DOMQuery: A large-scale Analysis of Browser Extensions

Interactions with Websites

Alexander Shevtsov

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisors:
Prof. *Evangelos Markatos*,
Dr. *Sotiris Ioannidis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

DOMQuery: A large-scale Analysis of Browser Extensions

Thesis submitted by
Alexander Shevtsov
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Alexander Shevtsov

Committee approvals: _____
Evangelos Markatos
Professor, Thesis Supervisor

_____
Sotiris Ioannidis
Research Director, Thesis Advisor, Committee Member

_____
Polyvios Pratikakis
Assistant Professor, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, February 2019

DOMQuery: A large-scale Analysis of Browser Extensions

Abstract

Modern browsers increased and extended their functionality in order to become more flexible and attract a plethora of users. Indeed many of the big players such as Google Chrome, Mozilla Firefox and Microsoft Edge provide additional enhanced features and privacy solutions through the form of browser extensions. These extensions are available in the browser's market (an online store), which offers hundreds of thousands of extensions to the user; some of them being very popular with millions of downloads.

In this master thesis, we present DOMQuery, a system that analyzes the interactions between browser extensions and websites' DOM[1] elements. We selected 705,305 different versions out of 307,822 extensions and crawled the top one million Alexa websites while creating an index of all the DOM elements found in these websites. Our system identifies the webpages and the specific DOM elements that extensions manipulate in order to identify cases of extension misuse. Moreover, our system analyzes the extension's permissions and the JavaScripts used in order to cluster extensions based on their functionality. Using such an approach we can identify quickly extensions that perform a suspicious activity.

Analyzing thousands of extensions is a problematic and (execution) time-consuming task and requires tackling several challenges. We address these issues with the use of Kubernetes and running multiple Docker containers in parallel. Our analysis of 6.4 billions lines of HTML and 85 millions lines of JavaScript code resulted in identifying extensions that target specific websites. Furthermore, by analyzing the permissions from the extension's manifest, we found 8,340 extensions with wrong permission usage. To foster more research and shed more light on this phenomenon, we will publicly release our dataset.

[1]The Document Object Model is a cross-platform and language-independent application programming interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document.

DomQuery: Ανάλυση ευρείας κλίμακας των επεκτάσεων προγραμμάτων περιήγησης

Περίληψη

Τα σύγχρονα προγράμματα περιήγησης αποκτούν έξτρα λειτουργικότητα, ώστε να γίνουν πιο ευέλικτα και να προσελκύσουν πολλούς χρήστες. Πράγματι, πολλοί από τους μεγάλους παίκτες όπως το Google Chrome, Mozilla Firefox και το Microsoft Edge παρέχουν πρόσθετες βελτιωμένες δυνατότητες και λύσεις απορρήτου μεσώ της μορφής των επεκτάσεων. Αυτές οι επεκτάσεις είναι διαθέσιμες στην αγορά του προγράμματος περιήγησης (ηλεκτρονικό κατάστημα), η οποία προσφέρει χιλιάδες επεκτάσεις στον χρήστη. Μερικές από αυτές είναι πολύ δημοφιλής με εκατομμύρια λήψεις.

Σε αυτή τη μεταπτυχιακή εργασία παρουσιάζουμε το DOMQuery, ένα σύστημα που αναλύει τις αλληλεπιδράσεις μεταξύ των επεκτάσεων του προγράμματος περιήγησης και των στοιχείων DOM[2]των ιστοτόπων. Επιλέξαμε 705,305 διαφορετικές εκδόσεις από 307,822 επεκτάσεις και συλλέξαμε τα κορυφαία ένα εκατομμύριο ιστότοπους της Αλέξα δημιουργώντας ένα ευρετήριο όλων των στοιχείων DOM που βρίσκουνται σε αυτούς τους ιστότοπους. Το σύστημά μας, προσδιορίζει τις ιστοσελίδες και τα συγκεκριμένα στοιχεία DOM που χειρίζονται οι επεκτάσεις, προκειμένου να εντοπίσουν περιπτώσεις κακής χρήσης επέκτασης. Επιπλέον, το σύστημά μας αναλύει τα δικαιώματα επεκτάσεών και τα JavaScripts που χρησιμοποιούνται από αυτά για την ομαδοποίηση επεκτάσεων με βάση τη λειτουργικότητά τους. Χρησιμοποιώντας μια τέτοια προσέγγιση, μπορούμε να εντοπίσουμε γρήγορα επεκτάσεις που εκτελούν ύποπτη δραστηριότητα.

Η ανάλυση χιλιάδων επεκτάσεων είναι ένα δύσκολο και (εκτελεστικά) χρονοβόρο έργο και απαιτεί την αντιμετώπιση αρκετών προκλήσεων. Αντιμετωπίζουμε αυτά τα ζητήματα με τη χρήση των Kubernetes και τρέχοντας πολλαπλά Docker παράλληλα. Η ανάλυση 6.4 δισεκατομμυρίων γραμμών HTML και 85 εκατομμυρίων γραμμών κώδικα JavaScript οδήγησε στην αναγνώριση επεκτάσεων που στοχεύουν συγκεκριμένους ιστότοπους. Επιπλέον, αναλύοντας τα δικαιώματα των επεκτάσεών βρήκαμε 8,340 περιπτώσεις με λάθος χρήση των δικαιωμάτων. Στην προσπάθειά μας να ερευνήσουμε περισσότερο και να ρίξουμε φως στο φαινόμενο αυτό, θα δημοσιεύσουμε το σύνολο δεδομένων μας.

[2]DOM (Το Μοντέλο Αντικειμένου του Εγγράφου) είναι μια διεπαφή προγραμματισμού εφαρμογών ανεξαρτήτως πλατφόρμας και γλώσσας που αντιμετωπίζει ένα έγγραφο HTML, XHTML ή XML ως δομή δέντρου όπου κάθε κόμβος είναι ένα αντικείμενο που αντιπροσωπεύει ένα μέρος του εγγράφου.

Ευχαριστίες

First of all, I would like to thank my supervisor, Professor Evangelos Markatos, for his valuable guidance. I also want to express my deepest gratitude to my advisor, Dr. Sotiris Ioannidis, for giving me the opportunity to work on so many different, challenging and interesting projects, over the past three years. His support and advice greatly contributed to my academic and technical growth. Moreover, I feel thankful to Thanasis Petsas, for his guidance during my first steps of this academic journey and for setting the foundations of this work. My warmest regards to Michalis Diamantaris, Kostas Solomos, Giorgos Tsirantonakis, Dimitris Deyannis, Evangelias Papadogiannaki, Panagiotis Papadopoulos, Thanasis Petsas, Evangelos Ladakis, Giorgos Christou, Konstantinos Kleftogiorgos, Panagiotis Ilias and all the other present and past members of the Distributed Computing System Laboratory, for their friendship, advice and commitment. Finally, I want to thank my family and friends for all their invaluable support and caring.

στους γονείς μου

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Internet browsers have become an integral part of computer systems, and most of the popular web browsers are implemented for desktop and mobile use in different operating systems (Windows, Linux, Mac, iOS and Android). According to `gs.statcounter.com` the most popular browser is Chrome, Safari, and UC Browser. Google Chrome browser has a higher market share index of 59.69% across all platforms and 67.6% of the desktop market. As the most popular browser, Google Chrome allows to use different modules over the browser, and these modules are called extensions; each extension adds additional functionality to the browser. Using browser extensions allows developers to reduce the size and complexity of browser source code. To facilitate this Google created an extension web store which allows developers across the world to publish extensions for specific user needs.

Most of the extensions in the Chrome web store are distributed free and include extensions such as shared document editing, forecast, news, ad-blockers, and other privacy tools. Unfortunately, as mentioned in different studies [14] [3] free functionality comes at a cost, where users are the product and pay with their personal data. Moreover, there are multiple cases free functionality comes with suspicious and even malicious functionality which is hidden from the user [7] [4]. Therefore it is crucial to understand and analyze what type of web pages these extensions are targeting. To this end, we created a web platform and uploaded all the data we collected during our experiments. We also perform a correlation between the data that is accessible to extensions when visiting the most popular websites.

Every web site is based on HTML basic DOM structure with some standard elements like header, body, and footer. However, many developers/companies use their own class/id/tags elements in their websites. Due to this fact an attacker can search in a document for specific elements and identify what is the current URL in a browser tab or even change different fields/frames on a web site such as replacing advertisement frame with his frame). As mentioned in [2], this type of attack can also be executed by extensions that don't have permission for tab URL location.

In this master thesis, we present a sophisticated technique that analyzes Chrome browser extensions with use of docker images and Kubernetes. We use our novel infrastructure to collect DOM elements for a significant amount of top-ranked websites. Finally, we crosschecked the results for these two experiments and provided access to our data through a web platform. The web platform allows users to identify what DOM elements a specific extension is looking for, on which websites these elements are located, or search for URLs that are known to be targeted by different extensions.

We created our extension dataset by collecting the most popular Google Chrome extension and different versions of them. Overall, our collections consist of more than 700k crx files. Also, in order to collect a large amount of DOM elements we crawled the top 1 million Alexa websites. In this thesis, first, we describe the collection technique for the extensions' parsing and the websites' crawling, and then we present the results from our analysis.

1.1 Contributions

To summarize, the main contributions of this master thesis are:

- a public framework to bridge the gap between extension analysis and visited sites
- we developed the sophisticated technique for large scale extension analysis, where we analyzed more than 700k of different extension crx files
- in-depth results we identify that 8,340 of the analyzed extensions have wrong permissions and can be reduced to have access in the limited amount of websites that they query.

Chapter 2

Background

As any Web application browser extension is third-party code. However, browser extensions executed with permissions, they have access to APIs which grant access to all content within the browser. Permission is the primary tool that allows developers to restrict execution, but extensions frequently over-request permissions. For example, [13] showed that 71% of the top 500 Chrome extensions use permissions that support leaking private information. As a solution, they have been proposed mandatory access control design to protect browser privacy.

One of the first research publications [8] show that Chrome extensions security model is not a panacea for the different type of attacks of extensions. Through a series of practical bot-based attacks, they demonstrated that malicious Chrome extensions pose serious thread even for browser them self. Authors propose new policies to enforce micro-privilege management and differentiate DOM elements.

In Hulk [2] researchers present a system that was designed for first large scale dynamic analysis of Chrome browser extensions with use of Honey pages. This work generates web content in order to trigger the malicious behaviour of extension. Some of the extensions, for raising the incomes, can maliciously add or even replace websites advertisements with their own. This type of extension behaviour was described in [6] work, where authors showed 249 Chrome web store extensions with advertisement manipulations.

Ex-Ray [9] authors confirmed Chrome extensions privacy leakage and third-party data processing. The Framework they create can automatically detect privacy leakage by dynamically analyzing Browser traffic generated during execution time with 96.9% accuracy and no false positives.

Privacy diffusion [12], develop large scale Chrome extensions analysis framework which analyzed 10k of most popular extensions. With this process, they discover 6.3% of evaluated extensions have some information leakage, where the majority of these cases being accidental.

Browser extension by them self can generate a fingerprint of the user. In [5] authors show enumerating attack, where attacker collect browser installed extensions and can use this information for fingerprinting a user on the web. Win use

of other fingerprinting techniques this extra information increase the accuracy of uniqueness score.

Another work [15] show that browser extensions that use ads as their monetization strategy often facilitate the deployment of malvertising. Moreover, while some extensions serve ads from ad networks that support malvertising, other extensions maliciously alter the content of visited webpages to force users into installing malware.

Some of the papers generate automated framework [11], where they monitor extension web store for a new extension. While they identify new extension, framework analyze the source of extension with a different technique that can identify if extension has any malicious functionality. With this framework, they achieve to remove 9,523 malicious extensions from the chrome web store.

Chapter 3

Methodology

Our methodology consists of two different experiments: analyzing extensions and collecting DOM elements. This section will describe the purpose of Chrome extensions and how they are executed, and then we will discuss the methodology for each of these experiments.

Google extensions are small software that customizes the browser. Using extensions the user can extend the Chrome functionality and behaviour to their individual preferences. Extensions are built using HTML and JavaScript. Every Chrome extension is composed of two types of scripts: content_scripts and core extensions script. The core type of scripts is executed with the Chrome Browser process; they cannot directly communicate with websites, background jobs, etc. Content scripts, on the contrary, can communicate with a web page directly and they are able to read and write DOM elements of a web page. These two types of scripts run as separate processes and communicate by sending clones over an authenticated channel. For each web page Chrome creates a separated and isolated instance of the content script in order to avoid privacy leakage from the extension side.

In summary, content scripts have enough privileges to interact with the user page, where they can look up and change elements in DOM structure and even create new elements. Such interaction with the user page needs to be controlled with more strict permissions. We describe permissions in Extension permissions section with more details, where some of the extensions are used in wrong way URL matching permissions.

3.1 Extension Analyzing

For our experiments, we utilize Google Chrome browser, but we have several limitations:

- We cannot use multiple instances of Chrome browser in the same operating system that will use separate user history

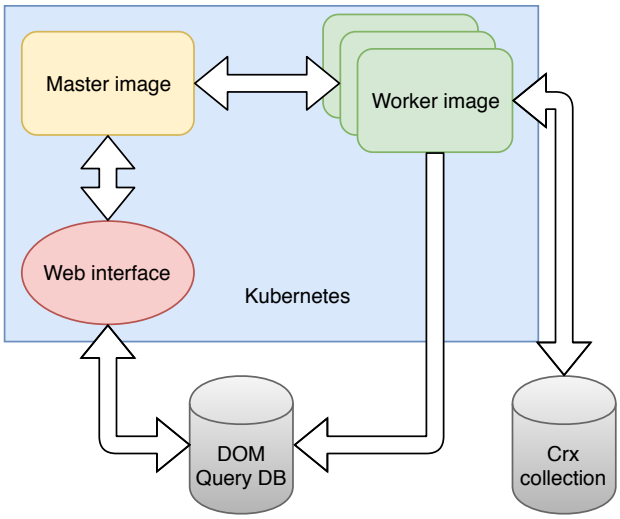

Figure 3.1: Our master worker implementation scheme. Where the web interface is the front end of the master image with access to the database, that allows us to monitor execution flow. Master images are the main component which creates a queue of job's that should be executed by worker image. Worker image pull jobs from the queue and next extension crx file from crx collection.

- Multiple instances of Chrome share the same installed extensions, and we cannot separate them in different execution instances

These limitations obliged us to use operating system virtualisation. To solve this problem we Docker containers that allow us to execute multiple instances with the same shared OS. Also for this experiment, we use Kubernetes as the primary execution platform, because it supports Docker images for content creation. Kubernetes is an open-source container-orchestration system for automating deployment. It provides automating deployment, scaling, and operations of application containers across clusters of hosts. Docker images are not architecture depended, and the same environment can be used by merely using a docker image. By using Kubernetes and Docker images, we can create a scalable execution which allows us to reconstruct the same experiment no matter of the hardware and the library dependencies.

We build our docker images on top of Ubuntu Linux 16.04 and installed the following applications and libraries:

- rq-queue is a library that implements a job's queue structure
- python3.5 (libraries: redis, flask, monogengine, mysql) these libraries implement the web interface and database communication
- Google Chrome browser

- catapult project is a local web cache proxy tool that server preloaded pages from the local store, in the way that it would be requested from the original web server
- acornjs is a JavaScritp code parser tool that removes any comments from source code

As can be seen in Figure 3.1, our implementation consists of two types of images: Master and Worker images. Each one of these images has a particular task to solve, which we describe more in the extensions parsing section. Another part of this implementation is the data collection storage. We utilize MongoDB and MySQL database platforms, which gives us the opportunity to create indexes over collected DOM query key-values and store different elements (big log files, and small indexed dom elements) in the more suitable database for each case.

3.2 DOM elements crawling

In order to get a complete picture of what extensions are targeting and correlate extensions and specific URLs, we need the DOM structure/tree of the most popular websites. To do this, we crawled the Alexa top one million websites. As with the previous experiment, we have a significant amount of data, and we need a scalable implementation to get the results fast for a large number of URLs. For this reason, we use Docker containers with RQ python queue, that allows us to create a modular framework that splits tasks in jobs that are processed by docker containers. We run these containers in a Kubernetes cluster, which allowed us to scale up the experiment with multiple parallel pods. Each pod contains a python script that leverages the Chrome DevTool protocol and monitors the DOM elements during their creation. In comparison with DOM screenshot, we can gather all the elements of a webpage even if they are removed afterwards. Since we are able to gather all the DOM elements of a webpage, we are also able to parse all the ids, tags, names and classes.

Chapter 4

Implementation

4.1 Extension parsing

In this section, we describe in details the two steps of our experiment. As we mentioned earlier, in the first step, we run two types of docker images (master and worker) over Kubernetes platform.

4.1.1 Master image

This is the main image/container that is responsible for user-friendly interface execution; this interface allows us to start the experiment, monitor work-flow execution and present DOM query elements that were collected in execution time. The second important part of this image/container is to push/create jobs into queue and monitor execution for failing job's and reschedule these jobs. For both of this purpose, we use python3.5 with webflask and rq-queue in order to achieve our goals.

4.1.2 Worker image

It's an image that takes orders from the master image/container. In this case, a worker is created in order to perform extensions analyzing phase which includes these steps:

- Pull jobs from the queue, get an extension identifier that worker needs to extract from the database
- Download extension crx file from the database and extract source code to a local folder
- Parse source code of JavaScript and HTML file and identify hardcoded URLs
- Collect content scripts files that are declared in the manifest file and generate the md5 hash for each of them

- Read all JavaScript files in extension folder and find matches in source code with "chrome.tabs.executeScript" function call
- For all of the matches generate an md5 hash for JavaScript code inside of the function call
- Patch all JavaScript files found inside the extension folder with our hook file (see Section 4.1.3 for details), where hook file is located at the beginning of the file and followed by original JavaScript code
- Start the local cache library (catapult) for the web pages we statically use in the experiment
- Run google-chrome for 20 sec. with static and extension hardcoded URLs with patched extension loaded in the browser instance
- After chrome execution analyze chrome log stream in order to identify hook logging stream with extension performed DOM queries
- Sort all elements by type (id/tag/name/class) and store unique values to our database
- Remove all Chrome cache, history and catapult logs from local folders

After patching and collecting hardcoded URLs from the extensions source code, we load them using an instance of the Chrome browser. From this point, we start the Chrome browser with this specific extension in use. The next step is to open several tabs in the same browser instance. Exploitation of multiple tabs with different URLs, trigger more DOM query events in same execution time. That occurs because of the extension matching technique where content script triggered on specific URL and in the case when the extension has hidden functionality that is triggered by URL. This method reduces computation time. We keep the browser open for 20 seconds, and during this time our hook file prints all the DOM queries in the Chrome log console. Each query has a key value, function type and tab URL. We collect this information at the end of the execution. Before submitting the data to the database, we sort all DOM query by call stack, to separate between events which were initiated by extension JavaScript and events that were triggered by injected JavaScript files. Here is very important to mention that extension can inject they JavaScript directly to the user page, with our hook method, we will collect DOM query generated by the page itself. For this purpose for each query event, we collect call stack, that shows the origin of the query. When the execution of the Chrome browser execution is over, we reverse the image to its original clear state by clearing the browser cache, removing all the logs, and the browsing history. We perform this procedure in order to keep the experiment environment consistent during repetitions for different extensions.

4.1.3 Hook file

Our hooks are inspired by HoneyPages [2]. In this experiment, we want to monitor function calls over DOM elements. This type of functions returns elements from DOM structure, when such elements are returned, different fields of these elements can be read or even changed by extensions. We monitor the following such function calls: getElementById, getElementsByClassName, getElementsByName, getElementsByTagName, getElementsByTagNameNS, querySelector and querySelectorAll. In order to intercept DOM queries, we create hooks in the source code that create a JavaScript function proxy [1] for all of these functions and before redirecting the function call back to the origin. Our proxy collects the function arguments (DOM query key value) and function name.

Such function calls also can be executed over "chrome.tabs.executeScript" functionality. This function call allows executing JavaScript directly on the user page. For this type of execution, we create a proxy that performs regular expression check over passing argument, in order to find our DOM query that we are interested in.

Google Chrome extensions also can perform this DOM query over extensions background page and not the real user page, that why we check in run time which chrome API call extension can make, and this information allows us to separate background page and real user page DOM query. In the case when a query is forwarded to the user page, our hook file prints this function key value, function type and executed tab URL link into chrome log console.

4.1.4 Patching extension

As we described in the previous section, the hook file allows us to register and separate function calls. The only problem with this type of implementation is that these functions should appear in every extensions JavaScript file. In order to solve this problem, we unpack the extension and patch every ".js" file with our hook at the beginning of a script. We place it in the begging of the script file, to catch any DOM queries from JavaScript.

One of the problems that we identify while we execute the first round of the experiment, some of the extensions copy their JavaScript code directly to the user page. With our method, this code will contain the hook file. This kind of extension develops a significant amount of noise (Dom query elements that legitimate website JavaScript will ask). Such legitimate DOM elements request will be filtered. For this reason, we use error elements, and we analyze the call stack for each call. With this method, we reduce noise elements in our database and also can prove that each element originates from JavaScript extensions and which line of code triggered this event.

After this procedure, we have an extension on which every DOM query will occur in the Chrome console log. In the execution the only thing needed is to collect all these logs and sort them by type of function call, tab URL link and make sure that every pair of (key, function type, URL) is unique, that will reduce

the size of our database.

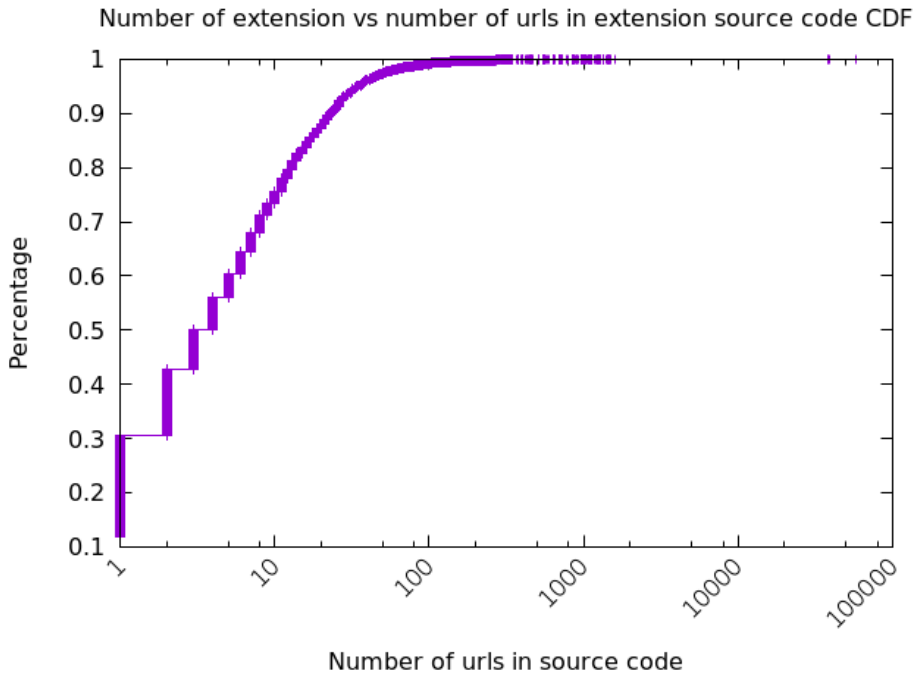

Figure 4.1: This CDF figure shows the numbers of unique URLs that were found in each extension crx JavaScript and HTML source code.

4.1.5 Url usage

Before we described patching of extension, and we mentioned that we use multiple tabs in the same browser instance. We utilize multiple tabs in the same browser session because different extension can have a hidden function, and this type of functionality can be triggered only by a specific URL. With knowledge of that, we use ten static web pages (some URLs that we choose before running the experiments like Google, Wikipedia, eBay, etc.), and also we parse source code of extension before execution, and with regular expressions, we collect URL links that were hardcoded in the source code of extension.

With all these URLs we collect a large amount of data about the extension, but also it reduces the stability of experiment because a web page can be change over execution time and some extensions will be executed over a new version of the web site. All that created a need for local cache catalogue for static web pages, this where we use the catapult project with some minor changes. Catapult project creates a cache of all browser requests and responses that were received from the web server and with use of this traffic catapult create the local cache file. All that

ByName	ByClassName	ByTagName	ById
14,046,495	157,433,375	25,428,878	63,331,199

Table 4.1: This table shows the number of registered DOM elements from Alexa most popular one million websites.

allows us to reconstruct communication between browser and web server for each extension.

4.2 DOM element collection

The second part of the experiment collects the DOM structure of the top one million Alexa websites. That procedure also needs to implement with the use of Kubernetes and docker image, also like the previous experiment, we use Ubuntu 16.04 as a base image. All that will allow us to get the results in a short amount of time, and make that experiments reproducible in the future. Let's describe each critical component.

4.2.1 Master image

In this implementation, we also need a web platform, because it creates a straightforward way to interact with experiment creation, execution, work-flow and failed jobs. For this purpose, we use python web-flask as in extension analyzing phase. Also, this implementation uses a large number of URLs, and we choose Kubernetes in order to run multiple containers/instances of workers images. In order to reduce complexity and create manageable job flow python rq-queue is used. Where web interface creates a list of jobs with one URL per job, and each of these jobs is pushed in the queue, from which each worker will pull a job for execution. One of the big pros of this implementation is managing failing jobs, in rq-queue, all jobs that were failed in execution time are reallocated in a different queue, that allow us to re-run this jobs again later.

As the master image of the previous experiment, this almost identical implementation with the use of python web-flask for the web platform that presents in user-friendly form the execution workflow and lives time results. Also, this image creates a job for each unique URL from Alexa CSV file, and the working queue contains each of the jobs that should be executed by workers.

4.2.2 Worker image

This is most important part of implementation because it contains google-chrome browser, and this image is responsible for experiment execution:

- pull jobs from queue

Domain names	Number of extension
w3.org	155,771
google.com	104,003
github.com	98,914
facebook.com	89,456
twitter.com	82,619
chrome.google.com	80,146
youtube.com	61,134
fonts.googleapis.com	52,590
ssl.google-analytics.com	49,889
google-analytics.com	44,674
plus.google.com	37,807
mail.google.com	36,009
fb.me	32,684
googleapis.com	30,390
momentjs.com	29,261
errors.angularjs.org	27,408
linkedin.com	26,847
vk.com	24,179
pinterest.com	22,897
docs.google.com	19,182

Table 4.2: Most popular domain names found in extensions JavaScript and HTML source code.

- start google-chrome with Dev Tools protocol
- create DOM event call-backs
- patch URL prefix
- open tab with one of the Alexa URLs
- register all DOM elements
- submit the data
- set google-chrome in clear stage

We use Google Chrome with Dev Tool's protocol which allows to inspect, debug and profile chrome browser. This instrumentation allows us to create interceptions events over different events in the browser through debugging port in chrome browser. This port should be defined in executions parameters.

In this experiment we use DOM.childNodeInserted, by this function, we intercept creation new node in DOM structure, whenever new DOM node created

we collect each element of this node. Nevertheless, DOM structure or elements inside the node can be changed in run time by js code execution, and new elements can be inserted. For this type of dynamic changes in web page structure, we use DOM.getDocument that retrieve all nodes from the DOM tree. Our implementation performs such calls frequently till now new elements not found for a couple of time in the row, in order to collect as many elements as possible from DOM structure. Whenever new elements are found, they would be sorted by there type (class, id, tag, name).

One of the problems was URL patching, Alexa web ranks CSV file do not contain a prefix of http/https is not a big problem while chrome can resolve URL only by it postfix part, but running chrome with Dev Tools and feeding URL through python into browser tab bypass this procedure. For this reason, we test different prefixes before feeding the URL into a web browser.

After execution, at the moment when all elements are collected, data is submitted into the database.

Chapter 5

Data analysis

Our extension collection contains 705,305 different crx files where 307,822 is different extensions, and other crx files are just different versions of extensions. After permission analysis, we identify that 439,222 of all extensions have permission, that allows them to perform DOM query. With dynamic analysis, we found that only 51,463 of these extensions perform DOM query. We managed to analyse all these crx files, and our collection of DOM query consists of 810,768 entries (see Table 5.1).

All registered DOM queries (Table: 5.2), reveals that the most popular elements that extension was looking for are basic HTML-structure (body, head, html, a, img, input, title, ..). Never less we identify a significant amount of extensions that was looking for sizzle, script and sizcache elements, after our experiments we identify that the JQuery library generated these elements. With the technique that we were using in Patching section, where we patch any js file in the extension folder, we also patch the JQuery local libraries that extension was using for their porpoises. These DOM query created by JQuery for library purposes.

In the same experiment, we also parse the source code of each extension/crx in order to match hardcoded URL links; as a result, we collect 290,310 unique URLs (Figure: 6.1). From our data we also found that 512,502 of extensions/crx file contain at least one URL in their source code (Table: 4.2), in Figure 4.1 we present CDF over the number of URLs found in each extension/crx file.

While execution time, we identify that some elements/lib use timestamp/pseudo-random number postfix of element key value(for example jquery sizzle.., sizcache.., etc.), such type can be used in order to make more difficult to select the right DOM element, but in our case is not necessary. Our implementation patches these elements before sending them to the database, by removing number postfix of the key value and replacing it with "+timestamp" string. This method reduces complexity in cross-checking data in two different collections because it is impossible to run both experiments at the same time for this number of extension/crx files and parallel for Alexa 1M web URLs.

Alexa web ranking creates a list of the most popular web, but some of these

ByName	ByClass	ByTag	ById	Total
312,669	27,559	278,698	191,842	810,768

Table 5.1: Number of registered extension DOM query.

URLs is offline or not available for public access. For this reason, Alexa 1m collection consists of 961,540 web service that responds for a web request. In this experiment, we generate four different tables which in sum have 260,239,947 entries, in more detail in Table 4.1.

```
"name": "Screen Shader",
"version": "1.7.230",
 "manifest_version": 2,
 "offline_enabled": true,
"description": "Some extension.",

    "content_scripts": [
       { "matches": ["http://*/"],
         "js": ["scripts/content.js"],
         "run_at": "document_start",
         "all_frames": true   },       ],

"permissions": [
    "tabs",       "storage",       "cookies",   "idle",

   "webRequest",       "webRequestBlocking"],
                              ...
```

Figure 5.1: The figure shows an example of the manifest structure with permissions list and content script fields. Url match can be defined for each JavaScript in content_script, or match pattern can be defined in the permissions list. In the second case, this match will be applied for all content_scripts in a manifest file.

5.1 Extension content scripts

In the worker image part, we describe, that one of the jobs is to identify content_scripts and "chrome.tabs.executeScript" JavaScript files. We collect their source code, in case of execute script we collect nested JavaScript code inside of function call parameters, where we compute MD5 hash of them. These hashes are used to identify usage of the same execution code across different extension.

After our experiment, we collect 397,182 unique content scripts and 54,145 unique execute scripts. In Figure: 5.2 we present for each of content script we identify the number of extensions which use this certain JavaScript. Since we compute uniqueness of each source code with a hash function, that means all of them have the exact, byte to byte, same source code. The most popular of them

Key value	Number of seen
body	26,407
sizzle-+timestamp	19,484
sizzle+timestamp	14,538
head	5,643
script+timestamp	4,597
sizcache+timestamp	3,427
script	2,690
html	1,980
a	1,445
iframe	928
img	827
base	594
input	586
title	453
meta	403
gbqfq	314
video	297
div	235
form	223
object	221

Table 5.2: Top twenty common DOM query values based on the number of extensions that are looking for it.

was JQuery library, in the plot big part of unique content scripts are JQuery library where we identify different version/implementations. In same tame (Figure: 5.3) executeScript are not widely used by extensions.

We identify 1277 different extension crx files (275 different extensions and multiple versions of them), which are using specific toolbar content script created by MindSpark. Which is a marketing company, that focuses on the interactive advertisement, which receiving negative comment for their aggressive toolbar. One of the weird finding we have, that 6,629 of extensions use zero byte content script (file name in each of them are different), but for some reason this file is empty. Maybe that file was in the folder by mistake of a developer.

5.2 Extension permissions

Chrome browser creates different methods that extensions can use, but for control of what extensions is allowed to do and what extensions cannot do over browser

Category	# of extensions
With content script	322,900
Specific match	118,340
General match	100,533
With both matches	103,983
Without URL match	44

Table 5.3: Extension with content script statistics of matching URLs list.

Permission	# with query	# without query
content_scripts	51432	271468
tabs	37087	261692
storage	26466	261815
contextMenus	13465	85484
webRequest	13448	95717
notifications	13135	139263
cookies	12140	83092
webRequestBlocking	10514	77635
activeTab	9699	95232
webNavigation	7861	52749
unlimitedStorage	6764	83295
management	3910	43655
background	2830	25216
clipboardWrite	2599	18889
alarms	2527	42884
identity	1970	25676
history	1667	13797
downloads	1569	17169
bookmarks	1513	16334
idle	1421	11189

Table 5.4: Most popular permissions found in extensions/crx files with DOM query and number of them with same permission without any registered DOM query.

and user page. For this reason, all extensions contain manifest file see Figure 5.1, that describe permissions of extensions and what script will be executed for specific match URL pattern (content scripts). Also, manifest have been used for the description of extension developer, version of the extension and specific update features for example update url [10]. In our experiments, we more considered about extensions permissions and matching URL patterns.

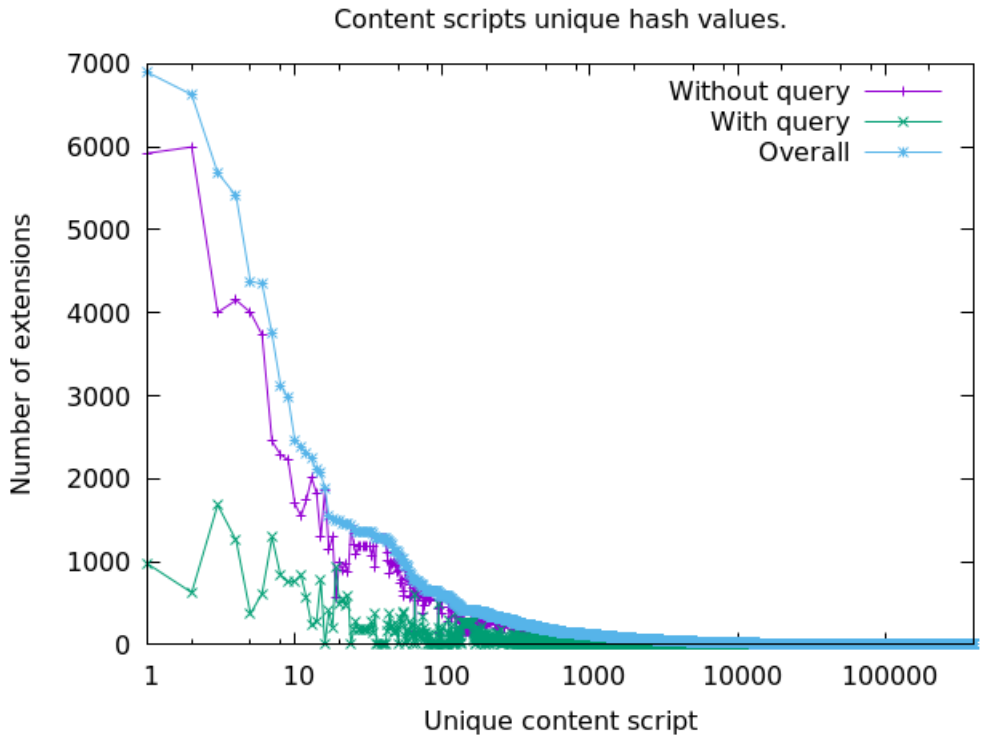

Figure 5.2: The uniqueness of the content script files that were defined in the manifest file for each extension. We compute for each unique content script three parameters: number of extensions that have registered DOM query, number of extensions that do not have any registered DOM query and summary of extension that generally uses this content script.

In-depth analyzing of collected data, we collect all DOM queries from extensions, and we measure for each of extension the number of Alexa 1 million websites where all these elements exist. Here is very important to mention that some of the extensions looking for elements that not exist in Alexa websites (Table: 6.1), so we cannot measure the number of websites with these elements and we ignore them. With all that said, we identify 9,883 of extensions/versions which perform DOM query for unique key values that we registered for not more than ten websites. In this group we notice wrong permission usage by 8,340 of them, in "content_script" field, these extensions define matching URLs with generic form of ('all_urls", "http://*/*", "https://*/*" or "*://*/*"). Such a URL matching technique is not necessary, [13] because they are targeting only a small amount of web pages. Beside that, in some cases even with script execution over all pages, they perform second matching over more strict URL list inside of JavaScript code. This type of permissions over usage can be reduced by analyzing extensions behaviour

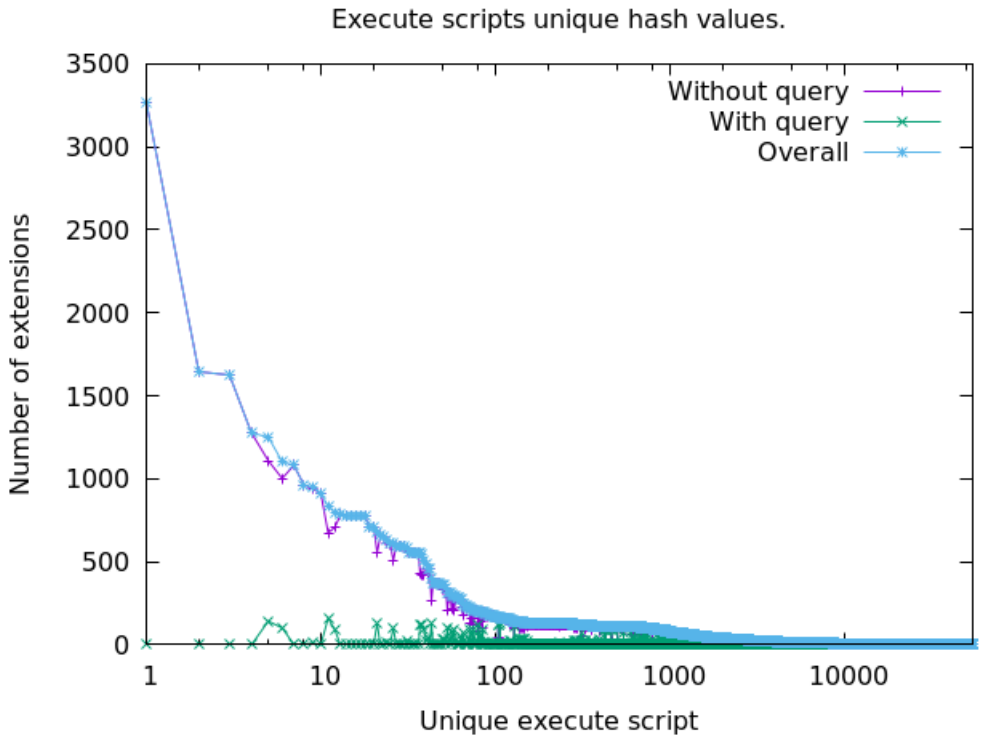

Figure 5.3: Figure present uniqueness of "chrome.tabs.executeScript" source code that we identify in all JavaScript in extension source. For each unique source code, we compute the number of extensions that have registered DOM query, number of extensions without registered DOM query and the overall number of extensions with this source code.

and DOM query analysis.

In same time 2,960 of collected extensions/versions that perform at least one DOM query are looking for key values that exist in more than 900k of Alexa websites.

We present the most popular extensions permissions in table 5.4. In this table, you can see that most popular field is content_scripts and tabs permission, also this field used by many extensions without registered DOM query, this shows us that a large amount of extensions/versions is allowed to execute their code over different matching URL patterns with permission to tab content.

Chapter 6

Results

During our analysis, we found extensions that contained in the source code long list of the different URL links. That information triggered our interest, so we manually analyzed the source code. One of the extension was very suspicious, it has a list of 57,440 URLs, and it was matching this list with visited URLs, and in the case when the user visited one of the websites from a list, the script executes "chrome.identity.getProfileUserInfo" that allows getting user e-mail and ID token. This extension submits URL, timestamp and the user token to their database by Ajax [14]. This extension is no longer available in Chrome Web Store. Not all extensions have a malicious purpose of URL usage; for example, we found one extension with multiple versions that contain a URL link to different radio stations or another extension is some Dev tool. Also, we were interested in identifying extensions that targeting specific unique IDs that only exist in a small amount of web site. While we analyzed them, we identify that these particular extensions were created to work only on a specific URL, for example, CancerTTV which looking for unique elements of twitch.tv and twitch.com and description justify that this extension is created specifically for a twitch web service. Another example of interesting extensions behaviour is F.lux for Chrome which seems to be matching tab URL and checking if the user already has this extension, and in the case when such extension not found, an advertisement with this extension will show off in Google search engine. We mentioned that we identify a group of extensions that targeting particular DOM elements, which exist not in more than ten websites.

Another finding from our experiments is that we collect a large amount of DOM queries which are not located on Alexa one million websites. For example, the most popular element is sizzle, this element as belonging to the Sizzle selector library, which creates extra functionality for the selection of CSS elements over the page. This library is used by the JQuery framework which allows to select DOM elements and manipulate them. We have such result, because of our patching method and we collect elements that any JavaScript in extension path use.

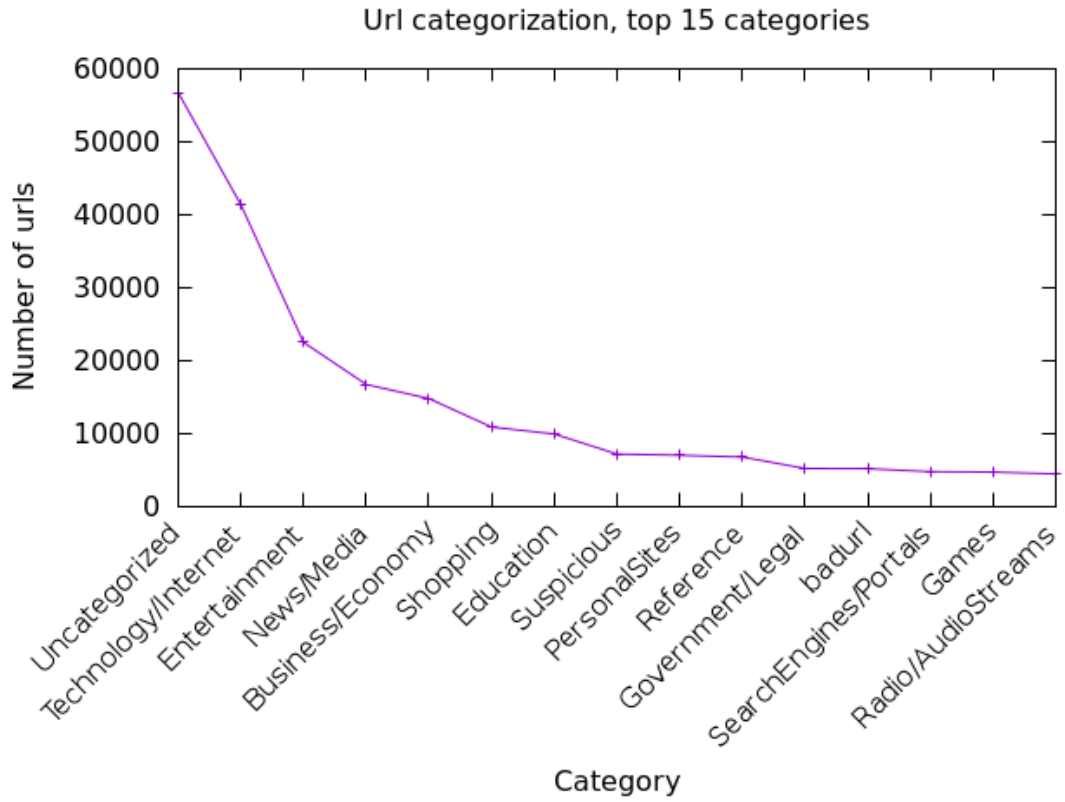

Figure 6.1: We identify 290,310 unique URLs in extensions JavaScript and HTML source code. Performing URL categorization with use of Symantec Bluecoat and Cisco Talos systems we generate this categorization figure, where a significant part of URLs are uncategorized.

From a security perspective, we identify extension, which interrupts the Facebook connection form. This particular extension, develop event handler on log-in action. Where user email and password values are collected by content script, and then redirected to attacker web database via a web request. When user credentials are stolen, the original Facebook log-in take place. While we analyzed permissions (Table: 5.3), results show us that some of the extensions do not have any matching URL in the content_script field. As we mentioned before we collect every crx file in chrome store, where some of the extensions are not unique, but they are only old versions of the same extension. In this particular case when a URL match does not exist, developer have a mistake in a specific version. In any case, this extension cannot be installed, because Chrome browser before loading the extension, perform validation checking of extension manifest.

key value	# of extensions
sizcache+timestamp	3427
add_to_chrome_container	173
producttitle	142
sw_html_pool	135
restorelink	123
sw_dashboard_url	86
sw_clb_extension_id	85
idadv	77
x-ng-app	69
gwdang_has_built	66
cherry-font	63
ra-querybar	61
repository_id	59
telephone_number_0	56
telephone_number_gv_0	52
telephone_number_gv_1	50
notification-install-extension	49
screenleapdiv	48
zbarholder	45
audio_search	45

Table 6.1: Top twenty values that extensions are looking for that not found in popular one million Alexa websites.

Chapter 7

Conclusion

Nowadays browsers offer many additional functionalities through the use of extensions and developers can share their extensions through the browsers' market. We implemented a sophisticated framework that can quickly crawl a vast amount of websites and analyze different versions of multiple extensions. We used a novel technique to cluster similar extensions based on the permissions they require and the JavaScripts they use. Our large scale analysis showed that extensions are able superstitiously to collect user data while some tend to advertise different products even though only 7% of the extensions perform the suspicious activity.

Working with so many extensions and large scale data make it more difficult to identify all type of techniques that malicious developer is possible to generate. The most important part of this work is to develop a method that can manage such a vast amount of extension where the first extension developed in March 2014 and the last one in February 2019. We manage to analyze almost five years developed extensions and version in less than a month. Also, it is essential to keep all data (extension, log files, Alexa DOM tree, DOM query, content scripts and execute scripts hash, manifest permissions) for future usage.

Interestingly we found cases were extensions have permission to download new JavaScripts from the web and use them at run time or inject such JavaScript directly into user page where it is challenging to separate if DOM query origin. In an effort to shed more light on this phenomenon we made our dataset publicly available.

For future work, we are interested in performing JavaScript clustering using different features and calculate code similarity based on the source code tree. Also, it is fascinating to identify the difference between similar executions and perform deep interaction with a page in order to collect accurate data which can be triggered by a specific page in the same domain.

Bibliography

[1] . Proxy - javascript — mdn. 2018. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`.

[2] A. Kapravelos, C Grier, N. Chachra, C. Kruegel, G. Vigna and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security Symposium*, 2014. .

[3] D. Martin Jr., R. Smith, M. Brittain, I. Fetch and H. Wu. The privacy practices of web browser extensions. In *Magazine Communications of the ACM*, 2001. .

[4] G.Varshney, M. Misra and P. Atrey. Browshing a new way of phishing using a malicious browser extension. In *Innovations in Power and Advanced Computing Technologies (i-PACT)*, 2017. .

[5] I. Sanchez-Rola, I. Santos and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX Security Symposium*, 2017. .

[6] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. Mccoy, A. Nappa, V. Paxson, P. Pearce, N. Provos and M. Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *IEEE Symposium on Security and Privacy*, 2015. .

[7] L. DeKoven, S. Savage, and G. Voelker. Malicious browser extensions at scale: Bridging the observability gap between web site and browser. In *CSET USENIX Workshop on Cyber Security Experimentation and Test*, 2017. .

[8] L. Liu, X. Zhang, G. Yan and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *Network and Distributed System Security Symposium*, 2012. .

[9] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson and E. Kirda. Ex-ray: Detection of history-leaking browser extensions. In *ACSAC Computer Security Applications Conference*, 2017. .

[10] N. Carlini, A. Porter Felt and D. Wagner. An evaluation of the google chrome extensions security architecture. In *USENIX Security Symposium*, 2012. .

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

[12] O. Starov, N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *World Wide Web International Conference*, 2017. .

[13] S. Heule, D. Rifkin, A. Russo and D. Stefan. The most dangerous code in the browser. In *USENIX Security Symposium*, 2015. .

[14] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens and W. Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *ASIA CSS*, 2014. .

[15] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci and W. Lee. Understanding malvertising through ad-injecting browser extensions. In *International Conference on World Wide Web*, 2015. .