale analysis with over 186K Chrome extensions to detect those extensions that modify the HTTP security headers in the server's response and the threats associated with it at the client. We proposed an automated framework that analyzes this behavior among extensions and further label it as either benign or suspicious. The framework first statically analyzes the permissions requested by individual extensions and their source code to filter out the irrelevant ones. Further, it crawls on the Tranco top 100 domains along with other target URLs to detect the header manipulations at runtime for each of these extensions. We find that approximately 76% of the 3,286 Chrome extensions, which requests for necessary permissions and can also intercept response headers, modifies at least one of the four widely deployed security-critical headers on the Web while 553 of them target all five target headers. We observe that extensions tend to target `X-Frame-Options` and `Content-Security-Policy` headers more than others to introduce additional seemingly benign functionalities on the visited webpage. Upon extended analysis on the modification pattern among CSP, we observe that extensions intend to bypass the content-restrictions and framing-control, initially enforced by the server, by modifying corresponding directives within the header. We assess the risk of modification of these security primitives and understand that tampering with them can disable critical security configurations at the client, exposing the application to nefarious entities on the Web.

REFERENCES

[1] Acorn. Acorn, 2020. URL https://github.com/acornjs/acorn.

[2] Anupama Aggarwal, Bimal Viswanath, Liang Zhang, Saravana Kumar, Ayush Shah, and Ponnurangam Kumaraguru. I spy with my little eye: Analysis and detection of spying browser extensions. In *IEEE EuroS&P*, 2018.

[3] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, 2010.

[4] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. *NDSS*, 2010.

[5] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, and Yuan Tian. Analyzing the dangers posed by chrome extensions. In *2014 IEEE Conference on Communications and Network Security*. IEEE, 2014.

[6] William J Buchanan, Scott Helme, and Alan Woodward. Analysis of the adoption of security headers in http. *IET Information Security*, 2017.

[7] Ahmet Salih Buyukkayhan, Kaan Onarlioglu, William K Robertson, and Engin Kirda. Crossfire: An analysis of firefox extension-reuse vulnerabilities. In *NDSS*, 2016.

[8] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: A formal analysis of inconsistent click-jacking protection on the web. In *USENIX Security Symposium*, 2020.

[9] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. Reining in the web's inconsistencies with site policy. In *NDSS*, 2021.

[10] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, 2012.

[11] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still don't have secure cross-domain requests: an empirical study of CORS. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[12] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *CCS*, 2018.

[13] Google Developers Community. Chrome DevTools Protocol, 2020. URL https://chromedevtools.github.io/devtools-protocol.

[14] Louis F. DeKoven, Stefan Savage, Geoffrey M. Voelker, and Nektarios Leontiadis. Malicious browser extensions at scale: Bridging the observability gap between web site and browser. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*. USENIX Association, 2017.

[15] Google Developers Community. chrome.activeTab, 2020. URL https://developer.chrome.com/extensions/activeTab.

[16] Google Developers Community. Match Patterns, 2020. URL https://developer.chrome.com/extensions/match_patterns.

[17] Google Developers Community. Puppeteer, 2020. URL https://developers.google.com/web/tools/puppeteer.

[18] Google Developers Community. webRequest, 2020. URL https://developer.chrome.com/extensions/webRequest.

[19] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. May i?-content security policy endorsement for browser extensions. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015.

[20] IETF. HTTP Strict Transport Security (HSTS), 2012. URL https://tools.ietf.org/html/rfc6797.

[21] IETF. HTTP Header Field X-Frame-Options, 2013. URL https://tools.ietf.org/rfc/rfc7034.

[22] IETF. Initial Assignment for the Content Security Policy Directives Registry, 2016. URL https://tools.ietf.org/html/rfc7762.

[23] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX Security Symposium*, 2015.

[24] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security Symposium*, 2014.

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

[26] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *CCS*, 2020.

[27] Raffaello Perrotta and Feng Hao. Botnet in the browser: Understanding threats caused by malicious browser extensions. *IEEE security & Privacy*, 2018.

[28] Pablo Picazo-Sanchez, Juan Tapiador, and Gerardo Schneider. After you, please: browser extensions order attacks and countermeasures. *International Journal of Information Security*, 2019.

[29] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *NDSS*, 2019.

[30] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. 2020.

[31] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015.

[32] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *USENIX Security Symposium*, 2019.

[33] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15. International World Wide Web Conferences Steering Committee, 2015.