Analysis and Identification of Malicious JavaScript Code

Mohammad Fraiwan¹, Rami Al-Salman¹, Natheer Khasawneh², and Stefan Conrad³
¹Department of Computer Engineering, Jordan University of Science and Technology, Irbid, Jordan
²Department of Software Engineering, Jordan University of Science and Technology, Irbid, Jordan
³Institute of Computer Science, Heinrich-Heine University, Düsseldorf, Germany

ABSTRACT Malicious JavaScript code has been actively and recently utilized as a vehicle for Web-based security attacks. By exploiting vulnerabilities such as cross-site scripting (XSS), attackers are able to spread worms, conduct Phishing attacks, and do Web page redirection to “typically” porn Web sites. These attacks can be preemptively prevented if the malicious code is detected before executing. Based on the fact that a malignant code will exhibit certain features, we propose a novel classification-based detection approach that will identify Web pages containing infected code. Using datasets of trusted and malicious Web sites, we analyze the behavior and properties of JavaScript code to point out its key features. These features form the basis of our identification system and are used to properly train the various classifiers on malicious and benign data. Performance evaluation results show that our approach achieves a 95% or higher detection accuracy, with very small (less than 3%) false positive and false negative ratios. Our solution surpasses the performance of the comparable literature.

KEYWORDS classification, malicious JavaScript, Web testing

1. INTRODUCTION

JavaScript, as a programming/scripting language, is used in Web development to add more features, effects, and improvements to the end-user experience. For example, JavaScript can help reduce the server-side load by conducting validation checks of filled forms prior to sending them to the Web server. Thus, it is shifting some of the computation to the users’ side. However, enabling such capabilities can endanger the end-user if the Web page is infected with malicious JavaScript code. Although JavaScript can be disabled by the Web browser, many Web sites require this feature to be enabled in order to view their JavaScript intensive content. Malicious JavaScript code is typically injected around benign code. Moreover, attackers will obfuscate this code to avoid detection mechanisms (e.g., de-compilation; see FIGURE 1). This infected code is used to spread worms and viruses, install Malware, conduct Phishing attacks, spread spam, and redirect Web pages to typically porn Web sites. The most recent attack employing JavaScript code at the time of this
writing has affected the Twitter social networking Web site. The JavaScript code was used to spread worms by injecting false tweets into users’ pages (“Twitter scrambles,” 2010).

Although it is imperative to use and enable JavaScript in browsers, it is hard to stop malicious code from downloading into end users’ (i.e., victims) machines. Attackers use various techniques to lure end users into downloading the infected Web content (e.g., P2P file sharing, pop up ads, spam emails). In the aforementioned Twitter attack, users were only required to move the mouse over the false tweets to get infected, without even clicking (“Twitter scrambles,” 2010). Moreover, most countermeasures that flag certain Web sites as dangerous may be ineffective and outdated (Ma, Saul, Savage, & Voelker, 2009).

For example, predefined blacklists, such as PhishTank (PhishTank, 2010) are ineffective as the attackers are fast in changing their IP addresses and URLs (Moore, Clayton, & Stern, 2009). Continuously updating these lists requires high overhead, making it difficult to keep up with the attackers. Also, Fast Flux attacks, where attackers exploit a continually changing network of compromised nodes, are becoming more prevalent (Holz, Gorecki, Rieck, & Freiling, 2008), thus incurring traditional countermeasures. JavaScript should always be enabled for the correct operation of a wide variety of Web pages. However, harmful code can be easily and inadvertently downloaded and run. Thus, we target the execution of the malicious code. Attacks can be preemptively prevented if this code is detected before executing. Because a malignant code will exhibit certain features, we propose a novel classification-based detection approach. This approach will identify Web pages containing infected code. The classification model is built based on attributes of malicious JavaScript codes, giving end users the option to make the decision on whether he/she wants to view and/or run the possibly infected Web content.

The objectives of this paper are as follows:

1. We study and analyze the syntax and semantics of malicious JavaScript code to point out its key distinguishing features and marks.
2. Based on these feature, we propose the use of classification algorithms to single out malicious code.
3. We conduct performance evaluation studies on various datasets to show the effectiveness of our proposed scheme.
4. We compare our work to recent related solutions in the literature.

The remainder of this paper is organized as follows. We discuss the related work in section 2. The analysis of malicious and benign JavaScript codes is presented in section 3. The system architecture is described in section 4. Section 5 describes the performance evaluation and comparison experiments. We conclude in section 6.

2. RELATED WORK

There has been a plethora of research into the detection of JavaScript malicious code. In this section, we survey the literature landscape, clarify key aspects and issues, and point out avenues for improvement and contribution. The majority of the works in the literature focus on a small subset of the malicious JavaScript features or distinguishing dimensions. These works failed in several ways:

1. They did not relate the JavaScript code with the content or other parts of the Web page.
2. They did not consider the behavior and semantics of the JavaScript code.
3. They focus on the behavior and ignore other factors.

The result of these shortcomings is a poor detection accuracy of malicious JavaScript code.

To detect any anomaly, it is natural to look for the behavior of that anomaly and its main characteristics as an indication of its existence. For example, a user may click on a trusted link (e.g., www.yahoo.
com); however, the malicious code will intentionally send a wrong request for an unavailable page (e.g., yahoo.com/IAMNOTAVAILABLE.html). After that, the trusted Web site will naturally return “Failed status 404 No document exists.” The user is then directed to a malicious Web site containing harmful code or open several pop-up windows. In an attempt to solve this problem, Hallaraker & Vigna (2005) monitor the behavior of the malicious code. Based on the number of windows opened as pop-up (i.e., using window.open()), they decide whether this code is malicious or not. The problem with their approach is that it will have many false positives (i.e., benign code classified as malicious) because they only consider the code behavior before and after running. The reputation of the URL also plays a role in its classification as malicious or not. Ma et al. (2009) used the Yahoo-PhishTank database of malicious Web sites in conjunction with other features to train a Support Vector Machine (SVM) classifier. This classifier is later used to identify malicious JavaScript code. They have reported an accuracy of 80%. Similarly, Likarish & Jung (2009) used a Web crawler to collect both malicious and benign JavaScript code. The collected data are later fed to a classifier for training. As before, the classifier is used at run time to detect malicious code.

Other machine learning-based approaches have also been developed. Kolter and Maloof (2006) use classification to detect malicious executables using a variety of inductive methods (e.g., Naïve Bayes). Several JavaScript-related tools and plug-ins exist in practice; Caffeine Monkey (Feinstein & Peck, 2007) is a customized release of the Mozilla SpiderMonkey (Mozilla C, n.d.) that has the capability to analyze and detect obfuscated malicious JavaScript and Flash code. Google Caja (google-caja, n.d.) is another tool that effectively sandboxes mistrusted code. It allows the containing application to completely control access rights of external content. Similarly in (Dewar, Holz, & Freiling, 2010), JavaScript code is executed in an isolated environment and every critical action is logged. An analysis of these logs decides whether the site is malicious. Some researchers have suggested adding policies that allow Web sites to control unexpected JavaScript execution (Jim, Swamy, & Hicks, 2007). This is especially helpful for Web sites importing and using external content. However, if the accessed Web site is already malicious, then there is no protection for the end user.

3. ANALYSIS OF MALICIOUS JAVASCRIPT CODE

We identify four categories of features that we used in comparing benign and malicious codes: URL properties, JavaScript code output, JavaScript code behavior, and JavaScript code content. The features per category are as follows:

3.1. URL Properties

(a) Is the URL blacklisted by trusted entities (e.g., Symantec)?
(b) Is the corresponding IP address blacklisted?

3.2. JavaScript Code Output

We check the output of the JavaScript code against the list of the most frequent words in comparable malicious JavaScript code. The main intuition here is that malicious JavaScript code will try to solicit users with a Web content of certain nature (e.g., monetary or sexual). The feature is good if the most frequent words in malicious codes are different from the benign ones, which is mostly true.

Table 1 show that the most common words appearing in the output of a malicious code differ greatly from those of a benign one. This feature can be used to distinguish the two types of JavaScript codes. Table 2 shows the two top 30 lists. Fortunately, these two lists are highly disjointed as only one word “privacy” is shared among the two lists.
### TABLE 1

A Comparison of the Most Frequent Words Appearing in the Output of Both Malicious and Benign JavaScript Codes. It is Clear that the Two Lists are Highly Disjointed

<table>
<thead>
<tr>
<th>Number of common words</th>
<th>Common words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>Privacy</td>
</tr>
<tr>
<td>30</td>
<td>Privacy</td>
</tr>
<tr>
<td>50</td>
<td>privacy, com, terms, jobs, information, business</td>
</tr>
<tr>
<td>70</td>
<td>privacy, com, terms, jobs, information, business, company</td>
</tr>
<tr>
<td>100</td>
<td>privacy, com, terms, jobs, information, business, company, see, email, here, sign, services, register, http</td>
</tr>
</tbody>
</table>

### TABLE 2

The List of Top 30 Most Frequent Words Appearing in Malicious and Benign URLs. Only Keyword Privacy is Shared Among the Two Lists

<table>
<thead>
<tr>
<th>Type of URL</th>
<th>Most frequent words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious</td>
<td>rights, fill, reserved, privacy, bill, work, online, have, home, password, policy, may, free, loans, permission, financial, money, information, making, personal, link, business, make, top, post, jobs, credit, debt, forum, apply, please</td>
</tr>
<tr>
<td>Benign</td>
<td>email, sign, 2010, keep, help, English, Facebook, logged, site careers, terms, forgot, privacy, advertising developers, people, friends, mobile, find, badges, life, share connect, search, helps, just, directory, use, center, pages, can</td>
</tr>
</tbody>
</table>

### TABLE 3

Percentage of Malicious and Benign Code Samples Containing More than a Certain Number of the Most Frequent Words Appearing in Malicious Code

<table>
<thead>
<tr>
<th></th>
<th>using &gt;2</th>
<th>using &gt;5</th>
<th>using &gt;10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious JS code</td>
<td>86%</td>
<td>35%</td>
<td>6%</td>
</tr>
<tr>
<td>Benign JS code</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 3 is a key to this distinguishing feature. The classification algorithm needs to learn the optimal bound on the number of matching words from the malicious set before flagging this feature as true (i.e., indicator of malicious code). This table shows that by analyzing malicious and benign code samples, the choice of more than two words is the best bound. The table shows that by increasing this number (e.g., more than 5) a large percentage of malicious samples will be excluded. However, only a small percentage of the benign samples contain these words. For example, only 6% of the malicious samples—and no benign samples—contain more than 10 of the most frequent words in malicious JS codes. On the other hand, about 86% of the malicious samples contain more than 2 words, but only 5% of the benign ones contain more than 2. Thus, we see it as a distinguishing feature.

### 3.3. JavaScript Code Behavior

Check for the `eval()` and `unescape()` functions. The combination of these functions is extensively used by attackers to execute obfuscated code. For example, in FIGURE 1, the function `find($)` replaces the obfuscated code with the actual intended malicious JavaScript code by means of character replacement. Then, the replacement is followed by script execution. The character replacement is done using the following method: first, a `for()` loop in the function inserts `%` at appropriate places in the obfuscated code. After that the `unescape()` function converts the hexadecimal code to the actual characters. For example, `unescape("It%27s%20me%21")` will result in producing the string “It’s me.” Finally, the `eval(find($))` function call is used to execute the malicious code. Similar functions, with comparable capabilities, can also be checked.

Table 4 show the percentage of malicious and benign code samples containing `eval()`, `unescape()`, the combination of `eval()/unescape()`, and XMLHttpRequest (feature 3c). It is clear that the `eval()` and `unescape()` functions alone are not strong indicators of malicious code. However, the combination of both is a strong indicator of malicious behavior.

### TABLE 4

Percentage of Malicious and Benign Code Samples Containing Eval(), Unescape(), Combination of Eval()/Unescape(), and XMLHttpRequest. Details of the Dataset are in Section 5

<table>
<thead>
<tr>
<th></th>
<th>eval()</th>
<th>unescape()</th>
<th>Combination of eval()/unescape()</th>
<th>XMLHttpRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious JS code</td>
<td>8%</td>
<td>8%</td>
<td>8%</td>
<td>3%</td>
</tr>
<tr>
<td>Benign JS code</td>
<td>2%</td>
<td>27%</td>
<td>0%</td>
<td>90%</td>
</tr>
</tbody>
</table>
Communicate with other programs (e.g., Applets). The malicious JavaScript code can communicate indirectly with outside entities (e.g., the attacker’s machine) by utilizing Applets and other similar components. This way, the JavaScript code will be more benign looking. However, it is extremely dangerous as it may be used to steal confidential user information.

If the JavaScript code communicates via the Document Object Model (DOM) XML HTTPREQUEST object with other servers, then this may be an indication of malice intent. For example, users’ information, including user names and passwords, can be communicated to other malicious servers. However, as Table 4 shows, this function is extensively used by benign code as well. Extra attention need to be paid on the destination of the communication and the type of information being communicated.

If the JavaScript code changes its behavior by redirecting users to other locations, then we can draw the same aforementioned conclusions. For example, redirect using the line `document.location` (“unsafeURL.com”).

### 3.4. JavaScript Code Content

This feature is related to the size-behavior and how the content of the code can be changed at run time. Much like the example in FIGURE 1, the de-obfuscation of the malicious code and its behavior may lead to changes in the size of the source code file.

Some of these indicators may be considered circumstantial or not enough for a clear-cut judgment. However, their combined existence provides a strong case to flag the code as malicious. Moreover, some of these features can only be detected at run time, which may seem like the user will need to run the harmful code and get infected in order to detect it. Fortunately, browser plug-ins can be easily developed to do a kind of test run of the suspicious content in a controlled manner before displaying to the user. Such a solution will suggest security versus response time tradeoff, which can be given as a “settings” option to the end user.

### 4. BUILDING THE CLASSIFICATION MODEL

Figure 2 shows the steps taken to build the classification model. We first collect and process the classification data (section 4.1), extract features of malicious code (section 4.2), and finally train the classifiers (section 4.3).

#### 4.1. Data Collection

In analyzing malicious JavaScript code (see Figure 2a), we collected two types of data: benign and malicious. For the latter, we used data from PhishTank, which contains about 3,000 malicious Web sites URL information. We trimmed this list to about 100, as many of the Web sites in the original list are closed, moved, or renamed. As for the benign list, we used data from the well-known Alexa Web information database.

#### 4.2. Feature Extraction

The next step is to extract the previously identified features from the collected JavaScript codes and
generate the features vector. For features 1.a and 1.b, the URL and IP will be mapped into string values in the vector. These values are set based on whether the IP or URL is blacklisted. To implement feature 2.a, we generate a list of the top most frequent words in both malicious and benign JavaScript-based Web sites. To this end, we used the Sphider search engine. Sphider accepts URLs and extracts the top most frequent words that appear in the input URLs. The same procedure is done for the list of trusted Web sites.

As for the remaining features, there are some useful JavaScript analysis tools in the literature that we have used (JavaScript debugger, n.d.; Firebug, n.d.). We used Firebug in detecting features 3.a, 3.b, and 3.c. Firebug is a suite of Web development tools that gave us the capability to monitor, analyze, and edit Web pages. Feature 3.d is detected by parsing the JavaScript code. Moreover, we used the JavaScript Debugger in analyzing the code content with respect to its size-behavior as indicated in feature 4.a. This Firefox add-on gave us the capability to debug the JavaScript code during run time.

4.3. Classifier Training

The Waikato Environment for Knowledge Analysis (WEKA) toolkit (Weka 3, n.d.) and its associated algorithms were used to generate four classification models: Neural Networks (NN), Decision Tree (DT), Support Vector Machine (SVM), and Naïve Bayes. These classifiers are trained and later used to check the input JavaScript code at run time (see Figure 3). The WEKA input is required to be in a certain file format; we used the “.arff” (Attribute-Relation File Format) file format. A vector of the eight aforementioned features was generated for each input sample. The URL and IP features were of type String while all other features were of type Boolean. Five-fold cross-validation was used to train the classification algorithms. In this training method, the training data are repeatedly divided into five parts of equal size, out of which four are used for training and one for testing.

4.4. Implementation Details

In this section, we go through some of the implementation-specific details not previously covered. In extracting the values of features, step 5 of Figure 3, some thresholds need to be defined and set based on the input. These thresholds are implementation specific and can be provided as a “control settings” option for the user. The user then decides based on his/her security preferences. As for our implementation, we chose the following values in setting the corresponding variable in the features vector; an example of the features vector is shown in FIGURE 4.

- For feature 2.a, we check the top 30 most frequent words in the output of the JavaScript output. The feature is flagged as true if the code shares more than two words with the corresponding list of most frequent words in the malicious data set (see Tables 1 and 2). Previously identified shared words (e.g., the word “privacy”) are excluded from this check.
- As for feature 3.a, the Boolean value in the features vector corresponding to this feature will be set as malicious if the eval() and unescape() function calls each appear once.
- Regarding feature 3.b, the JavaScript code is checked if it uses DOM tree to access other HTML documents. If it does, then the corresponding flag is set as malicious. Remember that DOM is a platform and an interface that can be used by many benign codes. However, its existence in conjunction with the other features constitutes enough circumstantial evidence to decide that the JavaScript code is malicious.
- Feature 3.c is decided by checking if the JavaScript code uses DOM XMLHttpRequest object to communicate with dubious external servers.
FIGURE 4  An example of the features vector.

• Feature 3.d is decided by checking if the codes make redirections to other malicious Web pages. Such a check can be conducted by verifying if the code uses methods such as `document.location(attakWebsite.com)`, as previously mentioned.

• For feature 4, the code is run offline and in a controlled environment to check the code size before and after execution. If the code maintains its size, then it is considered benign when setting the corresponding feature value.

Using Apache server and PHP scripting language, an interface was built to insert features data into a Mysql database. After that, another PHP-based tool called PhpMyAdmin (http://www.phpmyadmin.net/home page/index.php) was used to export data in Comma Separated Values (CSV) format. Another code was written to transform the CSV files into the WEKA-required “.arff” format.

5. PERFORMANCE EVALUATION

5.1. Experimental Setup

We present the results of subjecting our proposed malicious JavaScript code detection system to a test group of 100 malicious JavaScript code samples and 100 benign JavaScript code samples. The main goal of this performance evaluation is to study and analyze the detection correctness of the proposed solution and compare it to similar studies in the literature. Three independent types of studies were conducted. In the first one, the testing dataset is the same as the training dataset, which gives us an indicator of the classifiers’ ability to learn from the training set. Second, five-fold cross-validation is used for testing. This is a standard way of testing and evaluating machine learning approaches (Kolter & Maloof, 2006), where the data set is repeatedly divided into five subsets, of which four are used for training and one for testing. The final approach is 10-fold cross-validation, which is the method used by the authors in (Likarish et al., 2009). We implement it to compare the performance of our approach to theirs. As for the performance metrics, we are mainly concerned with the classification accuracy defined in Eq. (1) as:

\[
\text{Accuracy} = \frac{\text{#Correctly Classified Benign JS codes}}{\text{Total #Benign Samples}} \times 100\%
\]

Eq. (1) comes in two flavors per se, one for correctly classified benign codes divided by the number of benign samples. The accuracy can also be expressed in terms of the number of correctly classified malicious code divided by the total number of malicious samples. This definition is used in Figure 7. The same goes for the other performance metrics. From a benign code perspective, the False Negative (FN) is defined by the percentage of benign code classified as malicious (see Eq. (2)). The same logic applies to the malicious code perspective. While the False Positive (FP) ratio represents the percentage of malicious code classified as benign (see Eq. (3), which is the complement of the Accuracy and included here for the sake of completeness.

\[
\text{FN} = \frac{\text{#Benign JS codes classified as Malicious}}{\text{Total #Benign Samples}} \times 100\%
\]

\[
\text{FP} = \frac{\text{#Malicious JS codes classified as Benign}}{\text{Total #Benign Samples}} \times 100\%
\]

5.2. Results

Figures 5a and 5b show that the four classifiers are highly accurate when tested with same set as the training set, with 100% accuracy in most cases. The decision
Malicious code, testing set same as training set. (b) Benign code, testing set same as training set. (c) Malicious code, testing using 5-fold cross validation. (d) Benign code, testing using 5-fold cross validation.

FIGURE 5 Classification accuracy of the four classifiers. The input data is 100 benign and 100 malicious codes. Testing is done using five-fold cross validation or using the same training data. (color figure available online.)

tree approach is slightly less accurate in comparison to the other classifiers (97.1% accuracy for the malicious code). More importantly, testing using five-fold cross-validation results in very high accuracy (+95%) for all the classifiers except for the neural networks one, which performed badly with an accuracy of only ~40% (see Figures 5c and 5d). This is because in our evaluation, we used five-fold cross-validation (i.e., 20 vectors of 5 elements each). The Neural Network (NN) classifier was sensitive due to the small size of the data set. This resulted in low performance.

As for the false negative ratio, the same trend applies as before (see Figure 6). The ratio is low (less than 3%) for all the classifiers except for the neural networks, which is consistent with the accuracy results. We also compare our results to those of the classification-based malicious JavaScript detection approach in Likarish et al. (2009) using 10-fold cross validation (see Figures 7 and 8). Figure 7 clearly demonstrates the superior accuracy of our proposed scheme. Please note that Ripper is just another type of neural networks classifier that had been used in Likarish et al. (2009). However, in comparing our work to the literature (i.e., Likarish et al.), we used the same method employed by Likarish et al., which is 10-fold cross-validation (10 vectors of 10 elements each). Hence, Ripper was less affected by the size of the vectors and achieved the higher accuracy 97% (Figure 7). As for Figure 8, the results show that their approach is plagued with false negatives, which is nearly nonexistent in ours.

5.3. Discussion

5.3.1. Data Set and Classification Algorithms

The data set used in this paper is based on Phishtank, which collects Phishing Web sites. Similar evaluation can be done using other data sets that focus on drive-by or any other form of attacks, but this is out of the scope of this paper. As of the classification algorithms, WEKA provides a rich set of these, and we only reported those that achieve the highest performance.

5.3.2. Effect of Blacklisted URLs and IPs

Figure 9 shows that the accuracy drops negligibly when these features are excluded. In fact, the accuracy of the Neural Network classifier increases due to
its sensitivity to the irrelevance of these two features. The reason for these changes was pointed out in the paper introduction; these blacklisted URLs and IPs are outdated, as attackers may be quick in changing them.

Having an up-to-date blacklist of malicious Web sites definitely inundates the need for other features or any other approach. However, a JavaScript code from a not-yet-blacklisted URL or IP may redirect or communicate with a blacklisted entity. Thus, the check for communication and redirection may be needed as done with some of the proposed features.

5.3.3. Handling Obfuscation

Obfuscation is common in drive-by download attacks. We did not discuss thoroughly how this is handled, but typically deobfuscation needs to be done.
This problem has been discussed in (Cova et al., 2010), and not all of our features will require deobfuscation.

5.3.4. Features Extraction

The feature extraction is done automatically by parsing the http files. A code was written to perform this task.

5.3.5. Runtime Overhead

The runtime and response time overhead of the tool was noticed to be negligible, which is understandable given the huge processing power of modern PCs.

6. CONCLUSION AND FUTURE WORK

The JavaScript programming/scripting language is a ubiquitous tool in Web development. It is useful in adding features, effects, and improvements to end users as well as to the server-side of Web applications. However, much like any technology, such capabilities open the door for security exploits and adversaries. Distinguishing malicious JavaScript code from benign one is not an easy task for the typical user. In this paper, we have analyzed the behavior of malicious JavaScript code to point out its key features. Those features, when considered together, have been shown to distinguish malicious code from benign. To this end, we have built a classification model based on those features. Such a model can be readily integrated with a browser (e.g., as a plug-in) to protect end users from malicious JavaScript code quite effectively. Performance evaluation results show that our proposed approach is quite effective in pointing out malicious JavaScript code. However, as attackers get more sophisticated, there is a continuous need for analysis and update of malicious code features and distinguishing footprints. Future work will focus on coming up with new distinguishing features. Also, there is a need to measure the performance impact of the proposed scheme on the perceived Web browsing experience.
REFERENCES


BIOGRAPHIES

Mohammad Fraiwan received his B.S. (with honors) degree in Computer Engineering from Jordan University of Science and Technology (JUST), Jordan, and Ph.D. degree from Iowa State University (ISU), USA, in 2003 and 2009 respectively. He is currently an assistant professor in the Computer Engineering Department at JUST. His main research interests include network monitoring, biomedical applications, and data mining. He worked at Akamai Technologies, Boston, in summer of 2008, and as a visiting scientist at ISU during the summers of 2009 and 2010.

Rami Al-Salman is currently a Ph.D. student in Germany. In Spring of 2011, he received his Masters degree from the department of Computer Engineering at Jordan University of Science and Technology (JUST). He received his B.S. degree in Computer Information Systems from JUST. He worked at the Heinrich-Heine University in Duesseldorf, Germany during the summer of 2010. His research interests include security, data mining, and Information Assurance.

Natheer Khasawneh has been an assistant professor in the Department of Software Engineering at Jordan University of Science and Technology since 2005. He received his B.S. in Electrical Engineering from Jordan University of Science and Technology in 1999. He received his Masters and Ph.D. degrees in Computer Engineering from the University of Akron, Akron, Ohio, USA in the years 2002 and 2005 respectively. His research interests are data mining, biomedical signals analysis, software engineering and web engineering. Currently he is the vice director of the Computer and Information Center at Jordan University of Science and Technology.

Stefan Conrad is currently a full professor in the Institute of computer Science at the University of Düsseldorf, Germany. He received his Ph.D. in Computer Science from the Braunschweig University of Technology, Germany in 1994. His research interests include database design, knowledge discovery, data mining, and multimedia information retrieval.