DeDup.js: Discovering Malicious and Vulnerable Extensions by Detecting Duplication

Pablo Picazo-Sanchez[a], Maximilian Algehed[b] and Andrei Sabelfeld[c]
Chalmers University of Technology, Gothenburg, Sweden

Keywords: Browser Extensions, Web Security, Web Privacy.

Abstract: Browser extensions are popular web applications that users install in modern browsers to enrich the user experience on the web. It is common for browser extensions to include static resources in the form of HTML, CSS, fonts, images, and JavaScript libraries. Unfortunately, the state of the art is that each extension ships its own version of a given resource. This paper presents DeDup.js, a framework that incorporates similarity analysis for achieving two goals: detecting potentially malicious extensions during the approval process, and given an extension as input, DeDup.js discovers similar extensions. We downloaded three snapshots of the Google Chrome Web Store during one year totaling more than 422k browser extensions and conclude that over 50% of the static resources are shared among the extensions. By implementing an instance of DeDup.js, we detect more than 7k extensions that should not have been published and were later deleted. Also, we discover more than 1k malicious extensions still online that send user's queries to external servers without the user's knowledge. Finally, we show the potential of DeDup.js by analyzing a set extensions part of CacheFlow, a recently discovered attack. We detect 53 malicious extensions of which 36 Google has already taken down and the rest are investigated.

1 INTRODUCTION

Browser extensions are small web applications that modern browsers allow users to install for enriching their experience when surfing the Web.Chrome's Web Store, the repository where Chrome's extensions are installed from, have over 188k extensions with more than 1B downloads (Extension Monitor, 2021).

Browser extensions have a compulsory file in browser extensions called the *manifest*. Additionally, extensions can include as many static files as needed, e.g., HTML, CSS, JavaScript, fonts, and images. Developers upload the extensions to vendors' repositories to be stored, checked, and publicly distributed.

Browser Extension Distribution. In Chrome, Google only allows users to install extensions from Google's Web Store. When a user selects the extension to be installed, Chrome downloads, unpacks, and installs it automatically. If the user installs another extension with similar files, they are duplicated in the file system.

[a] https://orcid.org/0000-0002-0303-3858
[b] https://orcid.org/0000-0002-1666-9994
[c] https://orcid.org/0000-0001-9344-9058

Code Duplication in Browser Extensions. Reusing pieces of source code is a common practice in software development (Gabel and Su, 2010), with 93% of the JavaScript code on GitHub cloned (Lopes et al., 2017). However, massive code reuse is also a major security threat as such cloning propagates bugs and vulnerabilities (Roy et al., 2009).

This paper puts a spotlight on the problem of code duplication in browser extensions. We propose DeDup.js, a novel approach that enables us to discover malicious as well as benign-but-vulnerable extensions by leveraging deduplication.

DeDup.js: Hardening Approval Process. When developers publish their extensions in the Web Store, these go through an automatic review and, in some cases, some manual checks (Google, 2021a; Dev.Opera, 2021). If during this process, vendors mark the extension as *malicious* (Google, 2021e) or if it violates the security policies (Google, 2021d; Google, 2021b), it might be removed, blocked, and returned back to the developer. In this paper, we propose a module called *Approval Process* which leverages DeDup.js as follows. Once vendors detect an extension to be malicious, DeDup.js automatically looks for extensions

Picazo-Sanchez, P., Algehed, M. and Sabelfeld, A.
DeDup.js: Discovering Malicious and Vulnerable Extensions by Detecting Duplication.
DOI: 10.5220/0010900600003120
In *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*, pages 528-535
ISBN: 978-989-758-553-1; ISSN: 2184-4356

with similar files in the repository, discovering similar ones. Also, when a new extension is uploaded, the module outputs two values: *Deletion* and *Malware* similarity scores that indicate how similar the new extension is to other previously deleted and malicious ones, thus providing vendors with more information about the new extension.

Since it is difficult to know the reason why extensions are deleted from the Web Store (e.g., malware, discontinuation, and compatibility), we run an instance of this module and use as input three snapshots of the Store we downloaded at three different times in 2020. We give empirical evidence that previously deleted extensions have a great impact on analyzing new extensions and demonstrate how this module can be used to detect malicious extensions. By manually analyzing the output of this module, we identify a set of over 1k active extensions that redirect users' queries to external servers without the users noticing, clearly violating the privacy policy of browser extensions.

DeDup.js: Discovering. We input DeDup.js with 17 malicious extensions that implement a recently discovered attack named CacheFlow (Avast, 2021). By leveraging deduplication, we detected and reported to Google 53 malicious extensions of which 36 have already been taken down and the rest are investigated, demonstrating how security researchers might benefit from our framework.

Our Contributions. To our knowledge, this is the first paper that presents an approach to discover potentially malicious as well as vulnerable-but-benign extensions due to their similarities. In more detail:

- We present DeDup.js, an approach that detects similarities in browser extensions (Section 3).
- In a large-scale empirical study with browser extensions we demonstrate how DeDup.js detect shared resources and reduced the number of JavaScript to analyze from 1.8M to 117k files (Section 5.1).
- We show how vendor's approval process can be improved by applying DeDup.js, providing them with more information (Section 5.2).
- Using as input the IDs of the extensions that implement a recently discovered attack (nic.cz, 2021; Avast, 2021), we discover 53 new malicious extensions (Section 5.3).

Coordinated Disclosure. We reported to Google all the malicious extensions detected in our empirical study as well as our methodology. Chrome Web Store team removed many of them while some others are still under investigation.

2 BROWSER EXTENSIONS

The amount of sensitive information that extension can retrieve from the user is invaluable. Not only they inject content scripts in all the web pages the users see (e.g., defining `"<all_urls>"` or `"http://*/*"`, `"https://*/*"` patterns) to get all the information displayed, but also background pages can use the most powerful API the browser exposes to extensions. These permission go from cookies, network capabilities, and history to managing other extensions the users have installed in the browser. Thus, the detection of malicious and vulnerable browser extensions at early stages, once they are publicly available for users, is a challenge among researchers.

We differentiate between *malicious* and *benign-but-vulnerable* extensions. While both types can leak sensitive information, the main difference is that the former are developed to attack the security and privacy of the users whereas the latter do so unintentionally.

Depending on the context where the scripts of the extensions run, the security implications vary. To understand how, we differentiate between:

Content Scripts. Are directly injected into a copy of the DOM that the browser keeps updated for each extension.

Background Pages. Are JavaScripts with access to protected APIs (e.g., history, cookies, network traffic). Chrome isolates these scripts from the DOM and the content scripts.

WARs. Scripts defined as Web Accessible Resources (WARs) can be fetched by any other script running in the browser by using the public link that the browser assigns to every WARs of the extensions.

3 DeDup.js ARCHITECTURE

DeDup.js can be deployed in extensions repositories like the Chrome Web Store. We do not require any structural change nor any user interaction. In more detail, in DeDup.js (see Figure 1) we define two main parts: 1) data gathering and filtering, and; 2) modules.

Data Gathering and Filtering. During this first step, DeDup.js uses as input a browser extension and automatically extracts all the static files the extension has. DeDup.js splits the static files into two main sets, one with JavaScript files and another one with CSS, fonts, HTML, and images.

Modules. Due to its architecture, DeDup.js allows extra functionality, what we call modules. Modules can work independently, share information, and/or

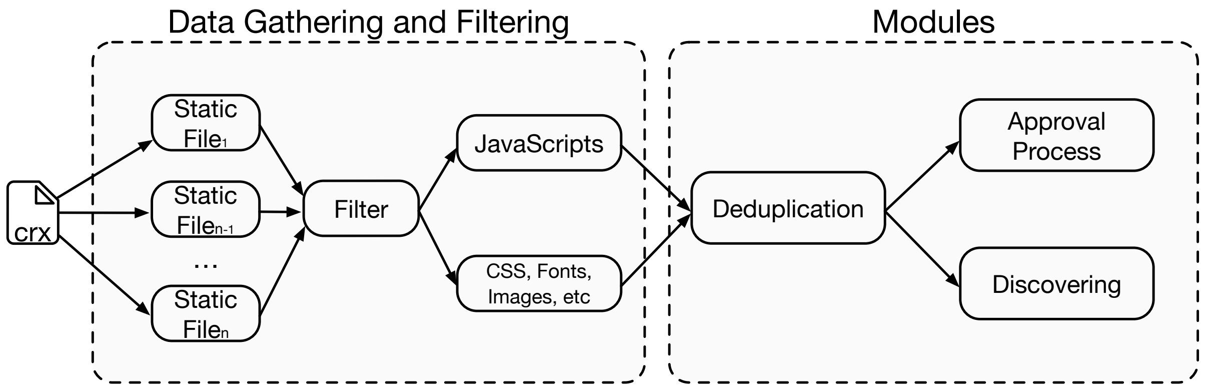

Figure 1: Architecture of DeDup.js.

use other's outputs as input. In particular, DeDup.js includes by default three main modules: 1) Deduplication; 2) Approval Process, and; 3) Discovering. In the following, we explain in more detail each module.

3.1 Modules

Deduplication. This is the main module that DeDup.js implements and it is in charge of detecting similar files in browser extensions and keeping track of those files in a database for further analysis. DeDup.js can specify predefined methods for determining when two files are equivalent or even adding ad-hoc ones. For example, DeDup.js currently supports comparing the Subresource Integrity (SRI) hash of two files but other more accurate methods of comparing files can also be easily accommodated (we discuss this in Section 6).

Approval Process. This module analyzes extensions looking for similarities with both known malware and previously deleted extensions (see Figure 2). DeDup.js extracts the static files of the extension and checks whether they are in the deduplication database or not. It finally outputs two values "Malware Similarity" and "Deleted Similarity" that correspond to how similar the extension is to other extensions previously identified as malware or deleted from the Web Store.

Vendors can include this module as part of their approval process (Google, 2021a; Dev.Opera, 2021). If so, it automatically analyzes extensions when they are uploaded by developers to the Web Store and provides vendors with useful insights and information to accept or to perform further security analysis.

The goal of this module is not automatically accepting or rejecting extensions when they are uploaded by developers nor detecting malicious extensions. DeDup.js aims at helping vendors with this, sometimes tedious, task since vendors perform some automatic and manually checks (Google, 2021a) or in

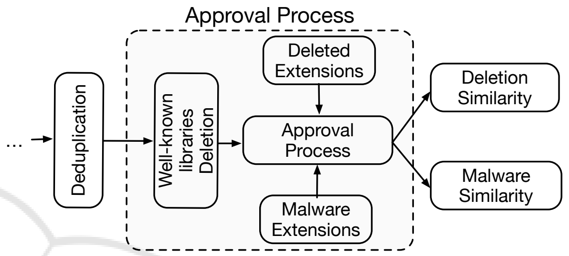

Figure 2: Approval Process module of DeDup.js.

some cases, just manual analysis (Dev.Opera, 2021) by providing a confident and automatic mechanism that prioritizes soundness over completeness. That is, with this module, vendors will be provided with high confidence information about the files that an extension shares with other already deleted or malicious extensions. Therefore, further security analysis should be performed to mark extensions as malicious or not.

Discovering. DeDup.js automatically analyzes all the extensions stored in the Web Store and provides a sorted list of extensions that share files with the one used as input. This module is specially useful when the input extension is either malicious or vulnerable as it outputs similar extensions to that one. If more information is provided, e.g., the malicious file that makes an extension to be malicious, DeDup.js can retrieve a list of extensions that not only include such a file but also those that use the file similarly.

4 A PROOF-OF-CONCEPT IMPLEMENTATION

We implemented a proof-of-concept of DeDup.js in Python on a Linux computer with Intel(R) Core(TM) i7-4790 CPU @3.60GHz, 16GB of RAM. In the following, we explain the implementation decisions we took to deploy such an instance and thus, facilitate the

Table 1: Deleted extensions between snapshots.

Snapshots	Dec 2019	July 2020	Dec 2020
Dec 2019	–	12,443	13,919
July 2020		–	11,148

reproduction of the results presented in this paper.

4.1 Data Gathering and Filtering

We downloaded 3 snapshots of the Web Store within a year and analyzed them independently. The first dataset belongs to December 2019 having 133,365 and 314GB. The second one is from July 2020 and has 138,895 extensions totaling 327GB. Finally, we downloaded the last dataset in December 2020 having 149,783 and 292GB. In total, we analyzed over 1TB of data corresponding to over 422k browser extensions.

To crawl the Web Store, we followed the same methodology for the three snapshots. First, we retrieved the sitemap of the Store and extracted the list of browser extensions. Second, we decompressed each extension and extracted both the manifest and all the files to be analyzed afterwards by DeDup.js.

Regarding the static files, we restricted ourselves to CSS, fonts, HTML, images and JavaScript. Because both fonts and images can have many different formats, DeDup.js uses `ttf`, `otf`, `woff` and `woff2` extensions as fonts whereas for images, it uses `jpg`, `jpeg`, `png`, `gif`, `bmp`, `g2` and `ico` files. Note that the analysis might be expanded with additional static files and formats.

4.2 Modules

Deduplication. We configured DeDup.js to use hash functions and more concretely SRI to detect whether static resources are the same or not. This is a conservative decision since we are forcing our framework to only detect files that are strictly the same. Even though this is an advantage for some static resources like CSS, fonts, and images, it might not be for HTML and JavaScript files. For these resources, other alternatives like sdhash (Roussev, 2010) or ssdeep (Kornblum, 2006) could have been implemented, but the reality is that the final results do not differ from the ones we got with our methodology in the Deduplication module.

For the clustering, we set up a local instance of PostgreSQL to store all the results the code clone process generates, i.e., the id of the extensions, the SRI of the files and their names. In total, it took us 5 hours to run this entire process.

Hardening the Approval Process. Since it is difficult to know the reason why a browser extension was

Table 2: Deduplicated files of December 2019 snapshot with 133,365 browser extensions.

	Files	Repeated	Saved Space (GB)
CSS	334,563 (7.0GB)	233,059 (4.7GB)	67.0%
Fonts	138,560 (8.6GB)	123,444 (7.1GB)	82.5%
HTML	243,144 (1.1GB)	100,671 (0.4GB)	33.6%
Images	1,797,985 (248.6GB)	903,174 (102.1GB)	41.0%
JavaScript	1,654,482 (48.6GB)	1,144,358 (22.8GB)	46.9%
TOTAL	4,168,734 (313.9GB)	2,504,706 (137.1GB)	43.6%

Table 3: Deduplicated files of July 2020 snapshot with 138,895 browser extensions.

	Files	Repeated	Saved Space (GB)
CSS	339,642 (7.1 GB)	243,276 (5.0GB)	71.0%
Fonts	141,045 (8.6 GB)	126,595 (7,1GB)	82.9%
HTML	265,404 (1.3GB)	105,687 (0.4GB)	39.8%
Images	1,801,581 (273.5 GB)	897,304 (109.9GB)	40.2%
JavaScript	1,661,680 (46.8 GB)	1,190,478 (23.7GB)	50.7%
TOTAL	3,943,948 (336GB)	2,457,653 (145.7GB)	43.3%

deleted from the Web Store (e.g., malware, discontinuation, and compatibility), we used the first snapshot as the initial dataset. We next performed the difference between the other two to know which extensions were in common and which ones were deleted. We included the number of deleted extensions between the three snapshots in Table 1. Finally, we assume all the deleted extensions to be malware. This is the most conservative decision since DeDup.js has to analyze more extensions than initially needed.

If DeDup.js were deployed by vendors, they can filter extensions and label them as either "Deleted Extensions" or "Malware Extensions", speeding up this process and not analyzing unnecessary extensions.

Discovering. For this module, we restricted ourselves to show how it can be used to track and discover uncaught malicious extensions still in the Web Store. We used as input 17 browser extensions that implement a recently discovered attack called CacheFlow (Avast, 2021)

5 EMPIRICAL RESULTS

In the following, we explain in more detail the results we got from the instance of DeDup.js we deployed.

5.1 Deduplication

CSS, Fonts, HTML, Images. After analyzing over 422k browser extensions, we observed that even though browser extensions varied the amount and the size of shared resources over a year, DeDup.js remains stable detecting common files. We checked this by comparing the space that shared files have in CSS, fonts, HTML and images over time (see Tables 2 to 4).

Table 4: Deduplicated files of December 2020 snapshot with 149,783 browser extensions.

	Files	Repeated	Saved Space (GB)
CSS	352,354 (8.6GB)	234,134 (5.6GB)	66.4%
Fonts	158,137 (10.7GB)	137,660 (8,5GB)	87.0%
HTML	265,404 (1.3GB)	105,687 (0.4GB)	39.8%
Images	1,935,035 (218.6GB)	897,296 (75.5GB)	46.3%
JavaScript	1,874,275 (63.1GB)	1,300,455 (26.7GB)	69.3%
TOTAL	4,585,205 (302.3GB)	2,675,232 (116.7GB)	58.3%

Using the last snapshot as an example, we extracted the most common shared files of each one of the analyzed static resources, CSS, fonts, HTML, and images and sorted these lists by those files which are heavier in terms of size in the disk. We see how by creating Shared Modules (Google, 2021c) of the analyzed repeated static resources (CSS, fonts, HTML, and images) DeDup.js can save over 85GB of space.

JavaScript. JavaScripts are the files that increase on each dataset; however, the total size does not, being the last dataset the one with more files and less size.

Focusing on the last dataset (Dec 2020), from 149,783 browser extensions, there are 1,874,275 JavaScript files. We computed the SRIs of the content and obtained 573,820 different scripts, meaning that there are over 1,3M repeated files. From the repeated files, we realized that there are only 117,093 different scripts, being around 9% of the repeated files. With this simple yet powerful methodology based on SRI, DeDup.js reduced the number of JavaScript files to analyze from 1.87M files to 117k files approximately.

Takeaway: DeDup.js implements a novel method that allows vendors to detect repeated files among extensions. We are conservative in our decision and implemented a method based on hash algorithms, SRI, to detect whether two files are similar or not. We showed how over 50% of the files of the extensions are shared among others and how DeDup.js can reduce the number of JavaScripts to analyze from over 1.8M to 117k.

5.2 Hardening the Approval Process

We analyzed 12,443 extensions corresponding to the extensions that are in the first snapshot and are not in the second one. One of the first groups of extensions that draw our attention was composed of 1,769 extensions with one common content script named `search_rdir.js`. Such a file is automatically injected when the user visits any of the URIs defined in the manifest file (see Listing 1). Once the script is injected, it redirects the search queries that users perform to external servers without the users noticing it. Interestingly, DeDup.js detected that there is one extension[1] that renamed that file to `search_dir.js` and is still online, evidencing that DeDup.js can detect variations of extensions that either intentionally or unintentionally try to bypass the approval process that Google in this case performs.

```
"matches": ["*://search.yahoo.com/
    search*", "*://duckduckgo.com/*", "
    *://www.google.com/search*", "*://
    www.bing.com/search*", "*://gl-
    search.com/*", "*://redirect.
    lovelytab.com/*", "*://str-search.
    com/*"],
```

Listing 1: Manifest of extensions with search_rdir.js script.

After that, we analyzed all the extensions and sorted the list by the Malware Similarity score. We notice that there are 4 files always in common, i.e., `sweetalert2.js` and `postit.min.js` and `popper.min.js` and `tingle.min.js`. In additiona to that, by anlayzing the manifest files of the extensions, we found out that a big set of them (2,433) also require the webRequestBlocking permission (440 in Dec 2019; 1,280 in July, and; 713 in Dec 2020).

By manually analyzing some of them, we found a subset of 1,374 extensions that redirect all the queries of the users—similar to previous extensions—but using the background page (`js/main.js`) of the extensions instead of content scripts (328 in Dec 2019, 962 in July 2020 and 412 in Dec 2020). Also, DeDup.js detected a new subset of 199 extensions in July's snapshot where they redirect the queries the users perform by using `js.js` instead. This number increased in Dec 2020 totaling 232 extensions.

There is another subset of extensions that start performing maliciously after a given amount of time. We realized that there is one extension that includes a file named `background.js`.The peculiarity of this file is that, after 5 days of the installation of the extension, it starts redirecting the queries that the users perform to an external web page(http://www.lovelychrometab.com/?a=gsp_nevada_00_00_ssg10&q=) encoded in base64 (see Listing 2). It turned out that this file was shared among 4 extensions in December 2019 but such a number increased in July's dataset up to 281 extensions whereas all of them were deleted in the last dataset. However, to our surprise, 5 new extensions with such a file were published again in Dec 2020. We also found slight variations of this `bakground.js` file where the queries of the user are sent to another server instead (https://localspeedtest.com/results.php?q=).

```
if (matchPattern && searchQuery) {
```

[1] dmnbnekngkimbdpmaimkficpllbahbpm

```
return {redirectUrl: '${atob("
    aHR0cDovL3d3dy5sb3ZlbHljaHJvbWV
   0YWIuY29tLz9hPWdzcF9uZXZhZGFfM
   DBfMDBfc3NnMTAmcT0=")}${
       searchQuery}'};}
```

Listing 2: User's queries redirection in background.js.

Finally, DeDup.js detected a different set made of 4,415 unique extensions (4,330 in Dec 2019; 4,396 in July, and; 3,520 in Dec 2020) where 1,045 are still online as of mid-January 2021. In this case, extensions share a file named `search-overwrite.js` in charge of redirecting the traffic to external servers. Also, DeDup.js detected multiple variants of the `search-overwrite.js` file (11 in Dec 2019; 10 in July, and; 6 in Dec 2020).

Takeaway: Detecting malware is a recurrent problem where DeDup.js finds a direct application due to the approval process module it includes. By running an instance of DeDup.js, we showed how vendors could have caught, at least, over 7k malicious extensions early on during the automatic approval process of the extensions. Also, we detected more than 1k new extensions similar to previously deleted extensions from the Web Store still online that steal user's search queries.

5.3 Discovering: CacheFlow Attack

CacheFlow is an attack discovered by the Czech national security team in December 2020 (nic.cz, 2021). Two months later Avast security researchers detected in total 17 malicious extensions for Chrome that were available in the Web Store since 2018 and downloaded by more than 3M users (Avast, 2021) that were exploiting it. Concretely, the extensions hid their Command and Control (C&C) traffic in a covert channel using the Cache-Control HTTP header of their analytics requests where the payload was injected. With this, extensions had a backdoor to execute remote code from the C&C as well as getting analytics information from the users.

We show the potential that DeDup.js has with a real example. By using as input the public list of IDs that Avast released, we discovered 53 new malicious extensions of which 29 are still online. We manually analyzed them and conclude that the attack is still ongoing with slight variations. The purpose of these malicious extensions goes from providing quick access to news websites[2] to an extension that claims to "*help users use Facebook more conveniently*"[3].

[2] ohafompdpdflhjajfhdcjlbjihgdcioc, ngfmaegenlnmgcbalfikhkmgimkedlkb

[3] djfbfhmfeenbkdffimkcagbiimjelgne

Similar to the initially reported CacheFlow extensions, these group of malicious extensions that DeDup.js found, check and send the installation time to the C&C every time the extensions fetch the news—which is exactly 200ms after the background page (`background/main.js`) is executed We observed that if in the response body of the C&C there is a field called `new_constants`, the original information provided in `constants.js` is overwritten.

Among the list of news that the server provides, we discovered that the C&C is sending arbitrary JavaScript code that the extensions execute (see Listing 3). In this case, it sends a snippet that modifies the local storage of the user; however, any other harmful code could be sent instead.

```
foSelectorDescription: ".ev"
foSelectorTitle: ".al"
selectorImagesSettings: "try{\n\tif(
    window.localStorage && !window.
    localStorage.showNotif)\n\t{\n\t\
    twindow.localStorage.showNotif=1;\n
    \t\tchrome.storage && chrome.
    storage.local && chrome.storage.
    local.set({\"push\":false})\n\t}\n}
    catch(e){}"
```

Listing 3: Payload sent by the C&C.

Finally, extensions execute the `eval()` hidden in a clever way: they call a function with a parameter *e*, `e = this.propertyName.substr(1)+ constants.foSelectorTitle.substr(1);`, which is indeed the string `"eval"` (see Listing 3). We detected over 10 extensions with such source code or a variation of it.

As an example, using as input the extension lgjogljbnbfjcaigalbhiagkboajmkkj, DeDup.js detected 8 files (6 .png, 1 .html, and 1 .js) in common with other 6 extensions—being one of them part of the CacheFlow set. We manually analyzed and installed each one of these 5 new and already available extensions in the Web Store and indeed corroborated that all of them can execute remote code in the client's machine after receiving it from the C&C as explained before. Note that after this process, we got 5 malicious extensions that should be analyzed to get similarities with new ones. However, we think that this example demonstrates how DeDup.js can help researchers in detecting new malicious extensions, and therefore, we restricted ourselves to the first level of similarity.

Takeaway: Due to the information gathered and computed in the Deduplication module, the Discovering module is where DeDup.js exhibits its potential. As an example, DeDup.js discovered (and we manually analyzed afterwards) that all the extensions in our most recent dataset sharing `meoptin.js` are malware. Similar cases are extensions sharing `options.html`, `download_big.png`, `thankyou.css`, and `pdf.gif`. We reported all the extensions that we manually analyzed to Google and many of them were deleted already whereas others are under investigation.

6 DISCUSSION & LIMITATIONS

In what follows, we discuss the main limitations that DeDup.js has and how we are working on an extension of the framework to address them.

Deduplication. The DeDup.js instance we implemented relies on SRI to detect similar files. Although this could be initially a coarse-grained code clone detection mechanism based on the whole-file where malicious developers can easily bypass by generating small changes like introducing random blank spaces, the reality is that DeDup.js drastically reduced the number of libraries to be analyzed going from 1,874,275 to 117,093 in the last dataset we have (as of December 2020)—meaning that 91% of the JavaScripts of the extensions are repeated and therefore concluding that the SRI works for our purpose. Determining whether two files are the same is out of the scope of this paper and is an entire research topic itself in formal methods and program verification called program equivalence (Badihi et al., 2020).

Evasion Techniques. Malware tends to reuse the same piece of software, especially the one that performs the exploits (Calleja et al., 2019). Even though we could not find evidence of extensions implementing evasion techniques, we cannot ignore it.Since DeDup.js relies on SRI, developers might be tempted to bypass it. Yet, this is not the case of benign-but-vulnerable extensions, where developers are not aware of the vulnerabilities introduced (being malicious otherwise). However, for malicious extensions, the consequence is that both Malware and Deleted scores will be lower than they should be. We implemented DeDup.js modularly such that replacing the actual SRI algorithm and incorporating fuzzy hashing techniques like sdhash (Roussev, 2010) or ssdeep (Kornblum, 2006) will be as easy as including such new libraries.

Approval Process. The goal of DeDup.js is not to automatically detect malicious extensions but rather provide an automatic mechanism that alerts vendors about extensions that look similar to previously deleted and malicious ones. Thus, we do not include any statistical classification analysis (i.e., sensitivity and specificity). This initial version of DeDup.js does not take into account the severity of the vulnerability, i.e., an extension could just share one extremely vulnerable file out of 100 files and the Malware similarity will output 1%. We understand that during this process, completeness is not as important as soundness, and therefore, DeDup.js prioritizes soundness. Having said that, we plan to expand DeDup.js to allow vendors to include the severity of the vulnerabilities such that the Approval Process module not only outputs both Malware and Deletion Similarity values but also how dangerous the extensions are. In addition to that, we are developing an additional module that can automatically detect and classify both the vulnerabilities that extensions expose (vulnerable extensions) as well as the attacks they might exploit (malicious extensions).

7 RELATED WORK

Security of browser extensions is paramount due to the sensitivity of the information that they manipulate.

In Chrome, many works have been proposed using dynamic analysis. Hulk (Kapravelos et al., 2014) analyzes extensions based on honey pages and a fuzzer to fire event handlers that extensions rely upon. ExtensionGuard (Chang and Chen, 2016) is a dynamic taint system that prevents extensions from leaking sensitive information to web pages by modifying the source code of the extensions. Mystique (Chen and Kapravelos, 2018) performs a taint analysis of the information flow in browser extensions. A recent study (Pantelaios et al., 2020) detected 64 malicious extensions by analyzing the comments and ratings of the users.

In code clone detection, researchers have mainly focused on statically-typed languages (e.g., (Kamiya et al., 2002; Bowman and Huang, 2020; Li et al., 2016; Kim et al., 2017)). A recent study showed that JavaScript exhibits different clone properties and cloning practices, as well as reporting that developers duplicate code intentionally (Cheung et al., 2016). We relied on hash functions as code detection since we not only analyze source code but also other resources like CSS, fonts, and images. However, DeDup.js may benefit from fuzzy hashing techniques like sdhash (Roussev, 2010) or ssdeep (Kornblum, 2006). Having said that, DeDup.js is general enough that classical code cloning tools like LICCA (Vislavski et al., 2018) can be also added to the Deduplication module.

Complementing the prior work, we present DeDup.js, a framework that detects malicious browser extensions based on by leveraging deduplication. With DeDup.js a vast majority of malicious extensions can be caught during the acceptance process (Section 5.2) without modifying the browser's engine nor falling into computational overhead that dynamic analysis originates (Chang et al., 2008).

8 CONCLUSIONS

Our work puts a spotlight on the problem of discovering malicious and vulnerable browser extensions by detecting duplication. To address the problem, we presented DeDup.js, an approach that incorporates similarity analysis for achieving two goals: detecting potentially malicious extensions during the approval process and discovering malicious extensions.

We implemented and deployed an instance of DeDup.js and analyzed more than 422k browser extensions stored in the Web Store over a year. In summary, DeDup.js: 1) detected more than 7k extensions that should not have been published in the Web Store. Also, we found more than 1k malicious extensions still online that send user's queries to external servers without the user's knowledge, and; 2) detected 53 malicious extensions of which 36 Google has already taken down and the rest are investigated. We did so by using as input 17 already known malicious extensions IDs, thus demonstrating how DeDup.js can change the game of malware detection in browser extensions.

ACKNOWLEDGMENTS

This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Facebook.

REFERENCES

Avast (2021). Backdoored browser extensions hid malicious traffic in analytics requests. https://decoded.avast.io/janvojtesek/backdoored-browser-extensions-hid-malicious-traffic-in-analytics-requests/.

Badihi, S., Akinotcho, F., Li, Y., and Rubin, J. (2020). Ardiff: Scaling program equivalence checking via iterative abstraction and refinement of common code. In *FSE*.

Bowman, B. and Huang, H. H. (2020). Vgraph: A robust vulnerable code clone detection system using code property triplets. In *Euro S&P*.

Calleja, A., Tapiador, J., and Caballero, J. (2019). The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security*, 14(12).

Chang, W. and Chen, S. (2016). Extensionguard: Towards runtime browser extension information leakage detection. In *CNS*, pages 154–162.

Chang, W., Streiff, B., and Lin, C. (2008). Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS*, page 39–50.

Chen, Q. and Kapravelos, A. (2018). Mystique: Uncovering information leakage from browser extensions. In *CCS*, page 1687–1700.

Cheung, W. T., Ryu, S., and Kim, S. (2016). Development nature matters: An empirical study of code clones in javascript applications. *Empirical Softw. Engg.*, 21(2).

Dev.Opera (2021). Publishing guidelines. https://dev.opera.com/extensions/publishing-guidelines/.

Extension Monitor (2021). Breaking down the chrome web store. https://extensionmonitor.com/blog/breaking-down-the-chrome-web-store-part-1.

Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In *FSE*.

Google (2021a). Frequently asked questions. https://developer.chrome.com/docs/webstore/faq/#faq-listing-108.

Google (2021b). Program policies. https://developer.chrome.com/docs/webstore/program_policies/.

Google (2021c). Shared modules. https://developer.chrome.com/apps/shared_modules.

Google (2021d). Unwanted software policy. https://www.google.com/about/unwanted-software-policy.html.

Google (2021e). What types of apps or extensions are not allowed in the store? https://developer.chrome.com/docs/webstore/faq/#faq-gen-22.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670.

Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., and Paxson, V. (2014). Hulk: Eliciting malicious behavior in browser extensions. In *USENIX*.

Kim, S., Woo, S., Lee, H., and Oh, H. (2017). Vuddy: A scalable approach for vulnerable code clone discovery. In *S&P*.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., and Hu, J. (2016). Vulpecker: An automated vulnerability detection system based on code similarity analysis. In *ACSAC*.

Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. (2017). Déjàvu: A map of code duplicates on github. *ACM Program. Lang.*

nic.cz (2021). Hledání škodlivého kódu mezi doplňky. https://blog.nic.cz/2020/11/19/hledani-skodliveho-kodu-mezi-doplnky/.

Pantelaios, N., Nikiforakis, N., and Kapravelos, A. (2020). You've Changed: Detecting Malicious Browser Extensions through their Update Deltas. In *CCS*.

Roussev, V. (2010). Data fingerprinting with similarity digests. In Chow, K.-P. and Shenoi, S., editors, *Advances in Digital Forensics VI*, pages 207–226.

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495.

Vislavski, T., Rakić, G., Cardozo, N., and Budimac, Z. (2018). Licca: A tool for cross-language clone detection. In *SANER*, pages 512–516.