Detecting Obfuscated JavaScript Malware Using
Sequences of Internal Function Calls

Alireza Gorji
Tarbiat Modares University
Tehran, Iran
alireza.gorji@modares.ac.ir

Mahdi Abadi
Tarbiat Modares University
Tehran, Iran
abadi@modares.ac.ir

ABSTRACT
Web browsers are often used as a popular means for compromising Internet hosts. An attacker may inject a JavaScript malware into a web page. When a victim visits this page, the malware is executed and attempts to exploit a specific browser vulnerability or download an unwanted program. Obfuscated JavaScript malware can easily evade signature-based detection by changing the appearance of JavaScript code. To address this problem, some previous studies have used static analysis in which some features are extracted from both benign and malicious web pages, and then a classifier is trained to distinguish between them. Because nowadays benign JavaScript code is often obfuscated, static analysis techniques generate many false alarms. In this paper, we use dynamic analysis to monitor a web page for detecting obfuscated JavaScript malware. We first load a set of malicious web pages in a real web browser and collect a sequence of predictive function calls using internal function debugging for each of them. We then group similar sequences into the same cluster based on the normalized Levenshtein distance (NLD) metric and generate a so-called behavioral signature for each cluster. A web page is detected as malicious if the sequence of its intercepted function calls is matched with at least one generated behavioral signature. Our evaluation results show that the generated behavioral signatures are able to detect obfuscated JavaScript malware with a low false alarm rate.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Invasive software

General Terms
Security

Keywords
JavaScript Malware, Obfuscation, Internal Function Call, Behavioral Signature.

1. INTRODUCTION
Malicious web pages are one of the most common ways to compromise Internet hosts. These pages usually host drive-by-download attacks that exploit vulnerabilities in the victim’s web browser or one of its plugins. Therefore, protecting users from visiting malicious websites is the main concern of many researchers in the malware analysis field. JavaScript is an ideal language for malware writers, because it is a built-in part of almost every web browser and can be used as a powerful tool to make unintended downloads and redirects. Static signatures are heavily used in malware protection systems to match known patterns that are commonly found in malicious web pages. User is warned when trying to visit a compromised web page. Although signature-based detection is currently the most commonly used technique for malware detection, it is unable to detect zero-day JavaScript malware. Attackers often use obfuscation to evade signature-based detectors and make JavaScript code difficult to understand and analyze. Choosing different types of obfuscation techniques and combining them together makes it even more resistant to analysis.

Several studies have focused on static analysis for detecting both obfuscated and non-obfuscated JavaScript malware [2, 11, 12, 13]. This analysis takes into account discriminative features derived from the HTML contents, JavaScript code, and corresponding URL of web pages. A detection model is then built from these features based on machine learning techniques. Nowadays benign JavaScript code is often minimized, optimized, or obfuscated, hence static analysis techniques suffer from high false alarm rates that make them ineffective for using in a malware protection system.

In this paper, we focus on dynamic analysis for detecting obfuscated JavaScript malware. To this end, we use internal function debugging and map the JavaScript behavior of a web page to a sequence of internal function calls, which enables us to generate a behavioral signature for each JavaScript malware family.

The rest of this paper is organized as follows: Section 2 describes the most popular JavaScript obfuscation techniques. Section 3 reviews some related work and Section 4 introduces internal function debugging. Section 5 gives basic definitions. Section 6 presents our approach for detecting obfuscated JavaScript malware and Section 7 reports experimental results. Finally, Section 8 draws some conclusions.

2. OBFUSCATION TECHNIQUES
Obfuscation techniques are heavily used in JavaScript malware to evade the detection of malware protection systems and to hide their malicious intent. In the following subsections, we briefly describe the major categories of obfuscation techniques being used in wild [7, 20].

2.1 Randomization Obfuscation
JavaScript malware may be obfuscated by randomly inserting or changing some elements of code without changing the semantics at all. Variable and function name randomization, whitespace ran-
domination, and comment randomization are common techniques in this category. The variable and function name randomization replaces variable names and function names by random strings. This makes the code analysis too complex. Whitespace randomization randomly inserts whitespace characters including space, tab, line feed, and carriage return. The rationale behind it is that the JavaScript interpreter ignores whitespace characters. Figure 1 gives a demonstration of these two techniques.

```javascript
function myFunc(str) {
    document.write(str);
}
var myStr = "My Code";
myFunc(myStr);

// (a) Original code
// (b) Obfuscated code
```

Figure 1. An example of the randomization obfuscation

2.2 Data Obfuscation

The data obfuscation is as simple as splitting a string into multiple variables or substrings and concatenating them later, perhaps by using the `document.write` or `eval` functions. An attacker may also change the order of variables to make the code even harder to analyze (see Figure 2).

```javascript
var myStr = "document.write(\"My String\")";
eval(myStr);
var yq = "img";
var sq = "doc";
var sm = "ument";
var kw = "\"My Str\"; myStr = sq + sm + ",.write" + kw + yq;
eval(myStr);

// (a) Original code
// (b) Obfuscated code
```

Figure 2. An example of the string splitting

2.3 Encoding Obfuscation

In general, there are two techniques for the encoding obfuscation. The first technique is to convert characters into their ASCII or Unicode values. Figure 3 shows an example of using Unicode values to obfuscate the string "<iframe src='http://www.devil.com'>".

```javascript
document.write("\u0069\u0066\u0072\u0061\u006D\u003E\".write\"");
eval(yq);
var yq = "img";
var sq = "doc";
var sm = "ument";
var kw = "\"My Str\"; myStr = sq + sm + ",.write" + kw + yq;
eval(myStr);

// (a) Original code
// (b) Obfuscated code
```

Figure 3. An example of using Unicode values

The second technique uses customized encryption. In this technique, an attacker writes an encryption function to encrypt a given string and attaches a decryption function to decrypt it during execution. Figure 4 shows an example of using the custom encryption in which the Unicode value of each character is increased by the value of the parameter key, where the string "document.write('My Code')" is encrypted into "ithzjsy3\%wnyj-\%Htij-\".

```javascript
function encrypt(str, key) {
    var result = "\";
    for (var i = 0; i < str.length; i++) {
        result += String.fromCharCode(str.charCodeAt(i) + key);
    }
    return result;
}
var encStr = encrypt("document.write(\"My Code\")\", 5);
document.write(encStr);

// (a) Original code
// (b) Obfuscated code
```

Figure 4. An example of the custom encryption

2.4 Logic Structure Obfuscation

An attacker may manipulate the execution path of the JavaScript code by changing its logic structure. An example is inserting dead code which is a piece of code that will never run during the execution.

3. RELATED WORK

Several recent studies have focused on the detection of malicious web pages. Since 2004, web browser based high interaction client honeypots have become an attractive tool to detect malware attacks on clients [15, 17]. They typically detect the attacks by loading suspicious web pages in a vulnerable web browser and monitoring changes in files, registry, and processes. Some other works try to detect malicious content before it reaches the victim's web browser in network layers or through a crawler.

Cova et al. [3] presented a tool, called JSAND, which combines anomaly detection with emulation to identify JavaScript malware. JSAND uses a number of features and machine-learning techniques to establish profiles of normal JavaScript code. It is able to identify anomalous JavaScript code by emulating its behavior and comparing it to the established profiles. They made JSAND available as part of an online service, called Wepawet [19], where users can submit URLs that are automatically analyzed.

Riek et al. [16] presented CUJO, a system that combines static analysis with dynamic analysis to detect drive-by-download attacks. The static analysis component extracts lexical tokens from the JavaScript code of a web page and generates a static analysis report. The dynamic analysis component loads the web page in an enhanced version of ADSandbox [5] and generates a dynamic analysis report containing all monitored operations of the JavaScript code. These reports are then processed and two detection models are generated based on support vector machines. CUJO uses JavaScript emulation instead of loading the web page in a real web browser. But our approach differs from CUJO in that it uses the browser's JavaScript engine without requiring emulation.

Canali et al. [2] introduced Prophiler, a filtering system that uses static analysis to distinguish between malicious and benign web pages. It uses some features derived from the HTML content, associated JavaScript code, and URL of each web page to build detection models using supervised machine-learning techniques. Web pages that are found to be likely malicious are then analyzed with an analysis tool, such as Wepawet [19]. The drawback is that few features focus on the JavaScript obfuscation; hence Prophiler cannot effectively detect obfuscated JavaScript malware.

Curtisinger et al. [4] proposed ZOZZLE, a static JavaScript malware detector. ZOZZLE hooks JavaScript functions, such as `document.write` and `eval`, to obtain deobfuscated JavaScript code and pass it to a naive Bayesian classifier that is trained using features of the JavaScript AST (abstract syntax tree). However, they simply rely on the assumption that JavaScript malware have to be deobfuscated at last. Kim et al. [10] presented JSSandBox, a framework that monitors and analyzes the behavior of JavaScript malware using internal function hooking (IFH). Similar to our approach, they attach a debugger to a real web browser to inter-
cept internal function calls. However, they make a simple model for detecting the behavior of JavaScript malware that can be easily evaded by different obfuscation techniques. All of these works are different from the approach discussed in this paper.

4. INTERNAL FUNCTION DEBUGGING

API hooking is a powerful technique for analyzing the behavior of programs running on Microsoft Windows. It changes the execution flow of a program into an arbitrary flow and monitor the input and output parameters of function calls. However, its main drawback is that it incurs additional overhead to the monitoring process. In this paper, we refer to all exported and non-exported functions in a DLL as internal functions and use internal function debugging to intercept specific internal function calls when a web page is loaded into a web browser. In general, debugging information includes function names, local and global variables, register values, call stack, and so on. A Windows debugger needs debug symbols that are the name and starting address of functions and global variables. Microsoft provides debug symbols for the 

Internet Explorer uses several DLLs for loading web pages, including jsrct.dll (JavaScript engine) and nshdl.dll (HTML parser). Each JavaScript function is implemented by a number of internal functions in jsrct.dll. For example, the JavaScript function eval and unescape are implemented as the internal functions JsEval and JsUnescape, respectively. Also, each HTML tag is implemented as a class in nshdl.dll. For example, the HTML tag SCRIPT is implemented as the class CScriptElement.

We use WinDbg [14] to trace internal function calls in jsrct.dll and nshdl.dll when a web page is loaded into Internet Explorer. WinDbg is a multipurpose debugger for Microsoft Windows that can be attached to a running process to intercept internal function calls and view debugging information. WinDbg supports built-in commands for various debugging purposes. To be notified when specific internal functions are getting called, we send a sequence of commands to the WinDbg engine (adbeng.dll).

5. BASIC DEFINITIONS

In this section, we give some basic definitions which are used throughout the paper.

Definition 1 (F-sequence). Let F be a set of internal function calls. An ordered list \( f_i \) of internal function calls in \( F \) is called an F-sequence:
\[
\mathbf{f}_1 = \langle f_{i_1}, f_{i_2}, \ldots, f_{i_p} \rangle ,
\]
where \( f_{i_k} \in F \) is an internal function call and \( p = |\mathbf{f}_1| \) is the length of \( \mathbf{f}_1 \).

Definition 2 (F-vector). Let \( F \) be a set of internal function calls and \( \mathbf{f}_1 \) be an F-sequence. A vector \( \mathbf{f} \) whose elements is the number of internal function calls in \( \mathbf{f}_1 \) is called an F-vector of \( \mathbf{f}_1 \):
\[
\mathbf{f} = \langle n_{i_1}, n_{i_2}, \ldots, n_{i_d} \rangle ,
\]
where \( n_{i_k} \) is the frequency of an internal function call \( f_{i_k} \in F \) in \( \mathbf{f}_1 \), and \( d = |F| \) is the total number of internal function calls.

Definition 3 (Normalized Levenshtein distance). Let \( \text{lds}(\mathbf{f}_1, \mathbf{f}_j) \) be the Levenshtein distance between two F-sequences \( \mathbf{f}_1 \) and \( \mathbf{f}_j \), i.e., the minimum number of edits needed to convert \( \mathbf{f}_1 \) into \( \mathbf{f}_j \) or vice versa, with the edit operations of insertion, deletion, or substitution of function calls. The normalized Levenshtein distance (NLD) between \( \mathbf{f}_1 \) and \( \mathbf{f}_j \), denoted by \( \xi(\mathbf{f}_1, \mathbf{f}_j) \), is calculated as
\[
\xi(\mathbf{f}_1, \mathbf{f}_j) = \frac{\text{lds}(\mathbf{f}_1, \mathbf{f}_j)}{\max(|\mathbf{f}_1|, |\mathbf{f}_j|)} ,
\]
(3)

Definition 4 (Fixed-width cluster). A fixed-width cluster is a set of F-sequences whose normalized Levenshtein distance from the cluster centroid is less than a fixed radius \( p \).

Definition 5 (Cluster representative). Let \( c_k \) be a fixed-width cluster. The representative of \( c_k \), denoted by \( \varphi(c_k) \), is defined as the average of all F-vectors in \( c_k \):
\[
\varphi(c_k) = \frac{1}{|c_k|} \sum_{\mathbf{f}_j \in c_k} \mathbf{f}_j ,
\]
(4)

Definition 6 (Behavioral distance). The behavioral distance of an F-sequence \( \mathbf{f}_i \) from a fixed-width cluster \( c_k \), denoted by \( \delta(\mathbf{f}_i, c_k) \), is defined as the distance between \( \varphi(\mathbf{f}_i) \) and \( \varphi(c_k) \):
\[
\delta(\mathbf{f}_i, c_k) = \|\varphi(\mathbf{f}_i) - \varphi(c_k)\| ,
\]
(5)

where \( \| \cdot \| \) is the norm of a vector.

Definition 7 (F-substring). An F-substring of an F-sequence \( \mathbf{f}_i \) is an F-sequence \( \mathbf{f}_j = \langle f_{i_1}, \ldots, f_{i_k} \rangle \), where \( k \geq 0 \) and \( k + q \leq p \).

Definition 8 (F-token). Given a set \( S \) of F-sequences, an F-sequence \( \mathbf{f}_i \) is called an F-token for \( S \) iff \( \mathbf{f}_i \) is a common F-substring of all F-sequences in \( S \).

Definition 9 (F-signature). Given two sets \( S \) and \( T \) of F-sequences, \( T \) is called an F-signature for \( S \) iff each F-sequence in \( T \) is an F-token for \( S \).

Definition 10 (Redundant F-token). Let \( S \) be a set of F-sequences and \( T \) be an F-signature for \( S \). An F-token \( \mathbf{f}_i \) in \( T \) is redundant iff there exists an F-token \( \mathbf{f}_j \) in \( T \) such that \( \mathbf{f}_i \) is an F-substring of \( \mathbf{f}_j \) and \( \mathbf{f}_j \neq \mathbf{f}_i \).

Example 1. Consider two sets \( S \) and \( T \) of F-sequences:
\[
T = \{ (A, B, A, B, C, (H, H, H, A)) \} ,
\]
(3)

\( T \) is an F-signature for \( S \). The F-token \( (A, B, C) \) in \( T \) is redundant, because there exists an F-token \( (A, B, C) \) such that \( (A, B) \) is an F-substring of \( (A, B, C) \).

Definition 11 (Minimal F-signature). An F-signature \( T_m \) is minimal iff it contains no redundant F-token.

Example 2. A minimal F-signature for \( S \) in Example 1 is the set \( T_m = \{ (A, B, C), (H, H, H, A) \} \).

6. OUR APPROACH

In this section, we present an approach to detect obfuscated Java-

Script malware by monitoring and analyzing the behavior of sus-

picious web pages in a real web browser.

As previously mentioned, every standard web browser has a Java-

Script engine that interprets and executes JavaScript code embed-

ded in web pages. This engine is basically implemented in some

DLLs. When a web page is loaded into the browser, a sequence of

internal function calls is made. We are interested in the function

calls which are made by obfuscated JavaScript malware.
and malicious web pages is calculated as follows:
\[ \chi^2 = \frac{(AD - CB)^2}{(A + C)(B + D)(A + B)(C + D)}, \]
where \( A \) and \( C \) are the number of malicious web pages with and without \( f \), respectively. \( B \) and \( D \) are the number of benign web pages with and without \( f \), respectively. The higher the value of \( \chi^2 \), the more the correlation between \( f \) and malicious web pages.

We select top 15 most predictive internal function calls of \( \text{jsrcript.dll} \) and \( \text{mshtml.dll} \), as listed in Table 1. For ease of exposition, we map each of them to an English alphabet letter.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Function</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>JsEval</td>
<td>jscript</td>
</tr>
<tr>
<td>B</td>
<td>CScriptRuntime::InitEval</td>
<td>jscript</td>
</tr>
<tr>
<td>C</td>
<td>CDocument::Write</td>
<td>jscript</td>
</tr>
<tr>
<td>D</td>
<td>COleScript::ChangeType</td>
<td>jscript</td>
</tr>
<tr>
<td>E</td>
<td>StringFncObj::EnsureBuiltIn</td>
<td>jscript</td>
</tr>
<tr>
<td>F</td>
<td>ErrorFncObj::EnsureBuiltIn</td>
<td>jscript</td>
</tr>
<tr>
<td>G</td>
<td>NameTbl::Call</td>
<td>jscript</td>
</tr>
<tr>
<td>H</td>
<td>NatFacObj::Create</td>
<td>jscript</td>
</tr>
<tr>
<td>I</td>
<td>CScriptRuntime::SetNextStatement</td>
<td>jscript</td>
</tr>
<tr>
<td>J</td>
<td>StringFncObj::StringFncObj</td>
<td>jscript</td>
</tr>
<tr>
<td>K</td>
<td>StringFncObj::Create</td>
<td>jscript</td>
</tr>
<tr>
<td>L</td>
<td>ThreadGlobals::ThreadGlobals</td>
<td>jscript</td>
</tr>
<tr>
<td>M</td>
<td>DexCaller::QueryService</td>
<td>jscript</td>
</tr>
<tr>
<td>N</td>
<td>CScriptRuntime::RecordErrorContext</td>
<td>jscript</td>
</tr>
<tr>
<td>O</td>
<td>CScriptRuntime::GetCodeContext</td>
<td>jscript</td>
</tr>
</tbody>
</table>

6.2 Sequence-based Clustering
To identify malicious web pages with similar behavior, we use a fixed-width sequence-based clustering algorithm, called \( FSC \). The algorithm takes a set \( M \) of \( \mathcal{F} \)-sequences of malicious web pages as the input and groups them into a set \( C \) of fixed-width clusters based on the normalized Levenshtein distance metric. For each \( \mathcal{F} \)-sequence \( f_i \in M \), if \( C \) is empty, it creates a new cluster \( c_1 \) with \( f_i \) as its centroid (Lines 3–6). Otherwise, it adds \( f_i \) to the nearest cluster \( c_{\min} \in C \) and updates the cluster centroid \( \theta(c_{\min}) \) only if the normalized Levenshtein distance between \( f_i \) and \( \theta(c_{\min}) \) is less than a fixed radius \( \rho \) (Lines 8–11) and otherwise, it creates a new cluster \( c_{\rho} \) with \( f_i \) as its centroid (Lines 12–16). The pseudocode of \( FSC \) is shown in Figure 6.

6.3 Signature Generation
At the moment, we have a set \( C \) of clusters created from malicious web pages. The next step is to generate a set \( \mathcal{S} \) of minimal \( \mathcal{F} \)-signatures for all clusters in \( C \). Although generating a set of \( \mathcal{F} \)-signatures, each of which consists of only the longest \( \mathcal{F} \)-tokens for a cluster in \( C \), will achieve a high detection rate, but it may also result in a high false alarm rate. In addition to that, an attacker may insert benign pieces of code between malicious JavaScript code to evade signature-based matching, decreasing the detection rate. These facts motivates us to define more accurate \( \mathcal{F} \)-signatures that match with \( \mathcal{F} \)-sequences of different variants of malicious web pages as much as possible and, at the same time, do not match with \( \mathcal{F} \)-sequences of benign web pages. For this purpose, we define an \( \mathcal{F} \)-signature to be the set of all \( \mathcal{F} \)-tokens with a minimum length \( \delta \) for a cluster. There are well-known algorithms to find all common substrings of a set of strings in linear time \([8, 9]\). We can slightly modify them to find a set of all non-redundant \( \mathcal{F} \)-tokens with a minimum length \( \delta \) for each cluster in \( C \), which gives us the set \( \mathcal{S} \) of minimal \( \mathcal{F} \)-signatures. To increase the probability of matching an \( \mathcal{F} \)-signature with \( \mathcal{F} \)-sequences of similar malicious
web pages, we implement each F-token in $\Sigma$ as a regular expression. For example, the F-token (B, A, H, H, H, C, H, H, D, D, A) is implemented as the regular expression "BAH+CH+D+A". The plus sign in the regular expression shows that the internal function calls H and D are made more than once. In general, the number of occurrences of a repeated internal function call may not be the same in different variants of a JavaScript malware. Therefore, we generalize F-tokens with implementing them as regular expressions. Our evaluation shows that this conversion does not affect the false alarm rate that much. Also, some minimal F-signatures in $\Sigma$ may be incorrectly matched with F-sequences of benign web pages, increasing the false alarm rate. Therefore, we prune these so-called noisy F-signatures from $\Sigma$. For this purpose, we remove every minimal F-signature that matches with more than $\omega$ percent of benign web pages in a validation dataset.

7. EVALUATION

In this section, we evaluate the performance of our approach using a dataset of benign and malicious web pages. For this purpose, we collected more than 1000 malicious web pages, compromised through iFRAME injection attack, from urlQuery [18] which is a service for detecting and analyzing web-based malware. We manually analyzed all malicious web pages to ensure that obfuscation techniques are applied in their JavaScript code and selected 300 malicious web pages among them. The iFRAME injection has become an incredibly common attack these days. It hijacks the victim website's traffic to serve malware to visitors. The attacker injects one or more iframe tags into a web page's content; hence if a usual visitor of that web page opens it, the malicious content will be loaded into its web browser.

We also collected a set of 3,000 benign web pages from top 3,000 Alexa web sites [1]. We assume that these well-known websites are not compromised, as they are visited daily by millions of visitors and frequently analyzed by experts and anti-virus tools. We used 175 malicious web pages to generate minimal F-signatures, 1000 benign web pages to prune noisy F-signatures. We also used 2000 benign and 125 malicious web pages to evaluate the performance of our approach. To create an F-sequence for each web page, we loaded it into Internet Explorer and intercepted predictive internal function calls for 60 seconds. We believe this time is sufficient for a web page to execute all its JavaScript code. In all experiments, we set the pruning threshold $\omega$ to 0.01. Some generated minimal F-signatures are listed in Table 2.

We use the detection rate (DR) and false alarm rate (FAR) to measure the quality of generated minimal F-signatures. Figure 7 shows the effect of different values of cluster radius $\rho$, ranging from 0.4 to 0.7, and pruning threshold $\omega$ on the performance of our approach. Clearly, we can make a trade-off between the detection and false alarm rates by choosing $\rho = 0.55$ and $\omega = 0.01$.

Table 3 shows the number of minimal F-signatures, $|\Sigma|$, for different values of $\rho$, ranging from 0.4 to 0.7. The higher the value of $\rho$, the less the number of minimal F-signatures.

![Figure 6. The pseudo-code of FSC](image_url)

![Figure 7. Effect of $\rho$ and $\omega$ on the performance of our approach](image_url)
web pages in a real web browser. We performed several experiments using a dataset of benign and malicious web pages to evaluate the performance of our approach. The malicious web pages were compromised through iFRAME injection attack. The evaluation results demonstrated that by choosing the fixed cluster radius $\rho = 0.55$ and the pruning threshold $\omega = 0.01$, we can achieve a relatively high detection rate of 85.07% with a low false alarm rate of 0.26% which is less than 1%. In future work, we will generate minimal $F$-signatures for JavaScript malware other than those using the iFRAME injection attack and we will also generate minimal $F$-signatures from internal function calls of other web browsers, such as Mozilla Firefox, Google Chrome, and so on.

8. CONCLUSION

In this paper, we presented a dynamic approach to detect obfuscated JavaScript malware by analyzing the behavior of suspicious web pages in a real web browser. We performed several experiments using a dataset of benign and malicious web pages to evaluate the performance of our approach. The malicious web pages were compromised through iFRAME injection attack. The evaluation results demonstrated that by choosing the fixed cluster radius $\rho = 0.55$ and the pruning threshold $\omega = 0.01$, we can achieve a relatively high detection rate of 85.07% with a low false alarm rate of 0.26% which is less than 1%. In future work, we will generate minimal $F$-signatures for JavaScript malware other than those using the iFRAME injection attack and we will also generate minimal $F$-signatures from internal function calls of other web browsers, such as Mozilla Firefox, Google Chrome, and so on.

9. REFERENCES


