2013 7th International Conference on Software Security and Reliability

A Model-Based Detection of Vulnerable and Malicious Browser Extensions

Hossain Shahriar
Department of Computer Science
Kennesaw State University
hshahria@kennesaw.edu

Komminist Weldemariam
School of Computing
Queen's University
weldemar@cs.queensu.ca

Thibaud Lutellier
Telecom Saint-Etienne
Universite Jean Monet
thibaud.lutellier@telecom-st-etienne.fr

Mohammad Zulkernine
School of Computing
Queen's University
mzulker@cs.queensu.ca

***Abstract*—Attacks such as XSS and SQL injections are still common in browser extensions due to the presence of potential vulnerabilities in extensions and some extensions are also malicious by design. As a consequence, much effort in the past has been spent on detecting vulnerable and malicious browser extensions. These techniques are limited to only detect either new forms of vulnerable or malicious extensions but not both. In this paper, we present a model-based approach to detect vulnerable and malicious browser extensions by widening and complementing existing techniques. We observe and utilize various common and distinguishing characteristics of benign, vulnerable, and malicious extensions to build our detection models. The models are well trained using a set of features extracted from a number of widely used browser extensions together with user supplied specifications. We implemented the approach for Mozilla Firefox extensions and evaluated it in a number of browser extensions. Our evaluation indicates 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, Vulnerabilities, Hidden Markvok Model, Malware.**

I. Introduction

Browser extensions can power-up the underlying browser to handle a variety of tasks on top of vendor's implemented functionalities. The ultimate goal is to make users' online presence easier and provide better browsing experiences, such as help comparing prices while shopping, managing browser tabs across multiple machines, and managing sensitive credential information. The current trend also shows a continuously increasing use of extensions for various services (see [1]).

A browser extension can also contain vulnerabilities leading to security breaches [2], [3], [4], [5], [6]. For instance, a vulnerable extension can accept inputs from web pages and can generate contents without applying any input validation. Once a vulnerability is found, an attentive attacker can exploit it by supplying a maliciously crafted program to gain access, e.g., of a victim's local storage system (such as cookie manager) and steal sensitive information without the knowledge of the victim. Further, some extensions are malicious by their design and can manifest themselves in many forms to attract traffic such that victims are lured to install a potential malware.

Although most extensions are thoroughly checked by a team of security professionals before they are hosted on trusted websites for distribution, various reports confirm that the prevalence of vulnerable and malicious browser extensions is on a continuous raise [2], [7], [8], [9], [10]. The trends also show that such extensions often go through existing checking mechanisms without being undetected. Notice that once an extension has been installed, it can enjoy the same privilege level as an administrator of a local machine, ultimately getting information access by reading, writing, and modifying local storage systems. Thus, it is important to ensure that they are checked for vulnerabilities and maliciousness before allowing to install them to mitigate (some of) the unwanted consequences.

In the fight against these, a number of automated analysis and detection techniques have been proposed. These include information flow analysis to check suspicious data flows from sources to sinks in vulnerable extensions statically (source code level [2]) and dynamically (browser runtime engine level [11], [12]). Some approaches restrict the privilege of APIs that could be invoked by malicious extensions [5], generate and validate digital signatures for benign extensions so that they can be checked during load time for the presence of malicious extensions [3], and monitor process and memory level activities against a set of behavior of benign extensions [4]. All these approaches are effective to detect either vulnerable or malicious extensions, but not both within the same framework.

In line with this, our goal in this paper is to design an approach for detecting browser extension types by widening and complementing prior works. To do so, we observe a set of common and distinguishing functionalities that benign, vulnerable, and malicious extensions can perform. The major difference between malicious extension and the others is usually in the visibility of the user interface and performing actions without user supplied events —e.g., clicking a button, clicking on a menu. Benign and vulnerable extensions can also be differentiated based on the presence of input filtering mechanisms.

We constructed three independent models for each extension type based on hidden Markov model (HMM) constructs [13]. The essential entities (e.g., state, observation sequence) of the models are built by utilizing the above characteristic

978-0-7695-5021-3/13 $26.00 © 2013 IEEE
DOI 10.1109/SERE.2013.32

of benign, vulnerable and malicious browser extensions and our prior experience. We then used a set of extensions (training samples), which are collected from Firefox extension sources to train the three models. Vulnerable and malicious extension samples are specifically difficult to find. Hence, we defined rules and applied them to generate additional training samples such that the detection would be more accurate and efficient. Once the models are trained, we evaluated our approach using randomly selected set of benign, vulnerable, and malicious Firefox browser extensions (test samples). Our approach detected all types of these extensions successfully. The approach was also able to detect previously unknown vulnerable and malicious extensions. We also implemented a prototype tool for Firefox browser extensions. Finally, as compared to other related work, the performance overhead of our approach is found to be negligible.

The paper is organized as follows. The next section provides some background information. We present our approach for detecting extension types in Section III. Section IV discusses our benchmark and experimental setup. In Section V, we present the experimental evaluation. While Sections VI presents some related work, we conclude and discuss potential future work in Section VII.

II. Browser Extension Layout and Types

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, the intended purpose, the creator name and home page, the URL to obtain further information about an extension, and the supported browser version numbers [14].

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 the file path, which is overridden to the browser's default elements —i.e., `overlay chrome://browser/content/browser.xul chrome://XYZ/content/browser.xul`, where XYZ is the name of an extension. It 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 (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 invocation of JavaScript code based on user events (e.g., button click).

```
<toolbox flex="1">
<menubar id="menu1">
    <menu id="file-menu" label="File">
        <menuitem id="New" label="NewFile"
            oncommand="createFile();"/>
    </menu>
</menubar>
</toolbox>
```

Listing 1: A code snippet of XUL (adapted from [15]).

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  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 vulnerable JavaScript code snippet implementing `createFile` method. The vulnerability is present at Line 5, which displays a user supplied input from Line 1 (i.e., *fname*) to the browser without sanitization. We noted previously such code present in a browser extension has the administrative privileges to access local file systems, cookie managers, password managers, bookmark mangers, history managers, and DOMs of all open web pages that may contain sensitive personal information. By taking this advantage and the vulnerable code, an attentive attacker can provide an arbitrary JavaScript code to launch potential attacks successfully.

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

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

Similarly, 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 uses an `eval` method with an argument that creates an XPCOM object [16] instance to access the local

file system 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 a user expected functionality and the actual functionality.

The `createFile` method (shown in Listing 2) could be malicious where the actual operation performed does 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 is obfuscated [3].

```
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).

Browser extensions are increasingly being downloaded and used by end users. For instance, as of July 28 of 2012, more than two billions extensions have been downloaded only for Mozilla Firefox browser [17]. As demonstrated above and elsewhere, the seemingly benign extensions may contain vulnerable code which could be exploited by malicious users while introducing unwanted 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 unwanted activities without their knowledge. Thus, an approach that can detect vulnerable and malicious extensions automatically would prevent users from installing these extensions in their browsing environment.

III. Model-Based Detection Approach

In this section, we present an approach to detecting browser extension types by utilizing the hidden Markov model (HMM) [13] constructs to describe the behavior of browser extensions types based on extracted features and supplied specifications.

Figure 1 shows an overview of our approach. The first phase is essentially feature collection and training step. The feature extraction engine extracts potentially relevant features for constructing our detection 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. The models are then trained using our 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.

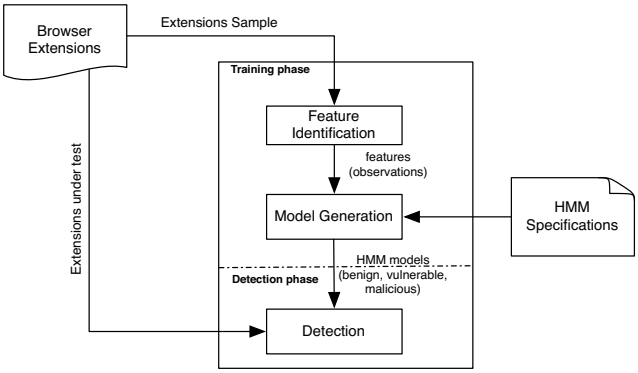

Figure 1: Overview of our approach.

In the detection phase, the generated models are used to detect unknown browser extensions. To do so, we extract features for an unknown extension and feed them to the detection models. If the benign HMM 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.

A. Feature Identification

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. In the following, we describe some of these features (the core to construct our detection models).

Benign, vulnerable, and malicious extensions can create, read, and write local machine and browser specific resources based on a set of method calls (API or XPCOM interfaces). An API invocation may or may not be related with user interactions. 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 (lack of sanitization function). A malicious extension may perform operations without user generated events and access and leak

Table I: Characteristics of benign, vulnerable and malicious browser extensions. We capture these characteristics by analyzing a number of samples from benign, vulnerable and malicious extensions types. Some characteristics overlap among the types.

Feature	Type	Sample Characteristics
v_0	Benign	Visible interfaces are present, user events initiate functionalities, APIs render web contents with input filtering, APIs access to browser and local storages (cookie, bookmark, preference, file system) for reading and writing, generated web requests supply information to white listed websites.
v_1	Vulnerable	Visible interfaces may or may not be present, user events may or may not initiate intended functionalities, APIs render web requests without input filtering to websites that are not white listed or related to the browser extension vendor's website.
v_2	Malicious	Visible interfaces are not present, user events do not initiate intended functionalities, APIs may or may not render web contents with input filtering, APIs render web requests to unknown websites (not included in whitelisted website), APIs generate new tabs based on information obtained from remote websites, APIs access to local storages (cookie, bookmark, preference, file system) 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 render web requests to white listed websites or to the browser extension vendor's website.
v_4	Vulnerable or Malicious	Visible interfaces may or may not be present, user events may or may not initiate intended functionalities, APIs render web requests without input filtering to websites that are not white listed or related to the browser extension vendor's website.

sensitive resources. 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 may be present in benign, vulnerable, and malicious extensions.

Based on the above characteristics of browser extensions, some heuristics in the literature (e.g., [3], [5], [18]) and based on our own investigation of browser extensions, we consider six features sets (see Table I) that can best characterize benign, vulnerable and malicious browser extensions. We also combine features such as visible interfaces in browsers, perform functionality based on user generated events, accept of inputs, input filtering or sanitization, and API call sequences.

Let $V = \{v_0, v_1, v_2, v_3, v_4\}$ be the set of all the features that can model benign, vulnerable, and malicious browser extensions shown in Table I. The features in v_0 are defined in way that benign extensions will have more occurrence probabilities followed by v_3. 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_1, also some properties in v_4. The properties in v_2 mostly exhibit the behavior of malicious extensions with some overlapping properties from v_3 and v_4. However, malicious extensions rarely contribute to the occurrence probability of v_0 and v_1.

To extract most of the above features, we rely on standard JavaScript APIs mainly by checking web content rendering functions (e.g., `document.innerHTML`, `document.write`) and input filtering functions (e.g., `document.replace` and `document.search`). For the two examples of input filtering functions, for instance, we check for meta characters (such as "<" and ">") in the target and search elements, respectively. This way the occurrence of a content generation function followed by any filtering function is identified.

B. 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 an HMM. A typical HMM consists of distinct states of the Markov process, state transition probabilities, a probability distribution for all possible output features with respect to each state, and initial state probabilities (details of core HMM can be found in [13]).

We represent the three distinct states of a given browser extension can be with S, $\{benign, vulnerable, malicious\}$ = $\{s_i\}$. The HMM for each type consists of a state transition probability matrix A —where each entry of the matrix has the form $a_{ij} = P(state\ s_j\ at\ t+1 | state\ s_i\ at\ t)$; an observation probability matrix B that describes the probabilities of recording different observation features given that an extension is in one of the states —i.e., each observation $b_j(m)$ is $P(observation\ m\ at\ t \mid state\ s_j\ at\ t)$; and, an initial state distribution π.

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, using the HMM terminology) 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 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 $<1, 1, 1, 1, 1, 1>$ will have 0.2 probability distribution for each feature.

To count the frequency of $v_i \in V$, we look the coincidence 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 elements of the feature v_0. Here, sub-features are the individual element (e.g., visible interfaces, user events) of v_0. We specifically scan the extension code (XUL, JS files) to count the frequencies. For our previous example (see Listing 1), we count the number of occurrences of `menuitem` for visible interface; the replace method calls in JavaScript files (to relate input filtering API, specifically to see how many APIs are using "<" or ">" to prevent attacks related to Cross Site Scripting); the `XPCOM` calls to access cookie, bookmark or preference (e.g., `nsICookieService` in JavaScript files); and, the number of web content rendered by counting the number of `innerHTML` on the left side of statements (e.g., `document.innerHTML(...) = ...`). The presence of the occurrence of all of the sub-features, which can span over multiple files, is also counted. This way, the three core models are designed and are ready to be trained.

C. HMM-based Training and Detection

1) Model Training: 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, which are subjected to the row stochastic condition.

For our purpose, we utilize the algorithm presented in [19] to train the three models. The 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 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 probability 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 maximize the $P(O)$, given the number of states (N = 3), observation symbols (M = 5) and the observation sequence (O).

2) 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 compute the occurrence probabilities for the observed features from 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.

IV. Benchmark and Training Setup

For validating our approach, we implemented the presented approach in a prototype tool on top of existing tools. More specifically, 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 Java Hidden Markov Model library [20]. 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.

A. Benchmark

The accuracy of the generated HMM models for each extension type depends on the size of a training dataset — i.e., a larger training dataset could result a more accurate detection model. For benign type, there exist a large number of extensions available in trusted websites (e.g., [21]). However, the number of vulnerable and malicious extensions that we have found in real-world is very few. We believe that

Table II: Rules we defined to enlarge our vulnerable and malicious extensions set.

Intended Extension type	Rule	Description
Vulnerable	R1	Assign the content of $innerHTML$ 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.
	R6	Access password and 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 and bookmark mangers and open new windows with the obtained URLs.
	R8	Delete a file from local disk randomly.

one of the reasons for this is that, vulnerable extensions are subjected to quick-fixing-release trend by the corresponding authors once discovered, ultimately removing the vulnerable versions. Moreover, those websites which host malicious extensions are often taken down 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 II. Specifically, we injected (similar work can be found in [2], [11], [22], [23]) four vulnerabilities based on rules $R1$ to $R4$. The 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 patterns.

We also created malicious extensions by adding an `onload` event handler in XUL files. The event handler randomly invokes a number of actions (following rules $R5$ to $R8$ shown in Table II) based on known attacks performed by malwares [24]. These vulnerable actions include *i*) changing of the default browser setting (e.g., change home page entries), *ii*) accessing 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.

B. Training Setup

Our initial HMM models were trained using 38 real-world Firefox extensions. To balance the mix of extension types, we considered 4 distinct classes of browser extensions as shown in the "Category" column of Table III. Examples from each category is also shown in Table IV. For each extension, first we analyzed the number `XUL` files, the number of visible interfaces (e.g., menus, buttons with event handler), and the number of JavaScript files. Second, the number of `XPCOM` APIs for all extensions in each category is evaluated. Finally, we evaluated the number of `window.open`, `search`, `replace`, `innerHTML`, and `eval` method calls in JavaScript code for each of the benign extensions.

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

Category	# of extensions
C1: Language & Support	10
C2: Photos, Music & Video	10
C3: Security & Privacy	10
C4: Social & Communication	8

During our experiment, we noticed that all the extensions generate visible user interface, except one extension (`Easyaccent`) from category C3 and two extensions (`Facebook_dislike` and `Youtube_video_replay`) from category C4. Our code analysis also revealed that these extensions include custom generated user events in JavaScript. All the extensions applied `nsIPrefService`, which is used to set the default values associated with these extensions.

Most of the experimented extensions have accessed local file system (`nsIIOService`), cookie (`nsICookieService`), RDF data source for reading and writing to RDF files (`nsIRDFService`, `nsIRDFContainer`), and windows open in the browser (`nsIWindowMediator`). 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. Moreover, only three extensions use `eval` method calls of which one is from category C2, and the remaining two are from C3 category.

We also generated vulnerable and malicious training datasets using the rules discussed in Table II. This allows us to alter the observation probabilities widely across extensions.

V. Evaluation

To assess the effectiveness of our approach in detecting browser extension types using the trained models, we tested on twenty real-world Firefox extensions (see Table V). Initially, all these extensions are assumed to be benign. However, other related work (e.g., [2], [3]) identified these ex-

Table IV: 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
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
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
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
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 V: The detection evaluation results for the twenty 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**
Facebook_dislike-3.0.2	**Benign**	**0.82**	**0.35**	**0.95**	**Malicious**
Facebook_Rosa	Malicious	0.45	0.5	0.76	Malicious

tensions as vulnerable. In particular, `Wikipedia`, `Fizzle` (Versions 0.5.1 and 0.5.2), and `Beatnik` have been reported to be vulnerable in [2]. The vulnerability is due to the lack of sanitization before generating contents from untrusted sources (RDF) to web page DOM node (`innerHTML` method call). Similarly, the `Facebook_Rosa` extension has been reported to be malicious in [25]. In fact, this extension is a known malware that allures a user to download an extension due to the displaying of a video in a web page. Once downloaded, the extension sends link of the page to other friends of the victim account. Here, the feature is to have a link of an extension in a web page, and installing this extension results in sending requests to other users automatically.

We extract the observation features from each of the extension and compute the observation probabilities for the three models. Columns 3 to 5 (see Table V) report the observation probabilities of features with respect to the three models. In all cases, our approach detected the benign, vulnerable, and malicious extensions. The approach also detected `Budaneki` and `Facebook_dislinke` extensions as vulnerable and malicious, respectively (the **bold** rows in Table V). 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;
```

Similarly, the snippet code in Listing 5 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

Table VI: Comparison of our detection with the related work.

Extension	Type	Our Approach	VEX [2]	Louw et al. [3]
Wikipedia Toolbar-0.5.9	Vulnerable	Yes	Yes	No
Fizzle 0.5.1	Vulnerable	Yes	Yes	No
Fizzle-0.5.2	Vulnerable	Yes	Yes	No
Beatnik-1.2	Vulnerable	Yes	Yes	No
Budaneki-2.0	Vulnerable	Yes	No	No
Facebook_dislike-3.0.2	Malicious	Yes	No	No
Facebook_Rosa	Malicious	Yes	No	No

information is obtained, the *userid* information is leaked to a remote website (`http://dislikenew.doweb.fr/get2.php`) that is not related to Facebook.

We also compared our approach with two other related work (i.e., [2], [3]) with respect to vulnerable and malicious extensions as shown in Table VI. We noticed that our approach performs better than VEX for identifying not only known, but also unknown vulnerable and malicious extensions. Thus, our HMM-based approach is complementary to other approaches for detecting vulnerable and malicious extensions.

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

Listing 5: Code snippet that is found to be malicious in `Facebook_dislike extension`.

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 shows 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 recorded 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., [3].

VI. Related Work

An approach based on static information flow analysis to detect (some of) the known vulnerabilities in Firefox browser extensions is discussed in [2]. 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 dynamic taint analysis in the Firefox browser to detect the execution of untrusted JavaScript code is also presented in [12]. In [26], 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.

An approach to mitigate the exploitation of vulnerable extensions based on separation of privilege levels is proposed in [5]. 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 what websites it intends to access in the `manifest` file. Thus, an attacker may not be able to access web pages from other websites to read sensitive information. If an extension requires executing arbitrary code, the corresponding binary file must be specified in the `manifest`. 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 [27]. The authors pesented a micro-privilege management tool to augment the existing privileges supported by Chrome and provide finer-grained privileges. Our work complements this work by widening the scope of the detection to identify vulnerable and malicious extensions.

A number of approaches have been proposed to discover malicious extensions in web browsers automatically [3], [4], [28], [29]. More specifically, an approach to mitigate malicious extensions from being installed or operating in browsers is developed [3]. This approach relies on ensuring the integrity of browser code based on 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. The 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.

Various work use HMM-based and stochastic techniques to detect malicious activities. In this area, the work closest in spirit to ours can be found in [4], [30], [31], [32], [33], [34]. A learning-based approach to detect malicious extensions for Internet Explorer browser is presented in [4]. The approach first 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. Moreover, while these approaches to detect malicious extensions and malwares 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 [32]. Similarly, an HMM-based detection model is built to detect rogue wireless access points [30]. The 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, e.g., the 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 [31]. 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. [33] apply HMM to extract the general pattern of XSS attack signatures with the objective to generate new attack signatures. An application level IDS to detect attacks in web applications is presented in [34]. In particular, the authors developed an HMM model based on legitimate HTTP traffic (request and response URL), which is then used for detection 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.

VII. Conclusion

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. 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 automatically detect real-world browser extensions 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.

Currently, the approach analyzes JavaScript source code based on common DOM and XPCOM APIs. In the future, we plan to apply a suitable JavaScript parser to other relevant static or dynamic method calls —e.g., to analyze parameters to dynamic JavaScript code generation functions such as `eval()`. To evaluate the effectiveness of our approach using large sample size, we plan to analyze and collect more extension samples from public incident reports such as firefox bug database [35]. Finally, we would like to expand our developed benchmark by introducing more general set of rules to convert benign extensions into vulnerable and malicious ones.

References

[1] T. Klosowski, "Master your browser's tabs with these tricks and extensions," http://lifehacker.com/5883299/master-your-browsers-tabs-with-these-tricks-and-extensions, 2012.

[2] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett, "Vetting browser extensions for security vulnerabilities with vex," *Commun. ACM*, vol. 54, pp. 91–99, Sep. 2011.

[3] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan, "Enhancing web browser security against malware extensions," *Journal in Computer Virology*, vol. 4, no. 3, pp. 179–195, 2008.

[4] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer, "Behavior-based spyware detection," in *USENIX Security Symposium*, 2006.

[5] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," in *NDSS*, 2010.

[6] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, and Z. Feng, "An empirical study of dangerous behaviors in firefox extensions," in *ISC*, pp. 188–203.

[7] Z. Chufeng and W. Qingxian, "Systematical vulnerability detection in browser validation mechanism," in *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, 2011, pp. 831–836.

[8] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: get the security of multiple browsers with just one," in *CCS*, 2011, pp. 227–238.

[9] Chris Grier et al., "Browser Exploits as a Service: The Monetization of Drive-by- Downloads," in *CCS*, 2012.

[10] B. Eshete, A. Villafiorita, and K. Weldemariam, "BINSPECT: Holistic Analysis and Detection of Malicious Web Pages," in *SecureComm*, 2012, pp. 149–166.

[11] M. Dhawan and V. Ganapathy, "Analyzing information flow in javascript-based browser extensions," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09. IEEE Computer Society, 2009, pp. 382–391.

[12] V. Djeric and A. Goel, "Securing script-based extensibility in web browsers," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10, 2010.

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

[14] R. Nyman, "How to develop a firefox extension," https://blog.mozilla.org/addons/2009/01/28/how-to-develop-a-firefox-extension/, Jan 2009.

[15] MOZILLA DEVELOPER NETWORK, "A simple menu bar tutorial," https://developer.mozilla.org/en-US/docs/XUL/Tutorial?redirectlocale=en-US&redirectslug=XUL_Tutorial.

[16] MOZILLA DEVELOPER NETWORK, "XPCOM Interface Reference by grouping," https://developer.mozilla.org/en/XPCOM_Interface_Reference_group, December 2010.

[17] Mozilla Firefox, "Best of 2 billion firefox add-ons," https://addons.mozilla.org/en-US/firefox/collections/mozilla/bestof2billion/?src=rockyourfirefox, December 2012.

[18] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, and Z. Feng, "An empirical study of dangerous behaviors in firefox extensions," in *ISC*, 2012, pp. 188–203.

[19] Robin, "Baum welch algorithm," http://language.worldofcomputing.net/pos-tagging/baum-welch-algorithm.html, December 2009.

[20] A. Milowski, "JHMM: An Implementation of Hidden Markov Models and Training in Java," http://code.google.com/p/jhmm/.

[21] Mozilla Firefox, "Add-ons," https://addons.mozilla.org/en-US/firefox/, 2012.

[22] N. Freeman and R. S. Liverani, "Exploiting Cross Context Scripting Vulnerabilities in Firefox," notehttp://www.security-assessment.com/files/documents/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf, April 2010.

[23] R. S. Liverani, "Cross Context Scripting with Firefox," notehttp://www.security-assessment.com/files/documents/whitepapers/Cross_Context_Scripting_with_Firefox.pdf, April 2010.

[24] Mozilla Firefox, "Firefox issues caused by Malware," http://support.mozilla.org/en-US/kb/troubleshoot-firefox-issues-caused-malware#w_how-do-i-know-that-my-firefox-problem-is-a-result-of-malware, 2011.

[25] SPAMfighter News, "Scammers using browser extensions to hack facebook accounts," http://www.spamfighter.com/News-17241-Scammers-Using-Browser-Extensions-to-Hack-Facebook-Accounts.htm, December 2011.

[26] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan, "An analysis of the mozilla jetpack extension framework," in *ECOOP*, 2012, pp. 333–355.

[27] G. Y. Lei Liu, Xinwen Zhang and S. Chen, "Chrome extensions: Threat analysis and countermeasures," in *Proceedings of the NDSS Symposium*, 2012.

[28] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: fast and precise in-browser javascript malware detection," in *USENIX Security Symposium*, ser. SEC'11, 2011.

[29] C. Kolbitsch, B. Livshits, B. G. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *IEEE Symposium on Security and Privacy*, 2012, pp. 443–457.

[30] G. Shivaraj, M. Song, and S. Shetty, "A hidden markov model based approach to detect rogue access points," in *Military Communications Conference.*, 2008, pp. 1 –7.

[31] S. Kazi, "Hidden markov models for software piracy detection, msc thesis, san jose state university,,," Master's thesis, San Jose State University, April 2012, http://scholarworks.sjsu.edu/etd_projects/236.

[32] M. Al-Subaie and M. Zulkernine, "Efficacy of hidden markov models over neural networks in anomaly intrusion detection," in *COMPSAC*, 2006, pp. 325–332.

[33] Y.-H. Wang, C.-H. Mao, and H.-M. Lee, "Structural learning of attack vectors for generating mutated xss attacks," in *TAVWEB*, 2010, pp. 15–26.

[34] I. Corona, D. Ariu, and G. Giacinto, "Hmm-web: a framework for the detection of attacks against web applications," in *Proceedings of the 2009 IEEE international conference on Communications*, 2009, pp. 747–752.

[35] Bugzilla, "Bugzilla mozilla," https://bugzilla.mozilla.org/, Last accessed, January 2013.