



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Delve into Malware on Browsers

Finding related web content alterations between browser extensions

Master's thesis in Algorithms, Languages and Logic

JOEL MORIANA BECERRA

MASTER'S THESIS 2017

Delve into Malware on Browsers

Finding related web content alterations between browser extensions

JOEL MORIANA BECERRA



Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Delve into Malware on Browsers
Finding related web content alterations between browser extensions
JOEL MORIANA BECERRA

© JOEL MORIANA BECERRA, 2017.

Supervisor: Pablo Picazo-Sanchez
Examiner: Andreas Abel

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2017

Delve into Malware on Browsers
Finding related web content alterations between browser extensions
JOEL MORIANA BECERRA
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Providing the possibility of installing extensions has become a must-have feature for all major browsers. Extensions allow users to enhance and customise the browser functionalities by, for example, modifying the appearance of the web pages, providing security suites or blocking ads.

In this work, we make a first step towards monitoring web content alterations coming from extensions. In particular, we focus on the identification of relations between the mutations performed by different extensions. The study is motivated by the sequential and event-driven execution model running on web pages. That model entails that browser extensions can react to web content alterations performed by other extensions; hence, extensions have access to the data introduced by other extensions.

We implement our prototype as a couple of logging extensions running on a modified version of Chromium. The approach relies on dynamic analysis of extensions and a simulation of a user surfing the web. Our system is capable of automatically detect web content alterations performed by extensions and identify the events that triggered them.

We analyse the 150 most downloaded extensions from Chrome Web Store and characterise the most common alterations as well as the events that cause those mutations. Finally, although we did not detect direct relations between the extensions analysed, we discuss the alterations identified and the implications of the actual execution model.

Keywords: Web security, Browser extension, Web content alterations

Contents

1	Introduction	1
1.1	Context	2
1.2	Problem Definition	2
1.3	Aim	3
1.4	Method of Accomplishment	4
1.5	Limitations	4
1.6	Disposition	4
2	Background	5
2.1	Extension Architecture	5
2.2	Background Pages	7
2.3	Content Scripts	8
2.3.1	Script Injection	9
2.3.2	Event-Based Tasks	10
2.3.3	Timer-Based Tasks	12
3	Methodology	15
3.1	Delimitations	15
3.2	Approach Overview	16
3.2.1	Characterising Extensions	16
3.2.2	Finding Relations	17
4	Implementation	19
4.1	Web Browser Automation	19
4.2	Logging Extensions	21
4.2.1	Monitoring script injections	21
4.2.2	Monitoring event-based tasks	22
4.2.3	Monitoring timer-based tasks	23
4.3	Server	24
5	Results	27
5.1	Event Listeners	27
5.2	DOM Mutations	29
5.3	Interaction Between Extensions	30

6	Conclusion	33
6.1	Discussion	33
6.2	Future work	34
6.3	Summary	34
	Bibliography	35
A	HTML Tags	I
B	Blink Events	III
C	Blink Events Dispatched During The Analysis	VII
D	Analysed Extensions	IX
E	Detailed Results	XIII

1

Introduction

Nowadays, all major browsers vendors include the possibility of installing extensions. These small applications, which work within the browser session, provide additional features and allow users to enhance and customise the browser. For instance, extensions can modify the appearance of the browser, integrate web services or block ads. Typically written in JavaScript, HTML and CSS, extensions are developed by third parties and are not maintained by the browser vendor.

The advent of online stores to distribute extensions, such as Chrome Web Store or Mozilla Add-ons have contributed to their popularisation. Apart from central repositories, extensions are also accessible from local marketplaces and in combination with the installation of third-party software [16].

To perform their functionalities browser extensions have access to a privileged API, acquiring a great degree of control over the browser. For example, extensions can modify the web content through the *Document Object Model* (DOM), change HTTP headers or interact with sensitive data.

In this context, the usage of browser extensions poses new challenges in terms of security and privacy. Many malicious behaviours have been detected since the early studies. Kapravelos et al. [16] discuss some of them: (1) ad injection or manipulation to divert revenue from content publisher to the extension owner, (2) affiliate fraud to monetize extensions by defrauding major merchants, (3) sensitive information theft, or (4) online social network abuse.

Because the protection provided to the user is low, browser extensions have become a major vector of attacks. That fact encourages the aim of this thesis to study the extensions that can be installed on browsers from the security and privacy point of view. With this in mind, this work is focused on the identification of relations between the mutations performed into the web content by different extensions. To

address this problem, we propose a solution that automatically monitors the activity of an extension and simulate its actions to find possible interactions.

1.1 Context

Recently, most browser vendors have been focused on the development of semi-automated examinations to detect and eliminate malicious extensions from their repositories, reporting quite good results [15]. These centralised examinations are principally based on reputation scans of the publishers, static analyses of the code base and dynamic analyses that emulate common tasks performed in browsers. Although these approaches have been successful in identifying and removing many kinds of illicit behaviours (e.g., Facebook hijacking, ad injection or affiliate frauds), malicious code can be easily hidden during the short period of analysis time and go unnoticed [13]. For example, malicious extensions can avoid being detected by these analyses by adding constraints, such as time or location checks, to the execution of their actions.

In this context, other approaches have been suggested. For instance, on the side-client, Arshad et al. present ORIGINTRACER [10], a fine-grained approach to reliably determine the source of web content (i.e., identify ads injected by extensions). The approach is based on a provenance study at the level of individual DOM elements that in conjunction with visual indicators allow the user to distinguish publisher content from content that has been originated by third parties.

Similarly, Kapravelos et al. [16] present HULK, a dynamic analysis system that detects malicious behaviour in browsers extensions by monitoring their execution and corresponding network activity. HULK creates a dynamic environment that satisfies the needs of extensions to trigger all their functionalities and verify the classification. In the study 48,332 extensions were analysed (47,940 from Chrome Web Store), reaching the following results: 130 malicious extensions, 4,712 suspicious extensions and 43,490 benign extensions.

It should be stated that not many investigations have been carried out in the field. Nevertheless, some Android approaches can be extrapolated within the scope of this thesis due to the similarities shared between the ecosystems of Android and browsers (both are based on *appified* environments). In this context, Fratantonio et al. [13] present TRIGGERSCOPE, a pioneering prototype to detect sophisticated malware based on triggered activation by studying the application logic. This type of malware hides its malicious actions under certain (often narrow) conditions, making it hard to trigger them during the analysis.

1.2 Problem Definition

Browser extensions perform web content modifications in a sequential manner [17] by executing their activities in an event-driven environment (i.e., extensions are subscribed to events that trigger their actions). For example, extensions may be

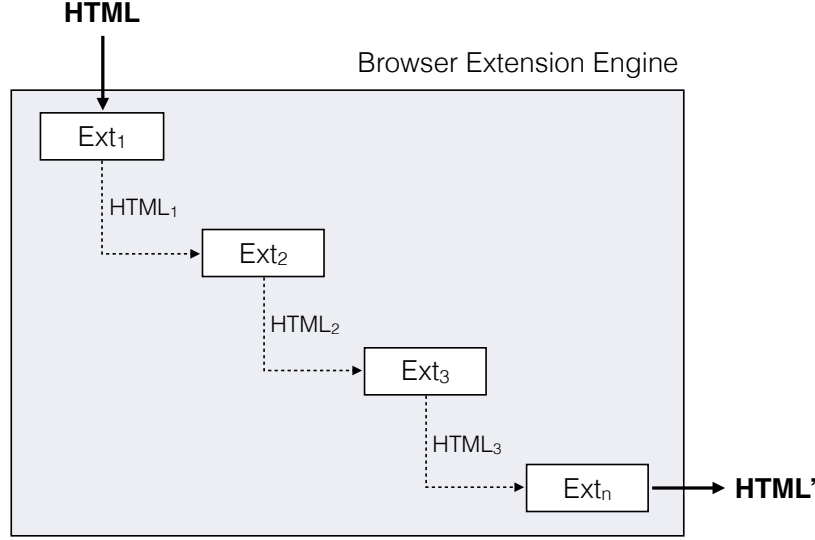


Figure 1.1: Browser extension engine [17].

triggered by visiting a specific website, by user generated events (such as keyboard or mouse events), or by modifications in the DOM structure among many others.

Figure 1.1 depicts an overview of the extension engine working in the web page context when an event is triggered. Note that the first extension takes the raw HTML from the browser, performs some actions, and passes the resulting HTML to the second extension. Therefore, this execution model reveals that browser extensions that execute later in the pipeline have access to all the information introduced by their predecessors.

This sequential execution model, combined with the event-driven environment, entails that an extension can react to DOM alterations performed by other extensions. By way of illustration, imagine two extensions running on the same browser session. Suppose that an *Extension X* removes ads from the web content and an *Extension Y* carries out translation functionalities. However, when the *Extension Y* detects that an ad has been removed from the web content, it modifies its “normal” output to inject the ad again. This situation depicts a relation between both extensions. As a result, an extension can perform its functionality correctly the major part of the time but modify its output under certain circumstances violating the expectations of the user.

1.3 Aim

The aim of this Thesis is to investigate the existence of relations based on DOM alterations between extensions running on the same browser. Particularly, we study if the alterations performed into the web content by one extension triggers another extension.

1.4 Method of Accomplishment

The proposed approach relies on dynamic execution of extensions and the monitoring of their activities. Specifically, we log the web content (i.e., HTML content) both before and after the extension executes. By retrieving the content modifications in the web pages, we can reliably determine relations between extensions.

The approach is carried out on Chromium, the open-source version of Google Chrome. This platform has been selected due to its popularisation (currently is the most used browser [4]) and the number of extensions available in its repository. Moreover, apart from being the base of Google Chrome, Chromium also provides source code to Opera browser [5]. In particular, we use the version 56.0.2924.76 of Chromium due to incompatibilities with newer versions of other packages needed for the implementation.

1.5 Limitations

The number of extensions to be evaluated is the main limitation. To this day, more than 140.000 extensions are available in Google Chrome repository [1]. Nevertheless, we do not analyse all of them due to the high time needed to examine a single extension. Specifically, we evaluate our solution against the 150 most downloaded extensions from Chrome Web Store; however, the set of extensions to study can be easily increased in future works.

1.6 Disposition

After this introduction, Chapter 2 introduces background information on Chrome extensions; Chapter 3 gives an overview of the proposed approach; Chapter 4 describes the prototype implementation of our solution, and Chapter 5 shows the experimental results. Finally, Chapter 6 presents the conclusions of this Thesis and indicates the future work.

2

Background

This chapter introduces background information on Chrome extensions presenting the basics of extensions architecture and focusing on the event-driven execution system.

2.1 Extension Architecture

Google Chrome extensions are composed of a set of JavaScript files, HTML files and any other resource needed by the extension (e.g., CSS or image files). JavaScript files contain the logic and the behaviour of the extension, while the HTML files define the content of the extension pages. A small number of extensions also include *native binaries* that can access the host machine with the user's full privileges [11].

Apart from these files, each extension contains a mandatory manifest file in which all the capabilities that the extension might use are indicated. The definition of this file responds to the spirit of limiting the number of resources accessible to the extension. For instance, extensions have to declare the set of `chrome.*` APIs that use in the manifest permissions section to have access to them. Despite the efforts to restrict the accessible resources, several investigations [11] [12] [14] claim that extensions are usually over-privileged and therefore needlessly increasing exploitable vulnerabilities. Apart from permissions, the manifest file also gathers the basic parameters of the extension, the list of resources that the browser should load and the set of web pages in which the extension might run. A complete list of the fields supported by the manifest file is gathered in the Manifest section of the Google Chrome Extensions Developer guide [2]. Figure 2.1 shows an example of a manifest file.

While building an extension, we have to differentiate between two contexts: (1) the extension context, in which background and other pages (such as popups) live,

```
"manifest_version": 2,
"name": "My Extension",
"icons": { ... },
...
"permissions": [
  "tabs",
  "webRequest",
  "<all_urls>"
],
"background": {"scripts": ["background.js"]},
"content_scripts": [
  {
    "matches": ["http://*/*", "https://*/*"],
    "js": ["jquery.js", "script.js"],
    "run_at": "document_start"
  }
]
...
```

Figure 2.1: Example of a manifest file.

and (2) the web page context, where content scripts operate. The developer makes use of the manifest file to indicate the context where each file has to be loaded (see Figure 2.1). Note that in conjunction with the content script files a set of web hosts (indicated as `matches` in the manifest file) has to be defined. In this way, content scripts will be only loaded into the web pages that fulfil the declared hosts.

Regarding the functionality, each context provides a set of distinctive features. On the one hand, background pages can access all the `chrome.*` APIs¹ that allow tight integration with the browser. On the other hand, content scripts are just like the scripts loaded by web pages and can use all the APIs that the browser provides to them (e.g., Standard JavaScript APIs, XMLHttpRequest, or HTML5) [3]. Although background and content scripts are isolated worlds, they can communicate to each other by exchanging messages through `chrome.extension` API. For instance, one can use message passing to perform an activity in the web page context as a result of an event occurred in the background. A typical use case consists in load scripts into the web page as a result of the user clicking the extension's icon. Figure 2.2 illustrates this architecture.

Both background and content scripts live in an event-driven environment (i.e., extensions performs activities in response to an event being dispatched). For instance, extensions may be triggered by visiting a specific web page, by user generated events (e.g., keyboard events), or by modifications in the DOM elements. Nevertheless, the manner in which each context is managed by Chrome differs widely. In the following, we present the peculiarities of both background and content scripts.

¹https://developer.chrome.com/extensions/api_index

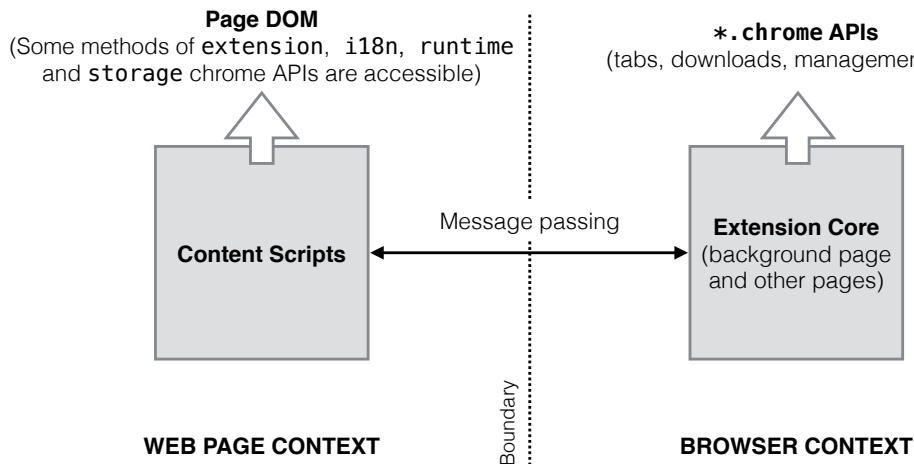


Figure 2.2: Extension architecture.

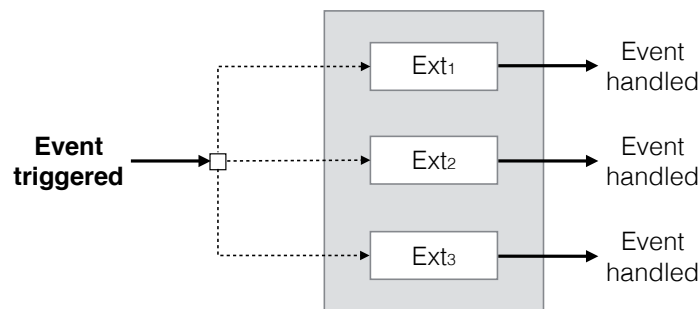


Figure 2.3: Example of how an event object is handled in background pages when three extensions are listening to it.

2.2 Background Pages

Background pages run in the thread of the extension, having each extension its thread (i.e., the execution of an extension does not block the execution of other extensions). Figure 2.3 shows background pages engine.

As stated above, background scripts can access to all the `chrome.*` APIs to perform their functionalities. Most of the methods of these APIs are executed asynchronous (i.e., they return immediately, without waiting for the operation to finish). A callback function passed into the scope of the method is used to know the outcome of the operation. Alternatively, other `chrome.*` methods are synchronous. Synchronous methods do not return until the work is completely done and therefore no callback function is required.

2.3 Content Scripts

In contrast to background pages, content scripts live in the thread of the web page, leading to a system in which only one block of code can be executed at a time. This feature is inherited from JavaScript engine. JavaScript engine has a concurrency model based on an event loop that is well known for having a single thread of execution. The event loop is managed by a user agent, and it is principally composed of (1) a call stack, (2) a set of task queues and (3) a microtask queue.

Call stack. The call stack is used to manage the execution of a block of code. In Javascript, the call stack is composed of frames. When a function is called, a frame, which encapsulates information about the called function (e.g., context and local variables), is added to the stack. As a result of a frame being processed, other frames may be added to the stack. The call stack works in a last-in-first-out model, and therefore the active frame (i.e., the frame being executed) is always the last which was added. When a frame is completely processed, the execution returns to the exact point of the previous frame in which the new frame was created. This process continues until the stack is empty.

Task queues. Tasks queues contain sets of functions to be executed. A task may be the callback of a timer, the callback associated with an event, or I/O operations among many others. JavaScript classifies tasks in different queues depending on their sources. For example, the source may be DOM manipulation, user interaction (e.g., mouse and keyboard events), or networking (i.e., tasks that trigger in response to network activity) [8]. As a result, task priorities can be established but always guarantying execution order within a source (task queues respond to a first-in-first-out model). Due to the nature of the call stack, only a single task can be executed at a time.

Microtask queue. Microtasks are tasks scheduled to happen in the most immediate future (i.e., straight after the currently executing script) [6]. In contrast to tasks, all the microtasks are collected in a single queue. Therefore, is not possible to establish preferences between them and the execution system only responds to a first-in-first-out model. The queued microtasks are executed as soon as the execution of the current task, if any, finishes. Until the microtask queue is not emptied (i.e., all the queued microtasks have been executed, even those queued as a result of the execution of other microtasks), the next task is not processed. For instance, `MutationObservers`² and `Promises`³ are microtasks.

The user agent of the event loop takes responsibility for coordinating the operations that are sent to the call stack. Apart from tasks and microtasks, the user agent also schedules when the rendering task of the web page takes place. The looping process is infinitely working and looking for new tasks to execute. That process is summarised in the next steps:

²<https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>

³https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise


```

while (eventLoop.waitForTask()) {
  const taskQueue = eventLoop.selectTaskQueue()
  if (taskQueue.hasNextTask()) {
    taskQueue.processNextTask()
  }

  const microtaskQueue = eventLoop.microTaskQueue
  while (microtaskQueue.hasNextMicrotask()) {
    microtaskQueue.processNextMicrotask()
  }

  if (shouldRender()) {
    render()
  }
}

```

Figure 2.4: Event loop pseudocode [9]

1. When the call stack is empty, the user agent looks for a new task to execute. If any, the task is chosen depending on source preferences and sent to the call stack.
2. Once the execution of the task has finished, the user agent looks for microtasks to execute. In contrast to tasks, which are executed one by one in each loop, all the microtasks queued are executed at this point.
3. If necessary, the user agent sends the rendering task to the call stack.
4. Jump to step 1.

A single iteration of this loop is called a tick. Figure 2.4 shows the pseudocode of the event loop model.

In the scope of this work, we focus on the monitoring of three different kinds of tasks: (1) the moment of injection of the script (i.e., the moment in which the script is loaded into the web page), (2) event-based tasks (e.g., keyboard events) and (3) timer-based tasks. Microtasks are left to be covered in future works.

2.3.1 Script Injection

Content scripts can be loaded in two different ways: (1) declaratively, indicating the script file in the manifest of the extension, or (2) programmatically, using the `chrome.tabs.executeScript()` method in the background page.

The way the developer uses to load the scripts establishes the time of injection. On the one hand, when loading the script declaratively, the developer can control the moment of injection through the `run_at` property (see Table 2.1). An example illustrating the usage of this property is shown in Figure 2.1. On the other hand, when the script is loaded programmatically, the developer can inject the scripts from the background page at any time (e.g., as a result of an event triggered in the background page).

document_start	The files are injected when the Document element has been created but after any files from CSS is loaded. Internally, Google Chrome loads the files when <code>DidCreateDocumentElement()</code> is triggered.
document_end	The files are injected just after DOM is completed but before any other subresource have been loaded (e.g., images and frames). Internally, Google Chrome injects the scripts when <code>DidFinishDocumentLoad()</code> is triggered.
document_idle	The files can be injected just at the same time as <code>document_end</code> or just after DOM is completely loaded (this includes subresources). The moment of injection depends on the complexity of the document and the necessary time to load it. Internally, Google Chrome tries to inject the files both when <code>DidFinishDocumentLoad()</code> and <code>DidFinishLoad()</code> are triggered.

Table 2.1: Possible values for `run_at` property and moment of injection.

It should be stated that while injecting declarative scripts from different extensions with the same `run_at` property, scripts are loaded by following the order of installation of each extension. Moreover, due to the event loop model, both in declaratively and programmatically methods, scripts are injected synchronously (i.e., the injection of the next script does not start until the previous script is not completely loaded).

2.3.2 Event-Based Tasks

Content scripts can attach event listeners to event targets (such as a Window, a Document, an Element in a document or any other objects that support events) using either vanilla JavaScript or any other third-party library such as jQuery. While registering an event listener, the developers indicate a callback function, which is the block of code executed when the event is fired. In particular, in vanilla JavaScript, one can make use of the method `addEventListener` to register and event to an event target. For instance, extensions can use the `beforeunload` event to intercept when a user is going to leave a web page. Nonetheless, the number of events that may be listened is limited. BLINK, which is the web render engine used by Chromium, provides 279 different events; however, custom events can be created and dispatched by the extension itself. See Appendix B for a table showing all the events provided by BLINK.

Different events occur in different objects. For instance, the `resize` event is only dispatched on the Window object, while the `DOMContentLoaded` is only dispatched on the Document object. However, some events propagate within the DOM

Capture phase	The event object propagates from the Window to the target's parent.
Target phase	The event object arrives to the event target. At this point, if the event does not bubble, the propagation will halt after completion this phase.
Bubbling phase	The event object propagates from target's parent to the Window.

Table 2.2: Event phases.

structure. For propagated events, a propagation path is computed when an event is triggered. This propagation path consists of an ordered set of targets through which the event will pass sequentially on the way to and back from the event target [7]. Altogether, event object propagates through three different phases: (1) capture phase, (2) target phase, and (3) bubbling phase. The explanation of each phase is gathered in Table 2.2.

Although the event flow always includes the three phases, the event is only handled in the target phase and in the capturing or bubbling phase. In this way, while attaching an event listener to an event target, the phase in which the event should be handled has to be selected. In vanilla JavaScript, the property that modifies this behaviour is `capture` and can be included as a parameter in the `addEventListener()` method. It should be stated that not all the events support bubble propagation and therefore, in these cases, that phase is skipped during the propagation path [7]. For instance, `error`, `focus` and `resize` events do not bubble. Moreover, event listeners can stop the propagation of the event object through the methods `stopPropagation` and `stopImmediatePropagation`.

Regarding the handling of an event, it is always processed sequentially and synchronously (i.e., until an event is not completely handled, the execution of the next event does not start). Even so, a distinction between synchronous events and asynchronous events should be stated. Synchronous events are those that are queued in a virtual list in a first-in-first-out model, ordered by temporal occurrence with respect to other events (mouse and keyboard events are synchronous). Alternatively, asynchronous events may be dispatched as the results of the actions being completed, with no relation to other events (i.e., asynchronous events are not queued in the virtual list of events). For example, `load` and `error` events are asynchronous.

When an event is fired, a new task is queued into the event loop to be executed. At the moment of the execution, the event listeners attached to this event are executed as soon as they are found in the propagation path. In this way, the event listeners registered in the capture phase will always be executed before the event listeners registered in the bubbling phase. If an event target has multiple event listeners registered in the same phase, event listeners are executed in the order they were attached (see Figure 2.5). Therefore, event listeners are in a virtual queue in a first-attached-first-handled model. It should be stated that the execution of an

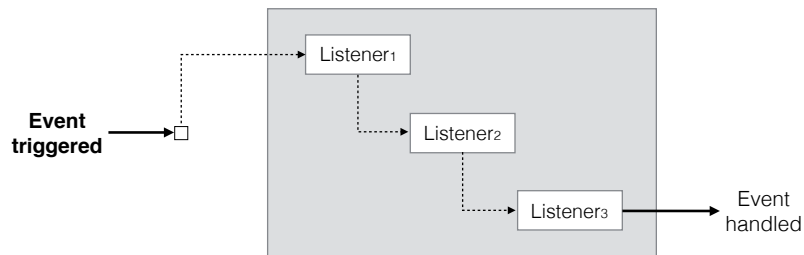


Figure 2.5: Example of how an event object is handled in web page context when three event listeners are listening to it

event may trigger other events. For instance, in response to the `DOMContentLoaded` event, an extension may inject a `<div>` element into the web content, triggering the `DOMNodeInserted` event. In that situation, these events do not generate an independent task but are executed synchronously as soon as they are triggered [8]. Figure 2.6 summarises the main keys of the event-based execution system that works in the web page context.

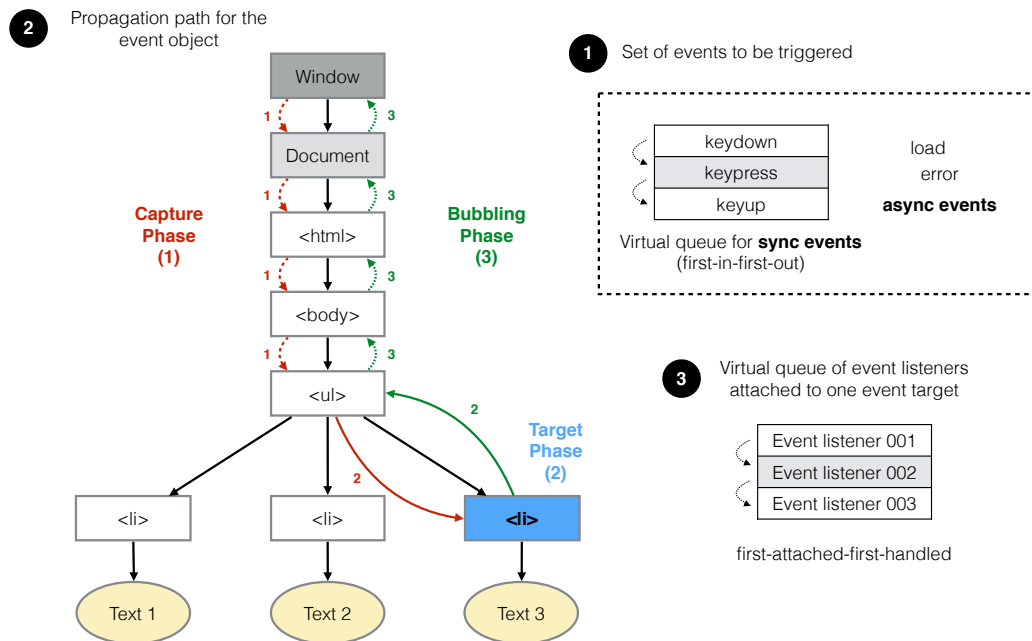


Figure 2.6: Event flow. Firstly, an event object is selected (❶). Secondly, if the event propagates, the event object flows through the propagation path (❷). For each object in the propagation path, all the attached event listeners are handled (❸).

2.3.3 Timer-Based Tasks

Developers can make use of timer events to perform actions after a specific period of time has passed. In practice, two kinds of timers can be generated: (1) one-

shot timers and (2) multiple-shot timers. On the one hand, one-shot timers can be injected via the `setTimeout` method and are only executed once. On the other hand, multiple-shot timers can be injected through the `setInterval` method and are executed infinitely, every time after the elapsed time expires. Both methods require the definition of a callback function, which is the block of code executed when the timer is fired.

When a timer is registered, a countdown starts. The expiration of that countdown implies to queue a new task into the event loop mechanism. Hence, the time defined while registering a timer is the minimum elapsed time and not the time that passes until the timer is executed (which will always be a major period of time). Multiple timers add tasks into the event loop every time the countdown expires.

3

Methodology

In this section, we describe the proposed methodology to identify relations between extensions. The approach relies on dynamic execution of extensions and the monitoring of their activities. Specifically, we develop a couple of logging extension that installed within the browser session and in conjunction with some modifications in BLINK rendering engine monitors changes in the DOM elements. Moreover, we automate the study of extensions simulating a user surfing the web. Using these techniques, we can retrieve content modifications in the web pages and reliably determine interactions between extensions.

Definitions. From this point forward the following nomenclature will be used:

- *Task*: We define a task as the execution of a block of code of an extension. In this work, we consider as a task the injection of a script into the web page and the callbacks associated with JavaScript events and DOM timers.
- *Action*: We define an action A_x as the set of mutations performed into the web content in the execution of a task.
- *Trigger*: We define a trigger T_x as the event that “activate” the execution of a task. In the scope of this work, the trigger may be the injection of a script, a JavaScript event, or a timer event.

3.1 Delimitations

In this project, we focus on the monitoring of declarative content scripts (i.e., scripts registered in the extension manifest file). Therefore, content scripts injected from the background pages via `executeScript` method are excluded. Along the same line,

message passing, which allows the performance of actions in the scope of the content script as a result of an event occurred in the background, are not tracked. Finally, content modifications by microtasks are neither monitored. To sum up, within the context of the web page, we track (1) the script injection, (2) the event-based tasks and (3) the timer-based tasks.

3.2 Approach Overview

To find related web content alterations between browser extensions, we need to respond two questions:

- *What does an extension do?*. Before analysing relations between extensions, we need to characterise the extensions according to DOM alterations. For that purpose, we monitor the activity of the extensions and gather all the mutations performed into the web content.
- *Are the extensions reacting to web content changes?*. By answering this question, we can find DOM alterations that trigger actions of other extensions. The characterisation of the extensions according to DOM alterations in the previous stage allows us to connect both parts and find relations between extensions.

In the following, we present a detailed description of each stage.

3.2.1 Characterising Extensions

To gather DOM alterations performed by an extension we track its tasks during a browser session. For that purpose, we carry out a system that takes responsibility for triggering and monitoring the tasks of an extension.

Triggering tasks. As discussed in chapter 2, extensions operate in an event-driven environment (i.e., extensions are registered to events that trigger their tasks), having the possibility of listening to different events; for example, an extension E_n may listen to `DOMContentLoaded` and `click` events. It should be remembered that not only JavaScript events but also the injection of the script may cause the execution of a task. The next set shows this situation.

$$\begin{aligned} & (E_1, T_1), (E_1, T_2), \dots, (E_1, T_3) \\ & (E_2, T_1), (E_2, T_2), \dots, (E_2, T_3) \\ & \vdots \\ & (E_3, T_1), (E_3, T_2), \dots, (E_3, T_3) \end{aligned}$$

Thus, to monitor the actions performed by an extension, we have to dispatch the events that trigger them. We automate the execution of those triggers by using Selenium, which allows the simulation of common tasks performed in browsers. In the next chapter, we present a detailed description of that simulation.

Monitoring tasks. Once a trigger has been dispatched, a couple of extensions installed within the environment of the browser session take responsibility for retrieving the HTML content either in the input or output of the task that is being handled. The way we monitor each type of task is detailed in the chapter 4. Figure 3.1 depicts the methodology followed when an event is triggered.

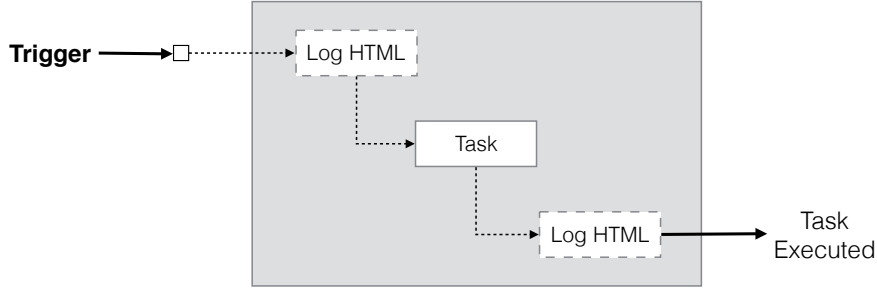


Figure 3.1: Log extension engine.

It should be stated that the actions performed by an extension may vary according to the event that triggered it. For instance, an extension may inject an `<iframe>` element when the `load` event is fired, but it may change an attribute of a `<div>` element when a `click` event is fired. This situation is depicted in the next set:

$$\begin{aligned}
 &(E_1, (T_1, A_1)), (E_1, (T_2, A_2)), \dots, (E_1, (T_3, A_3)) \\
 &(E_2, (T_1, A_1)), (E_2, (T_2, A_2)), \dots, (E_2, (T_3, A_3)) \\
 &\vdots \\
 &(E_3, (T_1, A_1)), (E_3, (T_2, A_2)), \dots, (E_3, (T_3, A_3))
 \end{aligned}$$

3.2.2 Finding Relations

In this stage, we excite the extensions by performing alterations into the web content. To simplify the analysis, we do not simulate the mutations gathered in the first stage but simulate all the possible mutations that an extension may perform. In that way, both stages are not dependent and can be executed at the same time. Identically to the previous stage, we track the extensions during the execution of their tasks, which allows us to identify mutations originated due to DOM alterations.

As a result of this stage, we can find triggers corresponding to actions that “activate” actions of other extensions. A relation between two extensions can be expressed as the next set:

$$(E_2, (E_1, (A_1, T_1)))$$

An alternative way of displaying this information is shown in Table 3.1.

Action \ Trigger						
	E_1	E_2	E_3	\dots	E_{n-1}	E_n
E_1	-	●	●	\dots	\times	\times
E_2	\times	-	\times	\dots	\times	●
E_3	\times	\times	-	\dots	●	\times
\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
E_{n-1}	\times	\times	\times	\dots	-	\times
E_n	\times	●	\times	\dots	\times	-

Table 3.1: Relation between extensions.

●=related; \times =not related

4

Implementation

In this chapter, we present our prototype implementation for identifying extension interactions. We implemented a dynamic analysis that comprises:

- A simulation of a user surfing the web as well as a simulation of all the possible DOM alterations.
- Two logging extensions that, installed within the environment of the browser session, take responsibility for retrieving the HTML content both before and after the execution of a task takes place.
- A server that receives the data retrieved by the logging extensions and computes the difference between HTMLs.

In the following, we provide more detail on the components that comprises our proposed solution.

4.1 Web Browser Automation

We simulate a user surfing the web to dispatch the events that trigger the extension's logic. The aim of this analysis is to fire the maximum number of events to “activate” all the possible functionalities of the extension. With this in mind, our analysis comprises two parts: a generic analysis and a specific analysis.

Generic Analysis. This analysis consists of the performance of a set of actions that are common to all the visited web pages. In it, we insert and remove one HTML tag of each type in the web content (see Appendix A for a table showing all the HTML tags used) as well as modify the attributes and character data of all

Google	Uses Google search engine and visits <i>wikipedia.org</i> with it.
Facebook	Tries to log into Facebook and tries to create a new account.
Amazon	Navigates the web and follows all the necessary steps to buy an item.
YouTube	Plays a video.

Table 4.1: Web pages visited during the analysis and principal activity performed.

of them. Note that these modifications allow us to simulate all the possible DOM mutations and therefore find interactions between extensions. Finally, other general actions (such as mouse movements, `resize` or `scroll`) are fired.

Specific Analysis. This part of the analysis consists of surfing a specific web page. Altogether we visit four of the most visited web pages¹. The web pages included in the analysis are Google, Facebook, Amazon and YouTube. Table 4.1 shows the principal activity performed on these web pages.

Altogether, the analysis fires 86 events out of the 279 events provided by BLINK (all the dispatched events are gathered in Appendix C). Although this analysis is a good starting point, we consider that further efforts are needed to extend it in future works. It should be stated that the only way to monitor an extension is being capable of triggering its logic. With this in mind, some tips considered for future works are: fulfil extensions needs via surfing the webs to which content script are registered and dispatch those events that the extensions attach during the analysis.

With respect to time, the analysis takes around 14-15 minutes to analyse a single extension. Our testing environment consists of a Ubuntu virtual machine with 8 GB RAM running on a Mac computer with 16 GB 1600 MHz DDR3 RAM and 3 GHz Intel Core i7 processor. This sandbox environment allows us to analyse the extensions without causing harm to the host machine.

We have implemented the simulation in Python, in conjunction with Selenium and PyAutoGUI libraries.

Selenium. This library allows the automation of the browser by interacting with CSS and HTML elements within a web page. Apart from common events (e.g., `click`, `dbclick` or `contextclick`), Selenium allows the usage of all the Standard JavaScript APIs via the injection of JavaScript code in the browser.

PyAutoGUI. This library is a GUI automation tool implemented as a Python module and used to control the mouse and keyboard programmatically. This library

¹Based on the ranking provided by Alexa Internet: <http://www.alexa.com/topsites>

allows us to execute events that Selenium does not trigger, specifically not HTML and CSS related events.

4.2 Logging Extensions

The logging engine is composed of two different extensions: `ADIVINOINI` and `ADIVINOFIN` that work together within the browser environment to monitor the activity of an extension. In particular, they take advantage of the event loop mechanism as well as some modifications introduced in `BLINK` to monitor (1) the script injection, (2) the event-based tasks and (3) the timer-based tasks.

Logging DOM changes. To monitor a task, we log the HTML both before and after the task is handled. However, as well as retrieving the HTML content, we also record a list with the mutations performed into the web page during the execution of the task. The mutation list is composed of mutation objects, which contains information about the alteration. Specifically, a mutation object gathers (1) the mutation type, (2) the node in which the alteration happened and (3) the parent of the node modified. Additionally, if the mutation consists in the modification of an attribute, the mutation object also contains the name of the attribute modified. That mutation list is useful for: (1) detecting changes that are done and undone within the same task and (2) retrieving easily all the information related to the mutation. To generate the mutation list accurately, we need to react synchronously to DOM mutations (i.e., gather the mutation just after it occurs). To this end, we have created a set of custom events that inform synchronously to the listeners attached to it about the DOM changes (see Table 4.2). As a result, we classify the mutations as (1) `NodeInserted`, (2) `NodeRemoved`, (3) `AttrModified` and (4) `CharacterDataModified`.

It should be noted that these events are only dispatched if the mutations alter elements that are part of the document. In this way, mutations that affect to elements that are not part of the document are not gathered.

In the following, we provide more detail on the methodology followed for the monitoring of each task.

4.2.1 Monitoring script injections

In this work, we only monitor content modifications by declarative content scripts (i.e., scripts registered in the extension manifest file). To monitor declarative script injection both `ADIVINOINI` and `ADIVINOFIN` load a script into the web page just before and after the extension of study injects its scripts.

Achieving the desired order of injection. As discussed in chapter 2, declarative scripts with the same `run_at` value are injected in the order in which the extensions they belong to were installed. Therefore, the desired order of injection can be achieved by installing the extensions in the proper order, i.e., (1) `ADIVINOINI`, (2) extension of study and (3) `ADIVINOFIN`.

CustomDOMNodeInserted	Fired when a new node is inserted into the document. We dispatch this event in ContainerNode class.
CustomDOMNodeRemoved	Fired when a node is removed from the document. We dispatch this event in ContainerNode class.
CustomDOMAttrModified	Fired when an attribute of a node is modified. The event includes modifications in the style of the node. We dispatch this event in Element and PropertySetCSSStyleDeclaration classes.
CustomDOMCharacterDataModified	Fired when the text of a node is modified. It should be stated that this mutation is only dispatched when the text of an existing node is changed, and not when a text is added to a node previously empty. We dispatch this event in the CharacterData class.

Table 4.2: Custom mutation events.

Monitoring all `run_at` values. As we do not know the exact moments in which the extension of study will inject its scripts, we load an `ADIVINOINI` and `ADIVINOFIN` script in each one of the possible moments (i.e., `document_start`, `document_end` and `document_idle`). As a result, all the possible moments of injection are covered.

4.2.2 Monitoring event-based tasks

To monitor event-based tasks, we place an event listener of our extension `ADIVINOINI` just before and after the event listeners registered by the extension of study. Due to the event loop mechanism, the event listeners of the extension of study will always be executed among our logging listeners.

Monitoring registration of event listeners. In order to monitor the registration of an event listener, we slightly modified the `EventTarget` class of `BLINK` to dispatch two new events: `beforeaddeventlistener` and `afteraddeventlistener`. These events communicate the exact moment in which an event listener is attached to an event target (Figure 4.3 gathers the information that encapsulates these events) and allow us to place the event listener of the extension of study among our logging listeners. It should be stated that this system not only works with `addEventListener` method but also with any other third-party library that allows

rTarget	A reference to the target in which the event listener is attached. May return a Window, a Document or an Element in a document.
rEventType	The name of the event that is attached.
rCapture	A boolean indicating whether the listener is registered in the capture or bubbling phase.

Table 4.3: Properties for `beforeaddeventlistener` and `afteraddeventlistener` events.

event registration, such as jQuery. Moreover, we also have added a `monitor` option in the `AddEventListenerOptions` class to control when these events are triggered. This is useful to avoid dispatching the events when `ADIVINOINI` and `ADIVINOFIN` are attaching event listeners.

Monitoring execution of event listeners. As discussed in chapter 2, the execution of an event may trigger other events that are handled synchronously. To avoid monitoring actions not performed by the extension of study but triggered because of its actions, we modified `EventTarget` class to inform about the initial and final moment in which the event listeners of an event target are handled. Specifically, we dispatch `beforehandlelisteners` and `afterhandlelisteners` to open and close respectively a monitoring window for those listeners. The monitoring window always starts disabled, and its state (enabled/disabled) is managed by the logging listeners that were previously registered. Figure 4.1 shows the monitoring engine for event-based tasks.

4.2.3 Monitoring timer-based tasks

Similarly to event-based tasks, we split the monitoring of timer-based tasks in two parts: (1) monitoring the registration of the timer and (2) monitoring its execution.

Monitoring registration of timers. To monitor the registration of a timer (either via `setTimeout` or `setInterval` methods), we modified `DOMTimerCoordinator` class to dispatch the `timerinstalled` event. This event is dispatched just before the timer is installed and contains the identifier i that BLINK assigns to it. Timers installed within an enabled monitoring window (i.e., timers registered by the extension of study) are gathered in a set of identifiers I .

Monitoring execution of timers. We slightly modified `DOMTimer` class to communicate the initial and final moment in which a timer is handled. Specifically, we dispatch the events `beforetimerexecuted` and `aftertimerexecuted` containing the identifier of the timer that is being handled. We monitor the execution of the timer if its identifier i belongs to the set of identifiers I .

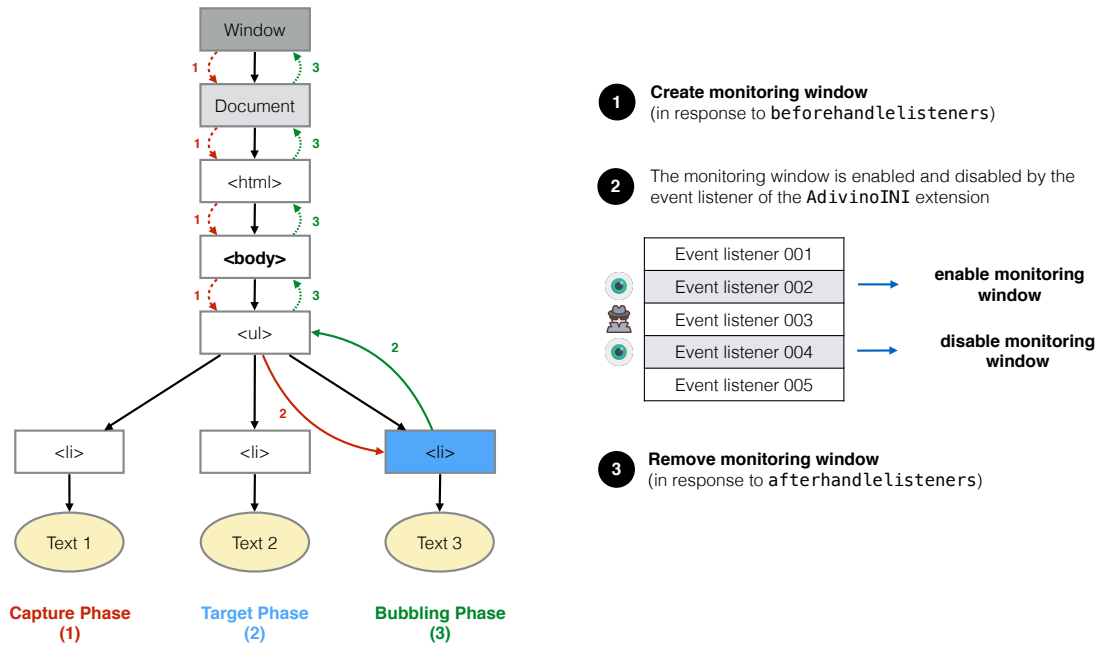


Figure 4.1: The example shows the monitoring engine for the event listeners attached to the body element when an event is triggered. We create a disabled monitoring window after the event listeners are handled (❶). The logging listeners (🕒) enable and disable the monitoring window (❷). While the monitoring window is enabled, we retrieve the DOM alterations performed by the extension of study (👤). We remove the monitoring window when all the event listeners attached to the target have been executed (❸).

4.3 Server

The data we send to the server fulfil two objectives: (1) communicate event listeners registered and (2) log the HTML content both before and after the execution of a task takes place.

Listener requests. We send a request to the server each time the extension of study attaches an event listener. Specifically, we send this request while handling the `beforeaddeventlistener` event. Table 4.4 shows the data contained in the request.

Record requests. Before and after a task is executed by the extension of study, we send a request to the server with the data shown in Table 4.5. To compute the difference, we intersect the list of mutations of the requests with the same `id` but different `sender`. Specifically, the `id` allows us to differentiate between tasks, and the `sender` parameter allows us to know which request belongs to the start of the task and which to the end.

url	A reference to the current URL in web browser.
target	A reference to the target to which the event listener was dispatched.
event	The name of the event.
capture	A boolean indicating whether the event uses capturing or not.

Table 4.4: Data in listener request.

id	Unique identifier of the event.
sender	A reference to the sender of the event. May be <code>ini</code> or <code>fin</code> .
event	The name of the event.
url	A reference to the URL in which the mutation was performed.
target	A reference to the target to which the event was dispatched.
current target	A reference to the currently registered target.
time stamp	The time at which the event was created, in milliseconds.
mutations	List of mutations records that contains complementary information about the DOM changes.
html	HTML content.

Table 4.5: Data in record request.

5

Results

This chapter presents the results of the analysis of the 150 most downloaded extensions from Chrome Web Store. See Appendix D for a table showing all the analysed extensions.

We organise this chapter as follows. Firstly, we present the events listeners registered by the extensions, analysing the target and phase to which they were attached. Secondly, we show the mutations introduced into the web content and the moments in which they were performed. Finally, we discuss the possible relations that the analysis reveals.

5.1 Event Listeners

In this section, we describe the event listeners registered during the analysis by identifying the type of the listener, the target to which was attached and the phase in which is handled.

Event type. A total of 381 different event listeners were attached by the extensions during the analysis. Table 5.1 shows the number of extensions that attach a specific event. The most commonly used, the `click`, `load` and `message` events, allows an extension to listen to an element being clicked, listen to an element being totally loaded and listen to data received from a server. The second most popular event, `DOMContentLoaded`, is fired when the DOM has been loaded and parsed but before any other subresource have been loaded (e.g., images and frames). It is worth pointing out the usage of custom events (6.67% of the extensions analysed use custom events to operate), which allow developers to fire their own events for their own particular needs. These events can be created by the `CustomEvent` interface and fired by the `dispatchEvent` method. Besides, more than a half of the logged event

Event	#ext.	Event	#ext.
click	17	auxclick	1
load	17	beforeunload	1
message	17	change	1
DOMContentLoaded	12	hashchange	1
keyup	11	mousemove	1
custom	10	mouseout	1
webkitvisibilitychange	10	mouseover	1
mouseup	9	mousewheel	1
error	6	paste	1
unload	6	popstate	1
contextmenu	5	resize	1
mousedown	5	scroll	1
focus	3	toggle	1
keydown	3	transitionend	1
visibilitychange	3	wheel	1
blur	2		

Table 5.1: Number of extensions that attach a specific event.

listeners were custom events (224 out of 381 listeners). In particular, we detected a great use of custom events in the extensions TamperMonkey, Honey, Ad Block and Ads Killer.

Event target We also computed the target to which these events were attached (see Table 5.2). These results reveal that around a 90% of the listeners are attached to elements that can be found in all the web pages. Specifically, the Window, Document, `<html>` and `<body>` objects contain a 92.34% of the targets. Attaching an element to a root element is easier than registering it to a specific element since the developer makes sure the element is contained in all the web pages. While choosing the object to which attach the event listener, we have to take into account that different events occur on different objects. Therefore, developers use the target that has the event in which they are interested in. On propagated events, which are dispatched either on Window or Document objects, the only difference is the timing. See Appendix E for a detailed table showing targets by event type.

Event phase. Table 5.3 shows the percentage of listeners attached to the capture and bubbling phase. The most used phase, the bubbling phase, is the default phase in vanilla JavaScript. Moreover, jQuery, which is a widely used library to interact with the DOM elements, does not allow the registration of listeners for the capture phase. The advantage of registering a listener for the capturing phase is that, in the pipeline, the listener is handled before the listeners registered for the bubbling phase. A detailed table showing the registered phases by event type is shown in Appendix E

Target	Pct (%)
Window	36.47
Document	35.88
<body>	14.71
<html>	5.29
<div>	2.94
<textArea>	1.76
<a>	1.18
<iframe>	0.59
	0.59
	0.59

Table 5.2: Targets to which listeners are attached

Phase	Pct (%)
Capture	16.47
Bubbling	83.53

Table 5.3: Phase to which listeners are attached

Finally, it should be mentioned the effect of the injection of jQuery into the web page as many extensions use this library instead of vanilla JavaScript. Specifically, the injection of jQuery implies the registration of the `DOMContentLoaded` event on the Document object and for the bubbling phase and the `load` event on the Window object and for the bubbling phase.

5.2 DOM Mutations

Regarding DOM changes, we have identified 20 extensions that perform mutations into the web content. Table 5.4 shows the percentage of mutations by type. The results reveal that modifications into the attribute of an element are the most performed mutation (50.47%), followed by the insertion of an element (31.66%) and the elimination of an element (17.87%). The analysis did not detect mutations in the character data of a node. It should be remembered that this mutation is only dispatched when the text of an existing node is modified, and not when a text is added to a node previously empty. The addition of new text to an element implies the creation of a `TextNode` element, and therefore a `NodeInserted` mutation is recorded.

Apart from the mutations performed, we also have identified the triggers that caused those mutations (see Table 5.5). Note that `init_start`, `init_end` and `init_idle` correspond to the injection of the scripts at `document_start`, `document_end` and `document_idle` respectively. The data shows that more than the 85% of the mutations were introduced as a result of the execution of a timer-based task, the handling of the `load` event and the injection of a script at `document_start`. Firstly, timer-based tasks allow developers to execute a block of

Action	Pct (%)
AttrModified	50.47
NodeInserted	31.66
NodeRemoved	17.87
CharacterDataModified	0.0

Table 5.4: Percentage of mutations by type

Trigger	Pct (%)
DOMTimer	33.23
load	29.47
init_start	23.2
init_idle	7.84
init_end	3.76
DOMContentLoaded	2.19
message	0.31

Table 5.5: Percentage of mutations by trigger

code after a specific period of time has passed. For instance, jQuery uses timers to dispatch some custom events (e.g., the event `ready` is dispatched certain time after the `DOMContentLoaded` event is triggered). Secondly, the `load` event is dispatched when an element is completely loaded. Finally, when a script is injected at `document_start`, only the Document object has been created, and therefore specific elements can not be modified. Mutations at this moment are addressed to root elements, such as the Window and Document objects. Table 5.6 shows the percentage of mutations by type and trigger. Appendix E gathers more specific information about the mutations identified.

As with the event listeners, jQuery performs changes into the web content during its injection. See Table 5.7 for a table gathering all the mutations performed.

5.3 Interaction Between Extensions

The analysis did not reveal relations between the extensions studied. In practice, an extension has to be listening to web content changes to be triggered by the mutations performed by another extension. Reacting to DOM alterations can be achieved by using (1) mutation events or (2) mutations observers.

On the one hand, the mutation events comprise the next set of events:

- `DOMNodeInserted`
- `DOMNodeInsertedIntoDocument`
- `DOMNodeRemoved`

AttrModified		NodeInserted	
Trigger	Pct (%)	Trigger	Pct (%)
DOMTimer	48.45	init_start	35.64
load	37.89	load	20.79
init_start	7.45	DOMTimer	19.8
DOMContentLoaded	3.11	init_idle	14.85
init_idle	1.86	init_end	5.94
init_end	1.24	DOMContentLoaded	1.98
		message	0.99

NodeRemoved	
Trigger	Pct (%)
init_start	45.61
load	21.05
DOMTimer	14.04
init_idle	12.28
init_end	7.02

Table 5.6: Percentage of mutations by mutation type and trigger.

Order	Mutation	Target	Parent
1	NodeInserted	<fieldset>	<html>
2	AttrModified	<fieldset>	<html>
3	NodeRemoved	<fieldset>	<html>
4	NodeInserted	<fieldset>	<html>
5	NodeInserted	<a>	<fieldset>
6	NodeInserted	<select>	<fieldset>
7	NodeRemoved	<fieldset>	<html>
8	NodeInserted	<fieldset>	<html>
9	AttrModified	<fieldset>	<html>
10	NodeRemoved	<fieldset>	<html>

Table 5.7: Mutations performed into web content when jQuery is injected

- `DOMNodeRemovedFromDocument`
- `DOMCharacterDataModified`
- `DOMSubtreeModified`

These events are handled synchronously as soon as the mutation happens. Note that attribute mutations can not be detected via the mutation events. The mutation events have been marked as deprecated in the DOM Events specification in favour of mutation observers.

On the other hand, mutation observers are implemented as microtasks, and therefore they were not tracked in this analysis. In contrast to mutation events, mutation observers allow tracking changes performed into the attributes of an element.

6

Conclusion

This chapter describes the conclusions made from the results of this Thesis. We organise this section as follows. Firstly, we discuss the results obtained and the possible implications of the actual execution model running on web pages. Secondly, we expound the potential future work. Finally, we summarise our main contributions.

6.1 Discussion

Out of 150 extensions analysed, we have identified 20 extensions that perform mutations into the web content, characterising the moments in which the mutations were made. On the one hand, we have detected that the modification of attributes is the most performed mutation, followed by the insertion of an element and the elimination of an element. On the other hand, we have found that timer callbacks, `load` event callbacks, and script injection at `document_start` tasks are the most used places to perform a mutation. With respect to event listeners, we have detected that most of them are registered to elements that are present in all the web pages (i.e., objects that always can be found in the DOM structure). Besides, the major part of them are attached for the bubbling phase.

Even though not relations were found, the execution model that operates within a web page make possible the existence of interactions between extensions. That execution model entails that extensions can react to DOM alterations performed by other extensions and therefore access to the data introduced. One might argue that this is an undesirable behaviour, since information— which might not be relevant for the task that other extensions perform—was disclosed. Thus, this can lead to a potential source of personal information leakage since extensions can intercept sensitive information embedded in web content alterations.

6.2 Future work

The most immediate future work is to extend the set of extensions analysed. Extending the analysis provides more reliable data, and relations between extensions may be found. Alternatively, improvements in the implemented prototype can be done. On the one hand, the analysis can be improved by fulfilling extensions needs via surfing the webs to which content script are registered and by dispatching those events that the extensions attach during the analysis. On the other hand, the monitoring of extensions can be extended by tracking the microtasks and the communication via message passing of content scripts with background pages.

Besides, other types of interactions between extensions can be studied. In this Thesis, we have focused on the investigation of “direct” relations (i.e., extensions that trigger other extensions as a result of a DOM alteration); however, extensions can access to the data introduced by other extensions in other ways. For instance, extensions not listening to DOM mutations can scan the web content in search of alterations as a result of other events being triggered. We find this study as a potential research area in which interesting results can be found.

6.3 Summary

In this work, an approach for identifying relations through web content modifications between extensions has been proposed. We have implemented our prototype as a couple of logging extensions running on a modified version of Chromium. Through a simulation of a user surfing the web as well as the simulation of all the possible DOM mutations, we have retrieved the event listeners registered by the extensions and the mutations performed. Specifically, we have analysed the first 150 most downloaded extensions from Chrome Web Store. Although not relations were found during the analysis, we have demonstrated that the nature of the environment where extensions operate make possible the existence of interactions between them. We also have characterised the event listeners attached by analysing the target and phase to which were registered.

Bibliography

- [1] Chrome extensions archive. <https://crx.dam.io>.
- [2] Google chrome extensions reference. a complete list of manifest file fields. <https://developer.chrome.com/extensions/manifest>.
- [3] Google chrome extensions reference. web apis. https://developer.chrome.com/extensions/api_other.
- [4] The most popular browsers. <http://www.w3schools.com/browsers/>.
- [5] Opera docs history. <http://www.opera.com/docs/history/>.
- [6] Tasks, microtasks, queues and schedules. <https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules>.
- [7] Web standard reference (events). <https://www.w3.org/TR/DOM-Level-3-Events/>.
- [8] Web standard reference (web application apis). <https://www.w3.org/TR/html5/webappapis.html>.
- [9] Writing a javascript framework - execution timing, beyond setTimeout. <https://blog.risingstack.com/writing-a-javascript-framework-execution-timing-beyond-settimeout>.
- [10] S. Arshad, A. Kharraz, and W. Robertson. *Identifying Extension-based Ad Injection via Fine-grained Web Content Provenance*. Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2016.
- [11] A. Barth, A. Felt, P. Saxena, and A. Boodman. *Protecting Browsers from Extension Vulnerabilities*. Proceedings of the Network and Distributed System Security Symposium (NDSS), 2010.
- [12] A. Felt, K. Greenwood, and D. Wagner. *Effectiveness of Application Permission*. Proceedings of the USENIX Conference on Web Application Development (WebApps), 2011.
- [13] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. *TriggerScope: Towards Detecting Logic Bombs in Android Applications*. Proceedings of the IEE Symposium on Security and Privacy, 2016.

- [14] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. *Verified Security for Browser Extensions*. IEEE Security and Privacy, 2011.
- [15] N. Jagpal, E. Dingle, J. Gravel, P. Mavrommatis, N. Provos, M. Rajab, and K. Thomas. *Trends and Lessons from Three Years Fighting Malicious Extensions*. 24th USENIX Security Symposium, 2015.
- [16] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. *Hulk: Eliciting Malicious Behaviour in Browser Extensions*. USENIX Security Symposium, 2014.
- [17] R. Pardo, P. Picazo-Sanchez, G. Schneider, and J. Tiapador. *A Runtime Monitoring System to Secure Browser Extensions*. HotSpot Workshop (to appear), 2017.

A

HTML Tags

Table A.1: HTML Tags.

a	center	footer	label	picture	table
abbr	cite	form	legend	pre	tbody
acronym	code	frame	li	progress	td
address	col	frameset	link	q	textarea
applet	colgroup	h1	main	rp	tfoot
area	datalist	h2	map	rt	th
article	dd	h3	mark	ruby	thead
aside	del	h4	menu	s	time
audio	details	h5	menuitem	samp	title
b	dfn	h6	meta	script	tr
base	dialog	head	meter	section	track
basefont	dir	header	nav	select	tt
bdi	div	hr	noframes	small	u
bdo	dl	html	noscript	source	ul
big	dt	i	object	span	var
blockquote	em	iframe	ol	strike	video
body	embed	img	optgroup	strong	wbr
br	fieldset	input	option	style	
button	figcaption	ins	output	sub	
canvas	figure	kbd	p	summary	
caption	font	keygen	param	sup	

B

Blink Events

Table B.1: BLINK events.

DOMActivate	midimessage
DOMCharacterDataModified	mousedown
DOMContentLoaded	mouseenter
DOMFocusIn	mouseleave
DOMFocusOut	mousemove
DOMNodeInserted	mouseout
DOMNodeInsertedIntoDocument	mouseover
DOMNodeRemoved	mouseup
DOMNodeRemovedFromDocument	mousewheel
DOMSubtreeModified	mute
abort	negotiationneeded
activate	nomatch
active	notificationclick
addsourcebuffer	notificationclose
addstream	notificationerror
addtrack	noupdate
animationend	obsolete
animationiteration	offline
animationstart	online
appinstalled	open
audioend	orientationchange
audioprocess	pagehide
Continued on next page	

Table B.1 – continued from previous page

audiostart	pageshow
auxclick	paste
availablechange	pause
beforecopy	paymentrequest
beforecut	periodicsync
beforeinput	play
beforeinstallprompt	playing
beforepaste	pointercancel
beforeunload	pointerdown
beginEvent	pointerenter
blocked	pointerleave
blur	pointerlockchange
boundary	pointerlockerror
bufferedamountlow	pointermove
cached	pointerout
cancel	pointerover
canplay	pointerup
canplaythrough	popstate
change	progress
characteristicvaluechanged	push
chargingchange	ratechange
chargingtimechange	readystatechange
checking	rejectionhandled
click	removesourcebuffer
close	removestream
complete	removetrack
compositionend	repeatEvent
compositionstart	reset
compositionupdate	resize
connect	resourcetimingbufferfull
connecting	result
connectionavailable	resume
contextlost	scroll
contextmenu	search
contextrestored	securitypolicyviolation
controllerchange	seeked
copy	seeking
crossoriginmessage	select
cuechange	selectionchange
cut	selectstart
dataavailable	shippingaddresschange
datachannel	shippingoptionchange
dblclick	show
Continued on next page	

Table B.1 – continued from previous page

defaultsessionstart	signalingstatechange
devicechange	slotchange
devicelight	soundend
devicemotion	soundstart
deviceorientation	sourceclose
deviceorientationabsolute	sourceended
dischargingtimechange	sourceopen
disconnect	speechend
display	speechstart
downloading	stalled
drag	start
dragend	statechange
dragenter	stop
dragleave	storage
dragover	submit
dragstart	success
drop	suspend
durationchange	sync
emptied	terminate
encrypted	textInput
end	timeout
endEvent	timeupdate
ended	toggle
enter	tonechange
error	touchcancel
exit	touchend
fetch	touchmove
finish	touchstart
focus	transitionend
focusin	typechange
focusout	unhandledrejection
foreignfetch	unload
frametimebufferfull	unmute
fullscreenchange	update
fullscreenerror	updateend
gamepadconnected	updatefound
gamepaddisconnected	updateready
gattserverdisconnected	updatestart
geofencecenter	upgradeneeded
geofenceleave	versionchange
gestureflingstart	visibilitychange
gesturelongpress	voiceschanged
gesturescrollend	volumechange
Continued on next page	

Table B.1 – continued from previous page

gesturescrollstart	vrdisplayactivate
gesturescrollupdate	vrdisplayblur
gestureshowpress	vrdisplayconnect
gesturetap	vrdisplaydeactivate
gesturetapdown	vrdisplaydisconnect
gesturetapunconfirmed	vrdisplayfocus
gotpointercapture	vrdisplaypresentchange
hashchange	waiting
icecandidate	waitingforkey
iceconnectionstatechange	webglcontextcreationerror
icegatheringstatechange	webglcontextlost
inactive	webglcontextrestored
input	webkitAnimationEnd
install	webkitAnimationIteration
invalid	webkitAnimationStart
keydown	webkitBeforeTextInserted
keypress	webkitEditableContentChanged
keystatuseschange	webkitTransitionEnd
keyup	webkitfullscreenchange
languagechange	webkitfullscreenerror
levelchange	webkitprerenderdomcontentloaded
load	webkitprerenderload
loadeddata	webkitprerenderstart
loadedmetadata	webkitprerenderstop
loadend	webkitspeechchange
loading	webkitvisibilitychange
loadingdone	wheel
loadingerror	write
loadstart	writeend
lostpointercapture	writestart
mark	zoom
message	

C

Blink Events Dispatched During The Analysis

Table C.1: BLINK events dispatched during the analysis.

DOMActivate	mousedown
DOMCharacterDataModified	mouseenter
DOMContentLoaded	mouseleave
DOMFocusIn	mousemove
DOMFocusOut	mouseout
DOMNodeInserted	mouseover
DOMNodeInsertedIntoDocument	mouseup
DOMNodeRemoved	pagehide
DOMNodeRemovedFromDocument	pageshow
DOMSubtreeModified	paste
animationiteration	play
animationstart	playing
auxclick	pointerdown
beforecopy	pointerenter
beforecut	pointerleave
beforeunload	pointermove
blur	pointerout
canplay	pointerover
canplaythrough	pointerup
change	progress
Continued on next page	

Table C.1 – continued from previous page

click	readystatechange
contextmenu	resize
copy	scroll
cut	select
dblclick	selectionchange
devicemotion	selectstart
deviceorientation	stalled
deviceorientationabsolute	submit
durationchange	textInput
emptied	timeupdate
error	transitionend
focus	unload
focusin	visibilitychange
focusout	volumechange
input	waiting
keydown	webkitBeforeTextInserted
keypress	webkitEditableContentChanged
keyup	webkitprerenderdomcontentloaded
load	webkitprerenderload
loadeddata	webkitprerenderstart
loadedmetadata	webkitprerenderstop
loadstart	webkitvisibilitychange
message	wheel

D

Analysed Extensions

Table D.1: Analysed extensions.

#	id	Name
1	chfdnecihphmhljaaejmgoiahnihplgn	AVG Web TuneUp
2	gighmmpiobklfepjocnamgkkbiglidom	AdBlock
3	cfhdojbkjhnklbpkdaibdccddilifddb	Adblock Plus
4	efaidnbmninnibpcapcglclefindmkaj	Adobe Acrobat
5	gomekmidlodglbbmalcneegieacbdmki	Avast Online Security
6	eofcbnmajmjmplflapaojjnihcjkgck	Avast SafePrice
7	flliilndjeohchalpbcbcdckjklbdgfk	Avira Browser Safety
8	jlmfmgmfgeifomenelglieghnjghma	Cisco WebEx Extension
9	mallpejgeafdahhflmliiahjdpgebepk	FromDocToPDF
10	ghbmnnjooekpmoecnnnilnbdlolhkh	Google Docs Offline
11	hcgmlmfclpfgljeaiahehebeoaiicbko	Google Photos
12	kbfnbcaeplbcioakkpcpgfkobkghlhen	Grammarly for Chrome
13	gpdjoidkbmbdfffahjcgigfpmkopogic	Pinterest Save Button
14	lifbcibllhkdhoafpjfnlhfpfgnpldfl	Skype
15	nafaimnnclfjfedmmabolbpcngeolgf	iLivid
16	dhdgffkkebhmkfjojejmpbldmpobfkfo	Tampermonkey
17	mppnoffgpafgpgbaigljliadgbnhljfl	Ask App for iLivid
18	gkojfkhlkighikafcpjkiklfbnlmeio	Unlimited Free VPN - Hola
19	mfhehppjhmmnnlfbopchdfldgimhfhfk	Google Classroom
20	aapbdbdomjkkjkaonfhkkikfgjllcleb	Google Translate
21	cjpalhdlnbpafamejdnhcphjbkeiagm	uBlock Origin
22	kfmjpgofbpmkihnammkhcoohnmipjkfjph	Social Color Changer for Chrome

Continued on next page

Table D.1 – continued from previous page

23	kpocjpoifmommoiiiamepombpeoaehfh	EasyPDFCombine
24	nckgahadagoaajggafhacjanaoiihapd	Google Hangouts
25	adokjfanafbkibffcbhihgihpigijcei	Share to Classroom
26	knipolnllmklapfncelgolnpehhpl	Google Hangouts
27	inoeonmfapjbbkmdafoankkfajkcphgd	Read&Write for Google Chrome™
28	ejidjjhkpkiempkbhmpbfngldlkgllhimk	Gmail Offline
29	hdokiejnpimakedhajhdcegeplioahd	LastPass: Free Password Manager
30	idkloemkmladbemijiamdiolojbffnjlh	G Suite Training
31	nmgcfemagnogdodbambjhdcmfcpicngl	Norton Safe Search as default for Chrome
32	okgjbfikeepgflmlglfgecmgjnmmnnb	WeVideo - Video Editor and Maker
33	glcimepnljoholdmjchklloafkggfoijh	360 Internet Protection
34	pioclpoplcbaefihamjohnefbikjilc	Evernote Web Clipper
35	mihcahmgcmnbcbchbopgniffhgnkff	Google Mail Checker
36	dpgdmhfocilnekecfjgimjdeckachfbc	Dropbox for Gmail
37	cmehdionkhpnaekndndgjdbohnhpckk	Adblock for Youtube™
38	bahkljjhdeciaaodlkppoonappfnheoi	Search Manager
39	phkdcinmmljblpnkohlipaodlonpinf	Поиск Mail.Ru
40	bmnlcjabgnpnenekpadlanbbkooimhnj	Honey
41	miijbmhjndcihicbljlcieiajhemmdcb	SuperBlock Adblocker
42	komhbcfkdcgmcdoenjcjheifdiabikfi	Google Play
43	kmljjoddjjkoidiahlgbgjgodcajhgf	Superblock Extended - Adblocker
44	mbckjcfnjmoinpgddefodcighgikkgn	AVG SafePrice
45	odijcgafkhpbjlnfdgiacpdenpmbgme	Домашняя страница Mail.Ru
46	bnbaboaihkhkjoaolfnfoablhlhahjee	GeoGebra Math Apps
47	obdbgibnhfcejmmpfijkpcihjeeedpfah	TypingClub
48	fdcgdnkidjaadafnichfpabhfomcebme	ZenMate VPN
49	ceopaldcnmhechacafgagdkklcogkgd	OnlineMapFinder
50	fhhbjgbiflinjbdggehcdcbncdddop	Postman
51	ejbdobdndcjdmljipngpeokdinlohe	Norton Home Page for Chrome
52	fkepacicchenbjecpbpbclokcabebhah	iCloud Bookmarks
53	mkaakpdehdafacodkkgkpghoibnmamcme	Google Drawings
54	ikgjlmllehlifdekcgaapkaplbdpje	VideoDownloadConverter
55	elicpjheidhpjomhibiffojpinpmpil	Video Downloader professional
56	heildphpnddilhkemkielfhnkaagiabh	Legacy Browser Support
57	lbfekoinhhcknnbdgnnmjhiladcgbol	Evernote Web
58	hehijbfgiekmjfkfjpbkbammjbdenadd	IE Tab
59	pbjikboenpfhbbbjgkoklgkhjpfogcam	Amazon Assistant for Chrome
60	mgijmajocgfcbeboacabfgobmjgjcoja	Google Dictionary (by Google)
61	icdipabjmbhpdkjaihfoikhjjeneebd	Kindle Cloud Reader
62	icmaknaampgiegkcjlimdiidlhopknpk	Pixlr Editor
63	bgnkhnnamicmpeenaenlfhikgbkllg	Adguard AdBlocker
64	kjagjnchnnlgiafjjlahaedeagnmhefi	uBlock Plus
65	mmeijimgabbpbgpdklnllpncmdofkcpn	Screencastify (Screen Video Recorder)
66	gmbmikajjgmabiglmofipeabaddhgne	Save to Google Drive
67	gbkeegbaiigmenfmjfcldgdpimamgkj	Office Editing for Docs Sheets & Slides
68	pdnfnkhpgegcpcingjbfihlkjeighnddk	Unblock Youku
69	ghgabhipcejjmhhchfonmamedcbeod	Click&Clean

Continued on next page

Table D.1 – continued from previous page

70	lpcaedmchfhocbbapmcbpinfpgnhiddi	Google Keep Chrome Extension
71	abjcfabbhafbcbdfjoecdgepllmfpcfief	Magic Actions for YouTube™
72	ioekoebejdcmnlefkjknokhhafglcjdl	Dropbox
73	jhknlonaaankphkkbnmjdlpehkinifeeg	Google Forms
74	honjcnefekfnompampcpmcdadibmjhlk	LanSchool Web Helper
75	niloccemoadcdkdjlinkgdfekeahmfj	Save to Pocket
76	aciahcmjmecflokailenpkdchphgkefd	Entanglement Web App
77	lalfiodohdgaiejccfgfmmngggpplmhp	Start Page - Yandex
78	laookkfknpbbblfpciffpaejjkokdgca	Momentum
79	omghfjlpggmjjaagoclmmbgdodcjbh	Browsec VPN
80	mlomiejdfolechcflejclcbmpeanii	Ghostery
81	bhdheahnajobgndecdbggfmcojekgdko	Desmos Graphing Calculator
82	ndjpnladcallmjembabefadecfnkepb	Office Online
83	fdjdjkkjoiomafnihnobkinnfjnnlhgd	Советник Яндекс.Маркета
84	cnkjkdjloflcpbemipjbcpfnglbgieh	Spotify - Music for every moment
85	klhphccnhdmlnljpdlljhehlmplnmini	Dell Activity Light
86	mcbkbpnkkipelfledbfocopglifcfmi	Poppit!
87	kbfmfpngjjgdlneeigpgjifpgocmfmb	Reddit Enhancement Suite
88	ocifcklkibdehekfnmflempfgjhbedch	Adblock Pro
89	pchfckkcldkbcldgdepkaonamkignan	Visual Bookmarks
90	ijjnmddphnlnelhbhefnfmimenjgbfc	MapsGalaxy
91	ppgplhcfmaadpnkmnkhgadmaekeldbnh	TelevisionFanatic
92	aiahmijlpehemcpleichkckheglfjl	Duolingo on the Web
93	fiombgjlkfdpkbhfioofeeinbehmajg	Word Online
94	jkfpcpilhkaemlpmpbebnlgkomamfeo	HP Network Check Launcher
95	menkifleemblindogmoihpfopnplikde	LINE
96	cnciopoikihiagdjbipnocolokfelagl	Videstream for Google Chromecast™
97	fdigbpdlfjdjelfdfocjhdkcpgmfcf	Unfriend Notify for Facebook
98	gmbgaklmjakoegefncnkhebmbhkjfic	Google Calendar (by Google)
99	bapebekcapehfapcilombbgepgedmmnm	SearchStart Tab
100	deeboegbjcnfgidliakhpoapnpomphji	uDev - Web Developer Toolbar
101	deceagebecbceejblnlcjooeohmmeldh	Netflix
102	oppjbdkgpfhlhllancffaoaemplhkgoc	Free Games Zone
103	gdbabpaggdgcakhjllleobffeghmhjme	Lucidchart for Education
104	mnehmlglkdbpcimikacjgegmpebacoab	Lightspeed User Agent
105	niojcggonafbneaajmkpkcigabaobmge	FilmFanatic
106	laddijjkcfpakbbnnedbhnnceicidncp	Yandex Search
107	bkgocclhfgjedhkiefaclppgbmoobnk	Audiotool
108	aaaaddliknddhjhjcofimffekgonpkom	Music Box
109	lfbgimoladefibpklnfmkpknadbklade	Webcam Toy
110	kbpnbonnhilfdihhodnflcplajklibbc	Yandex search
111	nlipoenfbbikpbjkfpcillcgkoblgpmj	Awesome Screenshot
112	anepigfegadpjpplagifaocofigipbjhn	네이버 Software 다운로드
113	jjckigopagkhaikodedjnmccfpmiea	Ads Killer
114	aiimdkdngfcipjohbjenkahhlhccpdbc	Flash Video Downloader
115	kbohafcopfpigkjdimdcdgenlhkmhbc	Hapara Highlights Extension
116	fngmhnnpilhploedifhcceomclgfbg	EditThisCookie

Continued on next page

Table D.1 – continued from previous page

117	dbaonaocldpohelilahfhnmjankmbcc	Orbitum Speed Dial
118	fdpohaocaechifimbbbbknoalclacl	Full Page Screen Capture
119	npmoikddpdgbhgbkkgjemncoegpojpng	MyTransitGuide
120	bfbmjmiodbnnplbbbfblcplfjjepjdn	Turn Off the Lights
121	fjnbnpbmkenffdnngjfgmeleoegfcfe	Stylish - Custom themes for any website
122	mjbepbhonbojpoaenhckjocchgfafo	Ace Stream Web Extension
123	dkpejdfnpdkhifgbancbammdijojoffk	Logitech Smooth Scrolling
124	gcbommkclmclpchllfjekcdonpmejbdp	HTTPS Everywhere
125	bigefpfhnfcobdlfbedofhhaibnlghod	MEGA
126	pfpeapihoiogbcmnmnibepnlkfnhoge	Outlook.com
127	mjcnihlhddpbdemagnpefmlkjdagkogk	Pocket
128	hbdpomandigafcibbmofojjchbcdagbl	TweetDeck by Twitter
129	iljnkagajgfdmfnidjijjobijlfjfgnb	Excel Online
130	bdehgigffdnkjpaindentkaniebfaepjm	MindMeister
131	knkapnclbofjjgicpkfoagdjohnlfjhp	Little Alchemy
132	pdabfienifkbhoihedcgeogidfmibmhp	Click&Clean App
133	ilmbnmigihncegckjgmkehgkdeohkhl	CK-12
134	ooadnieabchijkibjpeieelihjdijj	네이버 동영상 플러그인
135	beobeededemalmlhkmnmfembdimh	TV
136	hcmdpeobfoppdkhcneogcflmfceenlf	AdGuard Suit for Google Chrome™
137	mdafamggmaaaginoondinjkgecbpnhp	PowerPoint Online
138	opalpjboefohnelaemnhdlceibbcgl	Hola - Unlimited Proxy VPN
139	jjkofknkjdgkbbfdibgajealfbjhdj	Search By WowMovix
140	cmendinpapjjojakimjlmkkkcmnojeffg	McAfee SiteAdvisor Enterprise
141	oobklgpfnbcpokahmdidgbmlcdepkm	管家上防
142	chhjbpcepncaggjpdakmflnfcopglcmi	Ebates Cash Back
143	nbkekaeindpfpcoldfckljplboolgkfm	Video Downloader GetThemAll
144	chphlpgkkbolifaimnllloiipkdnihall	OneTab
145	mdanidgdpmkimeiojknlnekblgmpdll	Boomerang for Gmail
146	gjknpjjomckknofjidppipffbpoeikiipm	Betternet Unlimited Free VPN Proxy
147	ipmkfpcnmcccejididiaagpgchgjfajgp	Avira SafeSearch Plus
148	nffchahhjecejoiigmnhhicpoabngedk	OneDrive
149	nlbejmccbhkncgokjcmghpfloaajeffj	Hotspot Shield Free VPN Proxy
150	ghfmhofojkkfdnlfehkckbflohgiiicn	mixMovie Start

E

Detailed Results

Table E.1: Event targets by event type.

Event	Target
DOMContentLoaded	Document 71.43% Window 28.57%
auxclick	<body> 100.0%
beforeunload	Window 100.0%
blur	Window 50.0% <textarea> 50.0%
change	<div> 100.0%
click	<html> 36.36% Document 22.73% <body> 18.18% <a> 9.09% <div> 9.09% Window 4.55%
contextmenu	<body> 60.0% Document 40.0%
custom	Document 69.23% Window 23.08% <div> 7.69%
error	Document 83.33%

Continued on next page

Table E.1 – continued from previous page

	Window 16.67%
focus	Window 66.67% <textarea> 33.33%
hashchange	Window 100.0%
keydown	Window 66.67% Document 33.33%
keyup	<body> 72.73% Window 18.18% Document 9.09%
load	Window 70.0% Document 20.0% 5.0% <iframe> 5.0%
message	Window 100.0%
mousedown	Document 60.0% Window 40.0%
mousemove	Window 25.0% Document 25.0% 25.0% <html> 25.0%
mouseout	Document 100.0%
mouseover	Document 100.0%
mouseup	<body> 88.89% Window 11.11%
mousewheel	Document 100.0%
paste	Window 100.0%
popstate	Window 100.0%
resize	<body> 50.0% Window 50.0%
scroll	Document 50.0% <textarea> 50.0%
toggle	Document 100.0%
transitionend	<div> 100.0%
unload	Window 100.0%
visibilitychange	Document 100.0%

Continued on next page

Table E.1 – continued from previous page

webkitvisibilitychange	Document 100.0%
wheel	Document 100.0%

Table E.2: Event phase by event type.

Event	Phase
DOMContentLoaded	Bubble 100.0% Capture 0.0%
auxclick	Bubble 100.0% Capture 0.0%
beforeunload	Bubble 100.0% Capture 0.0%
blur	Capture 50.0% Bubble 50.0%
change	Bubble 100.0% Capture 0.0%
click	Bubble 81.82% Capture 18.18%
contextmenu	Bubble 60.0% Capture 40.0%
custom	Bubble 92.31% Capture 7.69%
error	Capture 83.33% Bubble 16.67%
focus	Bubble 66.67% Capture 33.33%
hashchange	Bubble 100.0% Capture 0.0%
keydown	Bubble 66.67% Capture 33.33%
keyup	Bubble 90.91% Capture 9.09%
load	Bubble 80.0% Capture 20.0%
message	Bubble 100.0%

Continued on next page

Table E.2 – continued from previous page

	Capture 0.0%
mousedown	Capture 80.0% Bubble 20.0%
mousemove	Bubble 75.0% Capture 25.0%
mouseout	Bubble 100.0% Capture 0.0%
mouseover	Bubble 100.0% Capture 0.0%
mouseup	Bubble 100.0% Capture 0.0%
mousewheel	Capture 50.0% Bubble 50.0%
paste	Capture 100.0% Bubble 0.0%
popstate	Bubble 100.0% Capture 0.0%
resize	Capture 50.0% Bubble 50.0%
scroll	Bubble 100.0% Capture 0.0%
toggle	Bubble 100.0% Capture 0.0%
transitionend	Bubble 100.0% Capture 0.0%
unload	Bubble 100.0% Capture 0.0%
visibilitychange	Bubble 100.0% Capture 0.0%
webkitvisibilitychange	Bubble 100.0% Capture 0.0%
wheel	Bubble 100.0% Capture 0.0%

Table E.3: Mutations by trigger.

Trigger	Mutation
DOMContentLoaded	 Attrmodified 28.57% <body> Attrmodified 28.57% <meta> Inserted 14.29% <div> Inserted 14.29% <div> Attrmodified 14.29%
DOMTimer	<a> Attrmodified 48.11% <div> Attrmodified 21.7% <div> Inserted 9.43% <link> Inserted 3.77% <div> Removed 3.77% <script> Removed 1.89% <td> Attrmodified 1.89% <html> Attrmodified 1.89% <script> Inserted 1.89% <table> Inserted 1.89% <table> Removed 0.94% Inserted 0.94% Removed 0.94% <Grammarly-Btn> Inserted 0.94%
init_end	<fieldset> Inserted 25.0% <fieldset> Removed 25.0% <fieldset> Attrmodified 16.67% <div> Removed 8.33% <select> Inserted 8.33% <a> Inserted 8.33% <div> Inserted 8.33%
init_idle	<div> Removed 28.0% <div> Inserted 28.0% <select> Inserted 12.0% <div> Attrmodified 12.0% <a> Inserted 12.0% <style> Inserted 4.0% <text> Inserted 4.0%
init_start	<script> Inserted 18.92% <script> Removed 14.86% <fieldset> Inserted 12.16% <fieldset> Removed 12.16% <fieldset> Attrmodified 8.11% <div> Removed 6.76% <div> Inserted 5.41%

Continued on next page

Table E.3 – continued from previous page

	<div> Attrmodified 5.41%
	<a> Inserted 5.41%
	<select> Inserted 5.41%
	<html> Attrmodified 2.7%
	<body> Removed 1.35%
	<body> Inserted 1.35%
load	<div> Attrmodified 58.51%
	<div> Inserted 15.96%
	<div> Removed 9.57%
	<td> Attrmodified 6.38%
	<table> Inserted 6.38%
	<table> Removed 3.19%
message	<div> Inserted 100.0%