Journal of Organizational and End User Computing
Volume 32 • Issue 4 • October-December 2020

Designing a XSS Defensive Framework for Web Servers Deployed in the Existing Smart City Infrastructure

Brij B. Gupta, National Institute of Technology, Kurukshetra, India & Asia University, Taiwan & Macquarie University, Australia

Pooja Chaudhary, National Institute of Technology, Kurukshetra, India

https://orcid.org/0000-0003-0766-0530

Shashank Gupta, Birla Institute of Technology and Science, Pilani, India

ABSTRACT

Cross-site scripting is one of the notable exceptions effecting almost every web application. Hence, this article proposed a framework to negate the impact of the XSS attack on web servers deployed in one of the major applications of the Internet of Things (IoT) i.e. the smart city environment. The proposed framework implements 2 approaches: first, it executes vulnerable flow tracking for filtering injected malicious scripting code in dynamic web pages. Second, it accomplished trusted remark generation and validation for unveiling any suspicious activity in static web pages. Finally, the filtered and modified webpage is interfaced to the user. The prototype of the framework has been evaluated on a suite of real-world web applications to detect XSS attack mitigation capability. The performance analysis of the framework has revealed that this framework recognizes the XSS worms with very low false positives, false negatives and acceptable performance overhead as compared to existent XSS defensive methodologies.

KEYWORDS

Smart City Cyber Security, Trusted Remark Statement Injection, Untrusted Javascript Code, XSS Attack

1. INTRODUCTION

Urbanization and migration require global development of economic, social, institutional and physical infrastructure. Consequently, it puts pressure on the city's organization as request for resources like education, healthcare, transportation, government, and safety exceed their availability. To overcome these issues, cities are focusing on the utilization of technology i.e. becoming 'smart'. Smart cities (Ferraz & Ferraz, 2014; Seth, 2013) are the cities that harness Information and Communication Technology to automate and enhance services for improving the living standard of their citizens and attain sustainable development. This concept of "smart cities" is the outcome of the new computing paradigm, that is, Internet of Things. Internet has been risen up to the level where everything nearby us is connected and turns out to be part of some form of network. Informally, we can define IoT as a network formed by devices capable of generating, sending and receiving information related to any business, accesses by any person, any time irrespective of the geographical location. Technology should be used to make cities smart in terms of the services provided such as smart traffic control,

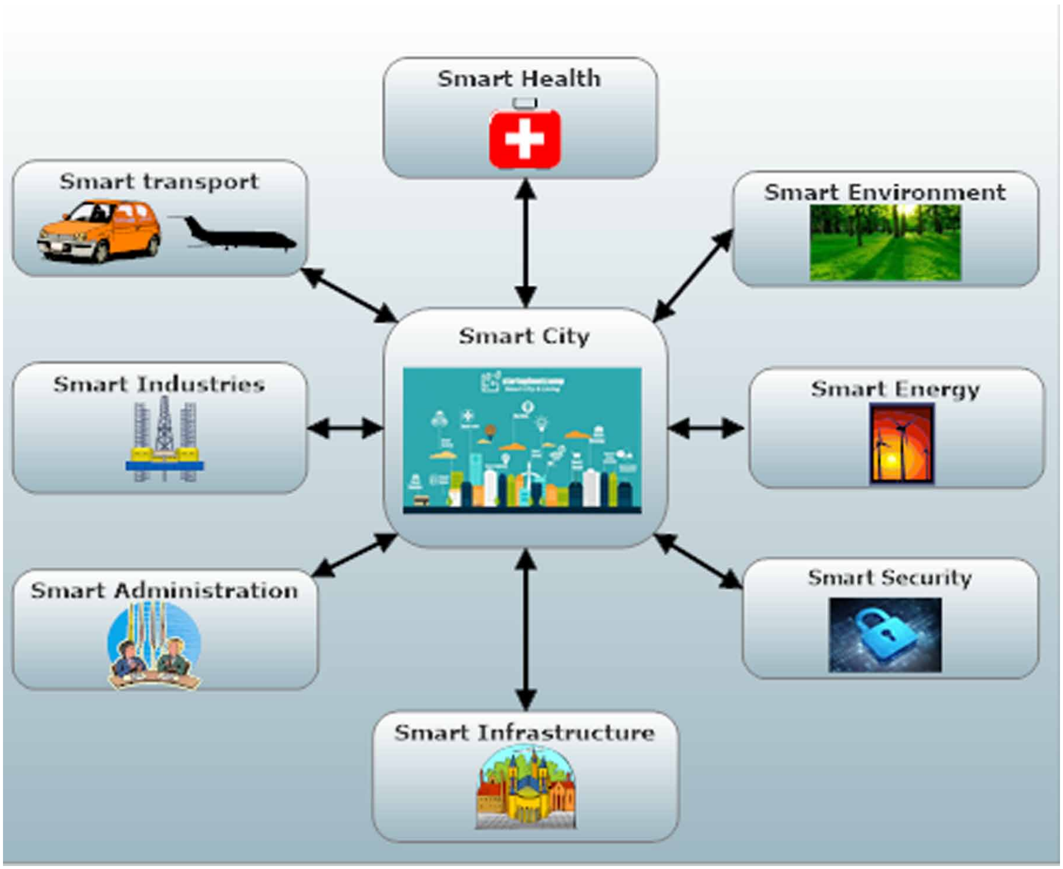
DOI: 10.4018/JOEUC.2020100105

This article, originally published under IGI Global's copyright on October 1, 2020 will proceed with publication as an Open Access article starting on January 21, 2021 in the gold Open Access journal, Journal of Organizational and End User Computing (converted to gold Open Access January 1, 2021), and will be distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

smart parking, smart health-care, smart transportation, smart city management system like waste management, water management, Smart Street lighting and so on (Hossain & Shamim, 2018; Li, & Daming, 2019). Therefore, in a nutshell, smart city means everything is embedded with sensors to enable them to interact with the environment. Smart cities comprise of some of the main components as illustrated in Figure 1. Indeed, smart city concept has given a new direction for nation's growth; nevertheless, for the exchange of the data, it utilizes server infrastructure which brings some major challenges also. Cyber security is the biggest challenge because people share large amount of information comprising personal and professional over the Internet (Li, Jianzhong, 2018; Almomani, Ammar, 2013; Parada, Raúl,2018; Drennan, Judy, 2019). There are numerous cyber-attacks that have contaminated web application.

Figure 2 shows detailed architecture of the smart city and also shows what type of attacks are launched at which layer in the architecture of smart city. It may include DDOS, phishing, XSS, SQL injection, spamming etc. Code injection vulnerabilities are the most common and dangerous threat on Internet. It includes Cross-Site Scripting (XSS) (Chaudhary, Gupta, & Gupta, 2019; Gupta & Gupta, 2016a, 2016b, 2018b), SQL injection, etc. XSS attack (Chaudhary, Gupta, & Gupta, 2016;) is a type of code injection attack in which adversary injects malicious script code into the source program of the web application, triggers malicious actions like cookie stealing, session hijacking, dis-information and so on. It covers 3 types: Persistent XSS (Gupta & Gupta, 2018d, 2018e, 2018f),

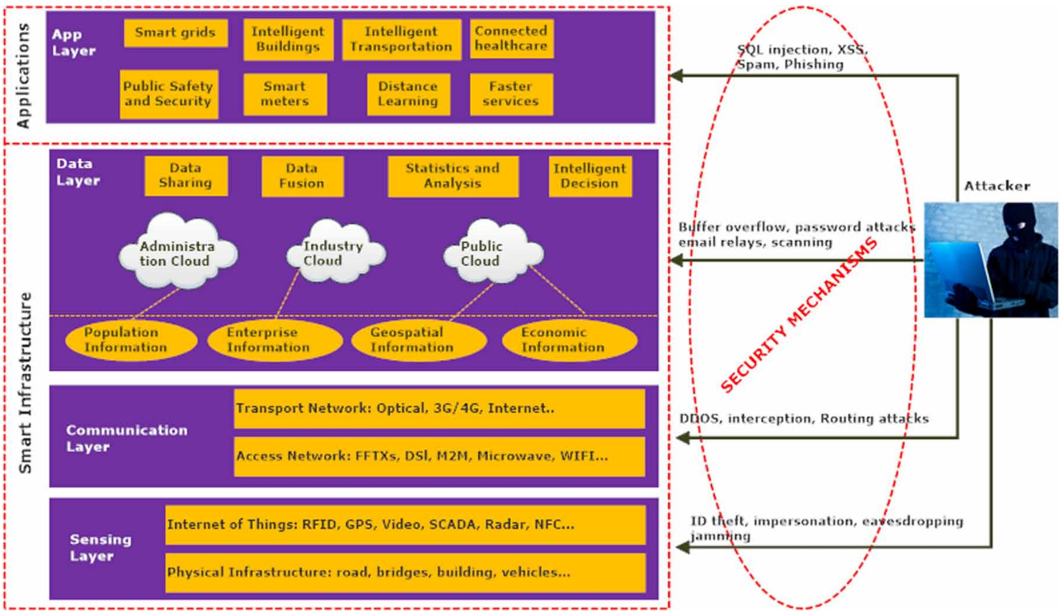
Figure 1. Smart city components



Reflected XSS (Gupta et al. 2017a, 2018c) and Document Object Model (DOM) based XSS attack (Gupta, Gupta, & Chaudhary, 2018a).

Numerous XSS defensive solutions have been proposed by the researchers for detecting and alleviating the effect of XSS vulnerabilities from the different platforms of Web applications. XSSFilt (Pelizzi & Sekar, 2012), a client-side XSS filter could discover non-persistent XSS vulnerabilities. This filter identifies and thwarts portions of address URL from giving an appearance in web page. This filter could also discover partial script insertions. XSS Auditor (Bates, Barth, & Jackson, 2010) is a filter that realizes equally extraordinary performance as well as high accuracy via jamming scripts following the HTML parsing and prior to execution. The filter can simply spot the components of the response which are considered as a script. BIXSAN (Chandra & Selvakumar, 2011) comprises of JavaScript Detector, which discovers the existence of JavaScript, an HTML parse tree producer which is used to diminish the inconsistent performance of web browser and for the recognition of static script tags for permitting legitimate HTML source code. ScriptGard (Saxena, Molnar, & Livshits, 2011) is a complementary technique that presumes the collection of accurate sanitizers and injects them to match the parsing context of web browser.

Figure 2. Architecture of smart city with possible threats


Applications
App Layer
Smart grids
Intelligent Buildings
Intelligent Transportation
Connected healthcare
Public Safety and Security
Smart meters
Distance Learning
Faster services
Smart Infrastructure
Data Layer
Data Sharing
Data Fusion
Statistics and Analysis
Intelligent Decision
Administration Cloud
Industry Cloud
Public Cloud
Population Information
Enterprise Information
Geospatial Information
Economic Information
Communication Layer
Transport Network: Optical, 3G/4G, Internet..
Access Network: FFTXs, DSl, M2M, Microwave, WIFI...
Sensing Layer
Internet of Things: RFID, GPS, Video, SCADA, Radar, NFC...
Physical Infrastructure: road, bridges, building, vehicles...
SQL injection, XSS, Spam, Phishing
Attacker
Buffer overflow, password attacks email relays, scanning
SECURITY MECHANISMS
DDOS, interception, Routing attacks
ID theft, impersonation, eavesdropping jamming

An automated technique proposed by (Livshits & Chong, 2013) of sanitizer placement by statically analyzing the stream of infected data in the program. However, placement of sanitizer is static and sometimes changes to dynamic wherever required. JSand (Agten et al., 2012) is a server-driven JavaScript-based sandboxing support, which implements a server-specific policy on the injected scripts with no requirement of filtering or modification of scripts. The technique facilitates the developer of a website to safely incorporate third-party scripts, with no requirement of disorderly alterations to both client and server-side infrastructure. XSS-Guard (Bisht & Venkatakrishnan, 2008) is a technique that detects the collection of scripts that a web application intends to create for any HTML web request. The technique creates a shadow web page to learn the web application's intent for every HTTP web response, including the legitimate and expected scripts. Any divergence between the real generated web page and the shadow web page points towards the possible script inclusions. However, main issue with these existing techniques is that they cannot solve the problem of isolating executable untrusted JavaScript code from the remaining data of HTML web page. Moreover, these are not able to effectively make distinguish between valid and injected JavaScript code in the web page. Some of the existing techniques demands major alterations in the existing infrastructure of Web applications.

1.1. Key Contributions

On the basis of these issues, authors have designed a framework based on vulnerable flow analysis and injection of trusted Remark statements in the web page. At the server-side, our framework performs 2 main functions: classification of response web page into static and dynamic web page; and injection of trusted remarks statements at the borders of valid JavaScript code present in the web page. These remarks help in differentiating malicious JavaScript from the valid JavaScript as it includes features of valid JavaScript in the form of protocols with randomly generated nonce. These protocols form the basis of comparison between JavaScript codes to detect XSS attack. At the client-side, it detects the XSS attack by applying different techniques for static and dynamic web page. For static response, it, firstly, extracts scripts and then make a comparison with the remark. If variance is found then, it indicates XSS attack. For dynamic response, initially, it identifies vulnerable source by completing vulnerable flow analysis. Then, it determines the context of this malicious source followed by applying filtering on it with the help of filtering APIs. Finally, modified response web page is displayed to the user.

1.2. Outline of this Paper

The rest of the paper is organized as follows: In section 2, we have introduced our work in detail. Implementation and evaluation of our work are discussed in section 3. Finally, section 4 concludes our work and discusses further scope of work.

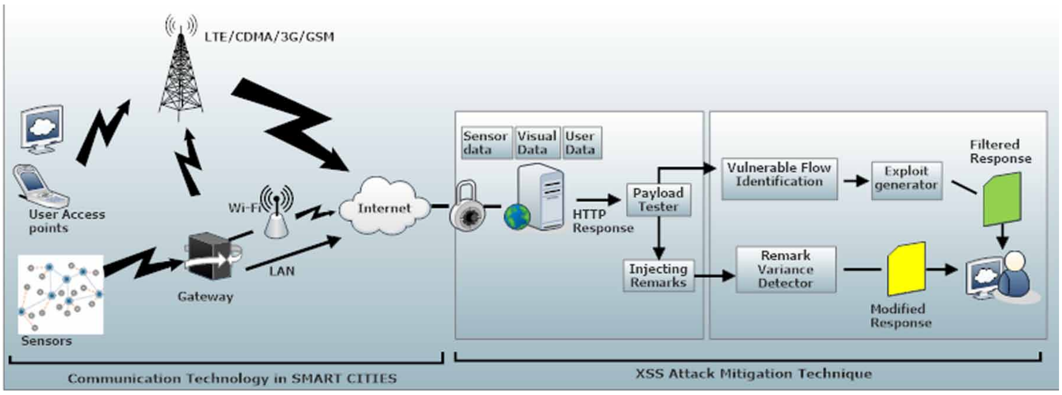
2. PROPOSED WORK

This paper presents a hybrid defensive framework to protect the Internet users from XSS attack, in smart city environment. The server-side framework, initially, explores the requested web application to extract all the web pages of that web application. Then, it statically analyzes the content of the web page and classifies them into 2 types: dynamic web page and static web page, depending on the presence of input field in the web page, respectively. Furthermore, it returns the dynamic web page to the browser and injects remark statements at the beginning and ending of the valid JavaScript code present in the static web page. This is done to ensure the proper discrimination between valid and malicious JavaScript code. The client-side framework dynamically executes the taint tracking to identify the part of the malicious injected string used in the sensitive functions in the source code of the dynamic webpage. Then, it substitutes the tainted string value with the testing attack vector to exploit XSS vulnerability. If attack is successful then, it filtered out the tainted source with the filtering APIs. In addition, it checks for the validity of remarks statement in the static web page to detect XSS attack. The next sub-section discusses the abstract overview of our framework.

2.1. Abstract Design View of the Framework

The proposed framework works in four main phases: 1) statically classifies the web pages of the web application; 2) generates remark statements comprising randomly generated nonces and features of valid JavaScript code block; 3) checks for the validity of injected remark statements, at the client-side, to detect XSS attack.; 4) dynamically performs the taint analysis and performs the filtering on the tainted string value with the filtering APIs. Figure 3 elaborates the abstract design overview of our framework.

Figure 3. Abstract design overview of client-server framework



This framework alleviates the propagation of XSS worm by performing two main mechanisms: dynamic taint analysis and remark statements injection and validation. Payload tester accomplishes the classification of web pages into dynamic and static web pages. Hereinafter, dynamic web page undergoes taint analysis procedure with the help of vulnerable flow identification component. Exploit generator component is used to identify suspicious web page with the help of inserting testing attack payload at the taint source location and launch the XSS attack. If attack is successful then, it filters the malicious string value. Otherwise, dynamic web page is free from XSS attack. Static web pages are examined for the identification of malicious injected JavaScript code. This is done by injecting remark statements at the border of the valid JavaScript code block. Remark variance detector seizes the web page and implements a number of tests to detect XSS attack. Firstly, it identifies if any JavaScript code block without remark statement is present or not. If present then, it is considered as injected code and removed from the response. Otherwise, it tests for the validity of remark statements comprising random nonce and features of valid JavaScript code. If a nonce is incorrect then, it is declared as injected code. Otherwise, it matches the suspected features of JavaScript code with the features included in the remark statements. If any deviation is found then, it is considered as injected. Finally, it checks for the presence of any duplicate remark statements to identify remark statements inserted by the attacker. If it identifies the presence of the injected JavaScript code in the response then, it is removed from the response web page along with the injected remark statements. Figure 4 highlights the detailed working procedure of our framework in the form of flow chart. Hereinafter, the next sub-section highlights the detailed illustration of our framework.

2.2. Detailed Design View of the Proposed Framework

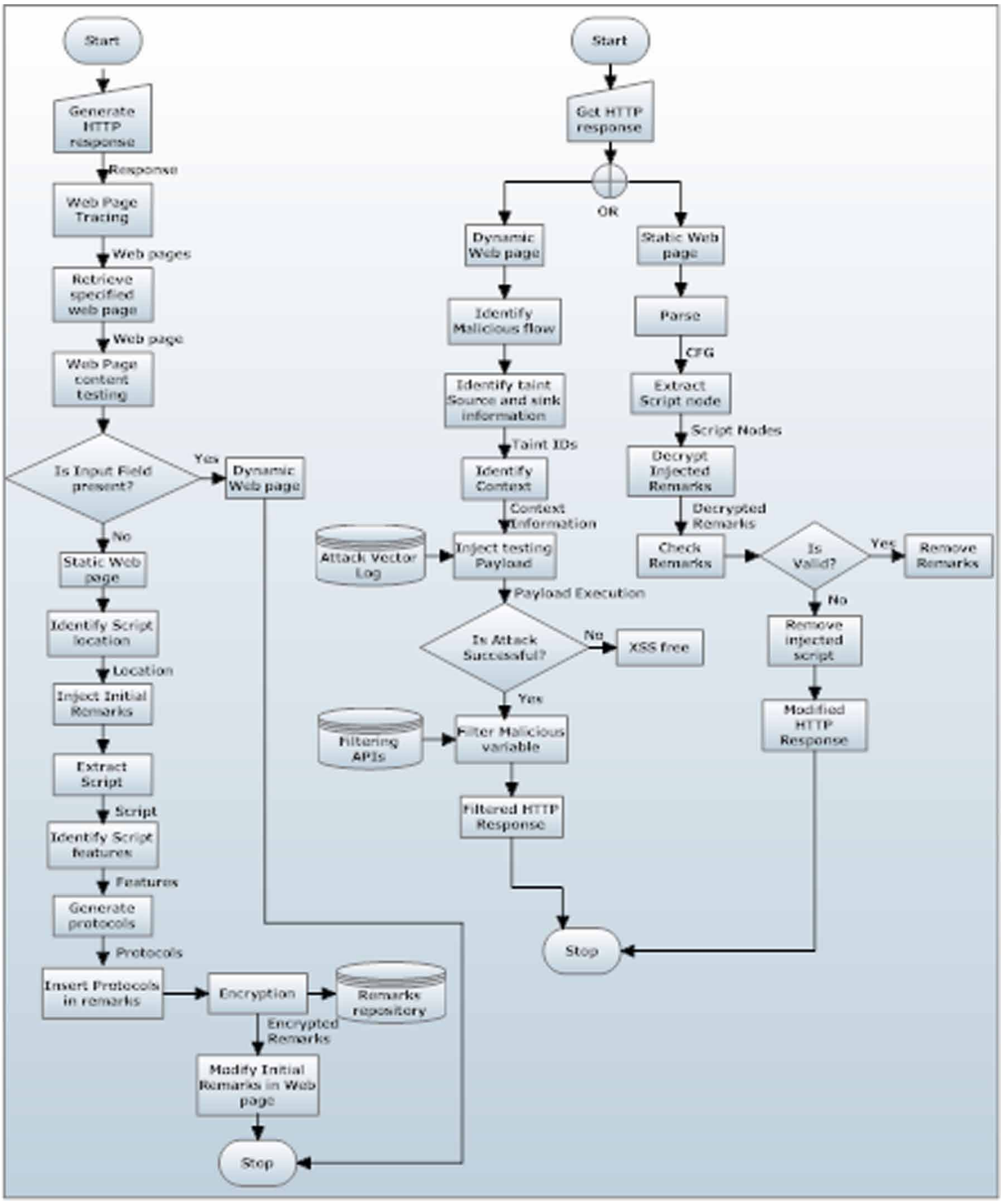
This section furnishes the comprehensive architectural detail of our hybrid framework. Figure 5 shows the detailed design overview of our client-server XSS defensive framework. The outlined framework executes in four high-level phases: 1) Classification of HTTP response web pages; 2) Remark generation; 3) Remark validation; 4) Exploitation and filtering phase.

2.2.1 Classification of HTTP Response Web Page

The key goal of the server-side implementation is to efficiently classify the generated HTTP response web page and insert remark statements at the borders of the JavaScript code. The key components which implement these operations are: Internal Web page Tracing, Web page retrieval, Payload tester.

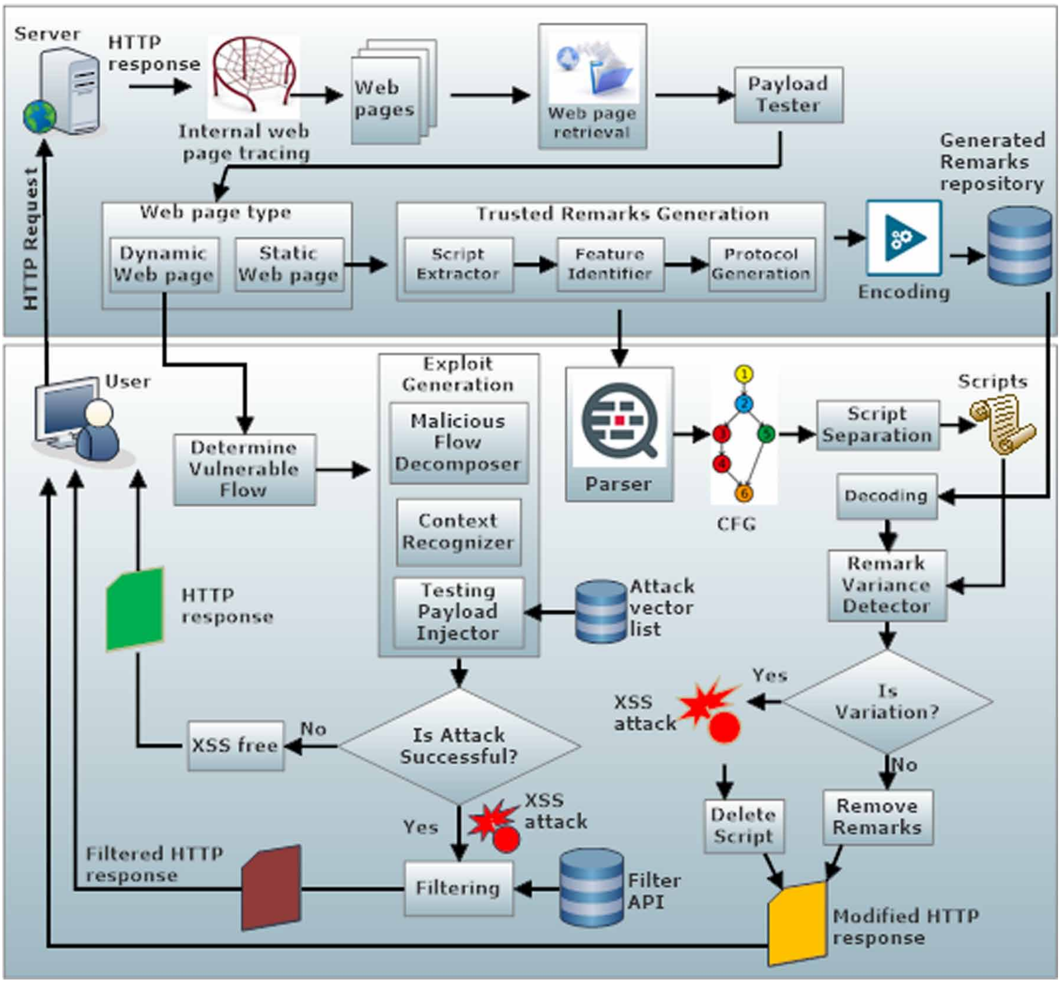
2.2.1.1 Internal Web Page Tracing

Figure 4. Flow chart of our proposed framework



It is a server-side component that implements scanning of the web page to extract all the web pages of the requested web application and save them for the later processing. Once the client enters the requested URL then, it inevitably crawls the web application with the help of a selenium-based crawler. Initially, it uproots all the internal and external URI links from the response web page and then makes a request to retrieve all web pages from the server.

Figure 5. Comprehensive design overview of our proposed framework



2.2.1.2. Web Page Retrieval

It is a server-side component which is responsible for the extraction of requested web page of the web application. This component receives the log provided by the internal web page tracing component as its input. Then, it checks the requested URL to identify the specified web page of web application as requested by the user. Finally, it extracts the web page from the log and supplied it to the other component for later processing.

2.2.1.3. Payload Tester

Its main aim is to classify the web page into two categories: dynamic web page and static web page.

2.2.1.3.1. Dynamic Web Page

Web page which contains any type of input field such as search box, comment box, form fields and so on. Payload tester has classified these web pages as dynamic because user can enter untrusted input value into the input field. For instance, web application demanding user to fill a form regarding personal information via a web page.

2.2.1.3.2. Static Web Page

Static web pages are the web pages which are read only. It does not contain any input field to receive user input. For instance, web page containing a product specification.

If the requested web page falls under the category of dynamic web page then, it is returned to the client for further processing. Otherwise, static web pages are forwarded to the trusted remark generation component for later processing.

2.2.2. Remark Generation Phase

This is the second phase which receives static web page. The key motto of this phase is to generate the trusted remark statement that is to be injected at the beginning and ending of the valid JavaScript code block. This is done to ensure that at the time of execution at the client-side, browser is able to distinguish between legitimate and malicious JavaScript code injected by the attacker. To accomplish this, firstly, we uproot all the legitimate JavaScript code embedded in the web page. Then, it analyzes each JavaScript code to find out the unique features and embed them in the protocol. These protocols are injected into the response web page at the starting and ending of the valid JavaScript code block in the encrypted form. The key modules which perform this functionality are: Trusted Remark generation and encoding. These are described below:

2.2.2.1. Trusted Remarks Generation

The trusted remark statement comprises of a randomly generated nonce and extracted features of the valid JavaScript code embedded in the static web page. These features are wrapped into the protocols and it injects these protocols in the remarks statement for execution time checking at the client-side. This module comprises the following key components: Script Extractor, feature Identifier and Protocol Generation.

2.2.2.1.1. Script Extractor

It is a server-side component which performs two main functionalities: first, extraction of legitimate JavaScript code embedded in the static web page. Second, injection of initial remark statements at the starting and ending of extracted JavaScript code block. To accomplish its first task, it utilizes HTML parser i.e. HtmlUnit [34], to parse web page and extract JavaScript code and store them in a separate log. To complete its second task, it inserts the initial remark statement (comprises of randomly generated nonce) in the extracted JavaScript code. To achieve this, it modifies the locations in the source code of the web page where JavaScript code is present. Then, it stores the modified code into the original source code of the web page. Here, authors have examine 5 cases where JavaScript code may be present in the web page: 1) inline scripts; 2) scripts inclusion via remote source file; 3) scripts inclusion via local external source file; 4) scripts inclusion via event handling code; 5) script inclusion via URL attribute value. Table 1 explain these cases and also show how it injects the remark statements in its initial stage. First example, shows a inline script inclusion with method named as *documents.cookie("ABC").* It injects remark statement at the starting and ending of the JavaScript code block. Initially, remark statements comprise of a randomly generated nonce (a 32-bit number) as /*N1*/. Remotely accessible JavaScript file cannot be modified; therefore, it injects remark statement with <script> tag to convert them into inline script code (example 2). JavaScript code included in the local external file is handled separately. So, no remark statement will be injected for them (example 5). This will reduce the overhead of injecting remark statements.

2.2.2.1.2. Feature Identifier

It is a server-side component which receives extracted JavaScript code log as its input. The aim of this component is to analyze each script in the log to identify the unique features. These identified features are then used for the protocol generation. To inject malicious JavaScript code, attacker either

Table 1. JavaScript code with initial remark statement injected with different source type

Source type	Code instance	Code with initial remarks
Inline	<script> document.write(“ABC”); </script>	<script>/*N1*/ document.write(“ABC”); /*N1*/</script>
Script call (remote site)	<script src= “http//www.example.com/exm.js”></script>	<script>/*N1*/</script> <script src= “http//www.example.com/exm.js”></script> <script>/*N1*/</script>
Event Handling	<body onLoad= “alert (“ABC”);”> ….. </body>	<body onLoad=/*N1*/alert (“ABC”);”/*N1*/> … </body>
URL property value	<a href= “javascript:window.alert(“ABC”)”>	<a href= /*N1*/ javascript:window.alert(“ABC”) /*N1*/>
External script injection	<script src= “external.js”></script>	<script src= external.js></script>

modifies the JavaScript function definition written by the programmer or injects a function call to maliciously written function. For instance, consider the code snippet as shown below:

```
<input type=”text” name= “username” value=”<%request.
getParameter(“U_name”)%>”>
```

Here, no filtering mechanism is applied on the username before it is used in the response web page. Thus, this field is vulnerable to the XSS attack. Suppose, an adversary injects a malicious function as: *<script>alert(“document.cookie”);</script>*. Therefore, the original code becomes *<input type=”text” name= “username” value=”<script>alert(“document.cookie”);</script>”>*. Consequently, when browser renders this response then attacker gets cookie information of the user. Therefore, function call and function definition patterns are extracted out from the valid JavaScript code, as its unique feature. Table 2 illustrates some of the examples related to the probable features of the valid JavaScript code including function call and function definition features. For example, first example describes the inbuilt function call as Math.pow(4,5), we represent the probable features as *{pow,2,4,5}*. It means function ‘pow’ has 2 parameters 4, 5. Similarly other examples are shown.

2.2.2.1.3. Protocol Generation

It is a server-side component which is responsible for the encapsulation of extracted JavaScript features in protocol. These protocols are then included in the initial remark statement. This is to ensure that legitimate JavaScript present in the response web page can be properly distinguished from the injected JavaScript code by comparing their features with ones stored in protocol. Table 3 describes the script code, protocol generation and remark generation. Protocols are stored by using Protocol ID, type, name and paramcount. According to the number of parameters, param field stores the actual parameters. Modified remarks statement comprises a nonce and protocol ID as /*N1, 1*/. In function call type, instead of paramcount, we use argcount and arg fields.

2.2.2.2. Encoding

This is the last working component at the server-side of our framework which preserves the integrity of the remarks statement i.e. protocol. If raw form of remark is exposed to browser then, it might be possible that an attacker modifies the protocols so that it reflects features similar to legitimate one but actually are malicious. To address this problem, we transform protocols into encoded format and

Table 2. Extracted probable features of the valid JavaScript code

Type	Example	Probable features
Inbuilt function call	Math.pow (4,5)	{pow,2,4,5}
User defined function call	function active(a, b, c){..};	{active, 3,a,b}
Nested method call (user defined)	pro(3, pro(6,7))	{pro,2,3,{pro,2,6,7}}
Nested method call (inbuilt)	Math.pow(2, Math.min(3,4))	{pow, 2, 2, {min, 2, 3, 4}}
Anonymous function call	var X= pro (a, b){…};	{X, 2, a, b}

Host object method call	Var ID=document.getElementByName("value");
	ID.innerHTML= "hello world";
		{document.getElementByName, 1, value}

Table 3. Protocol generation with modified remark statement

Script code	Protocol generation	Modified remarks
<script> var x= product(2,3); </script>	<protocolID>1</protocolID> <type>def</type> <name>product</name> <paramcount>2</paramcount> <param>2</param> <param>3</param>	<script>/*N1,1*/ var x= product(2,3) /*N1,1*/ </script>
<body onLoad= "active (a, b)">… </body>	<protocolID>2</protocolID> <type>call</type> <name>active</name> <argcount>2</argcount> <arg>a</arg> <arg>b</arg>	<body onLoad= /*N2, 2*/"active (a, b)" /*N2,2*/>..</body>
<a href= "javascript:window.alert (document.cookie)">	<protocolID>3</protocolID> <type>call</type> <name>window.alert</name> <argcount>1</argcount> <arg>document.cookie</arg>	<a href= /*N3, 3*/"javascript:window.alert (document.cookie)" /*N3, 3*/>

then embed them into remark statement. It uses Base 64 encoding representation to achieve our goal. It stores all encoded protocols injected in remark statement into a repository for processing at the client-side. These protocols are decoded at the client-side before comparison of features.

All the, aforementioned, phases are implemented at the server-side. In the entire process at the server-side, initially, we classify the response web page into dynamic and static web page. If web page is categorized as dynamic web page then it is forwarded at the client-side. Otherwise, static web page undergoes trusted remark statement injection process. Then, modified static web page is returned to the user.

2.2.3. Remark Validation Phase

Heretofore, our proposed framework has effectively classified response web page and inject trusted remark statement into static web page. In this phase, we perform the analysis of the static web page to detect injected JavaScript code for the identification of XSS attack. This phase is accountable for the authentication of the remark statement. The key components to accomplish this task are: Parser, Script separation, Decoding, and Remark Variance Detector.

2.2.3.1. Parser

It is the client-side component which receives the static web page with the remark statement. It is responsible for construction of the Parsed Tree (PT) corresponding to that web page. It is to ensure that the browser renders the web page correctly. For instance, consider the following code snippet as shown in listing 1. In the above example, untrusted user input is applied at S_GET('name') and $_GET('age'). The parse tree generated for the above code snippet is shown in Figure 6. Each node of the tree represents HTML tags or text. This tree will be processed to determine script node embedded in to the web page.

Listing 1. Example shows vulnerable HTML code produced by vulnerable server.

```
<html>
<body>
<div name= "val" onClick= "my()"> Click Me!!! </div>
<script>
function my() {
document.getElementByName("val").innerhtml= "hello" + "$_
GET('name')" + "you are" + "$_GET('age')" + "years old";}
</script>
</body></html>
```

2.2.3.2. Script Separation

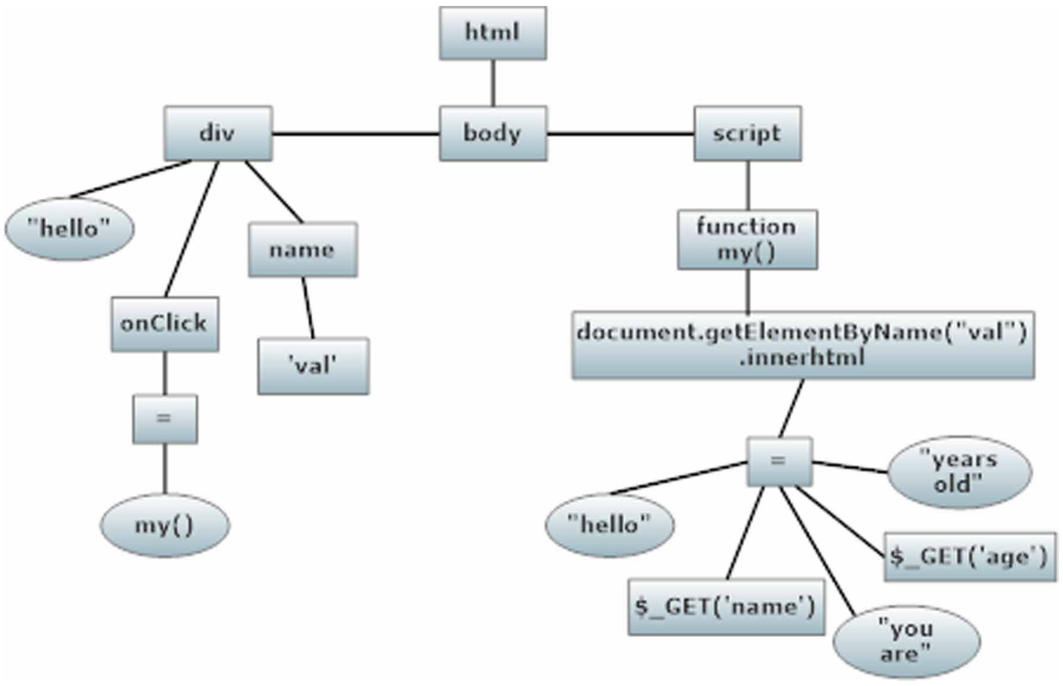
This component is a client-side process in which parse tree is processed to identify all JavaScript code with injected remark statement. As shown in listing 1, user provides the untrusted data by the $_GET['...'] variables. $_GET supposed to provide the name of the user who is currently logged in and $_GET provides the age of the corresponding user. As no sanitization procedure is applied on these variables and are directly processed by the browser, so, these are vulnerable to the XSS attack. Therefore, scripts nodes must be separated from the response page. To ease this problem, it searches in the parse tree to check for the opening and closing HTML tags. Then, for every couple of JavaScript tags (<script>…. </script>), it inspects each unique path between opening and closing tags by using graph traversal algorithm such as Depth First Search (DFS). Each identified path represents the JavaScript code that the web page code might result. Figure 7 illustrates the algorithm used for the Script separation. This algorithm works as follows: Script_log is a log maintained to store all possible recognized JavaScript code included in the web page. This algorithm processes Control Flow Graph (CFG) as follows:

It examines each node (v) in the graph (V, E) to check its content. If it is <script> tag, then it extracts the content of each node in Script_log that appears during the traversal of unique path, until a node with content </script> is found. Otherwise, ignore the content of that node. Finally, it outputs Script_log comprising of JavaScript code that a web page might result.

2.2.3.3. Decoding

It is a client-side process to decode the trusted remark statements injected at the borders of the valid JavaScript code present in the static web page. In the second phase of our framework, we perform encoding of the remark statement to ensure the integrity of the extracted features of valid JavaScript code, stored in remarks. In order to perform a comparison between these known features and script present in the response web page, there is a need to transform the encoded features into raw form. Therefore, this component performs the reversible process of encoding, at the client-side. Encoding, basically, builds up a mapping between content types from the features to encoded value and stores these mapping in a repository. In decoding process, this repository is used to which perform reverse

Figure 6. Parse tree generated for the code shown in listing 1



mapping i.e. encoded value to original content type. This component will produce decode features of the valid JavaScript code, as its output and it if forwarded to the next component for further processing.

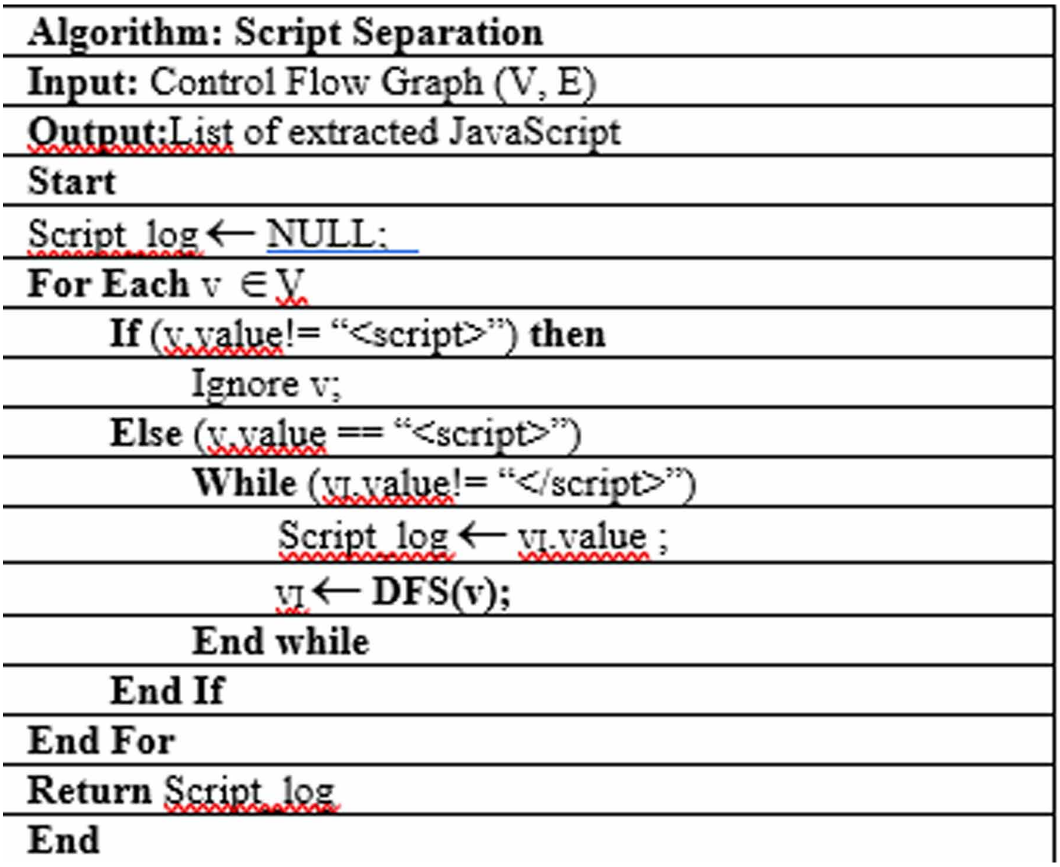
2.2.3.4. Remark Variance Detector

It is a client-side component which is responsible for detecting the variance in the script features extracted at client-side and the known features of the valid JavaScript code extracted at the server-side. It receives two things as its input: decoded features and extracted script code. Then, it checks for whether script without remark is present or not. If present then, it is injected malicious code. Otherwise, it checks for the remark validity. If nonce is invalid then, it is indicated as injected code. Otherwise, it checks for the validity of the embedded protocols (i.e. compare the features stored in it with the JavaScript extracted from the response page). If features are valid then, response is free from XSS attack, otherwise, it is considered as injected code. Finally, if the framework detected injected malicious JavaScript code then, it is removed from the response web page and modified response web page is displayed to the user. If no injected code is found then, framework removes the remark statements from the response web page and response is forwarded to the user. Figure 8 represents the entire working process of this component in the form of the flow chart.

2.2.4. Exploitation and Filtering Phase

This is the last phase of the framework at the client-side. The input to this phase is the dynamic web page. The key goal of this phase is to identify the location of the tainted source and the sensitive function where this value is used (i.e. determine the vulnerable points in the source code of the response web page). Moreover, it performs filtering on the taint source to negate the effect of the injected malicious script at the source point. Finally, filtered response is displayed to the user. The key Components to accomplish the entire functionality of this phase are: Determine Vulnerable flow, Exploit Generation, and Filtering.

Figure 7. Algorithm implemented for JavaScript separation



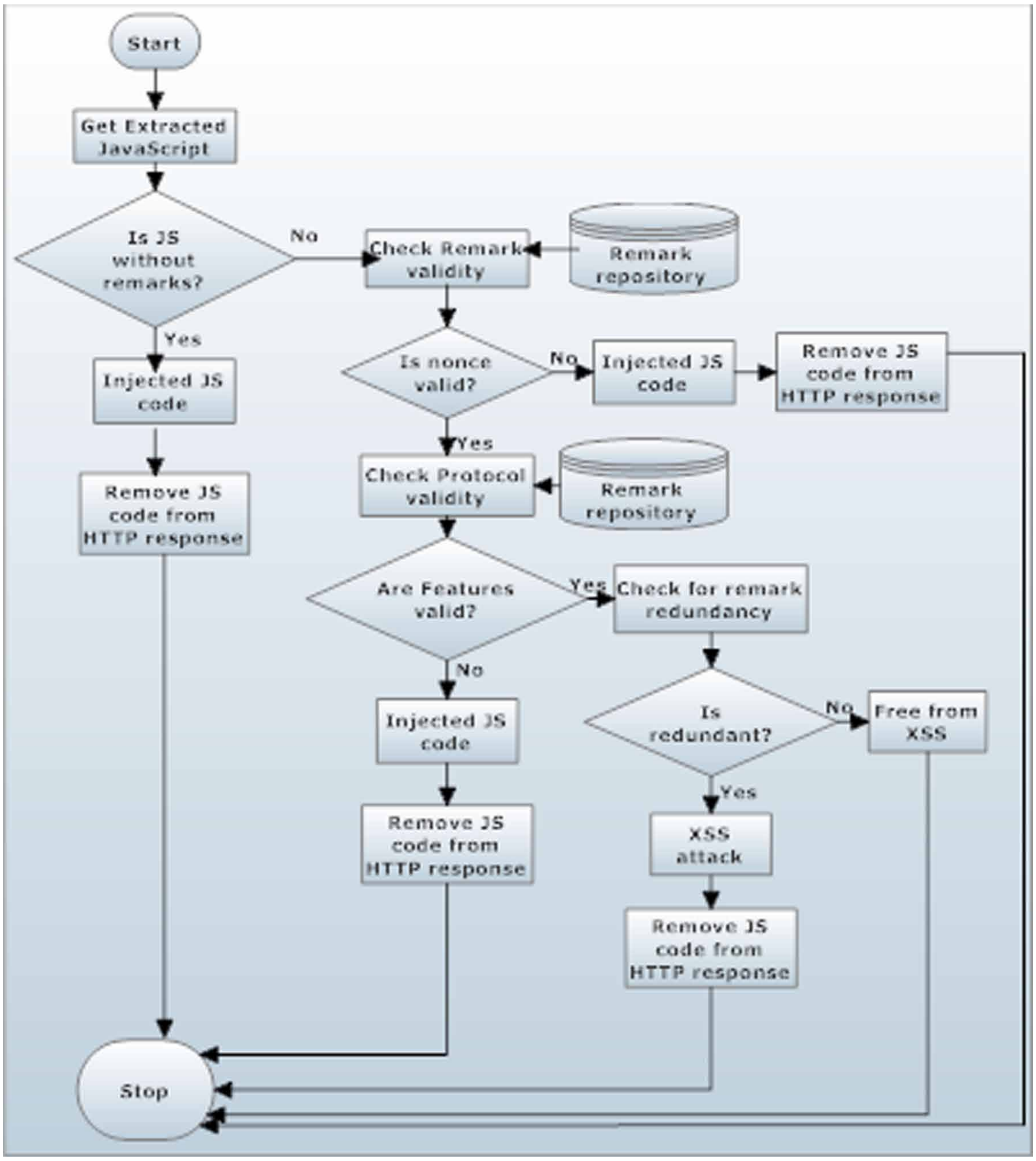
```
Algorithm: Script Separation
Input: Control Flow Graph (V, E)
Output:List of extracted JavaScript
Start
Script_log $\leftarrow$ NULL;
For Each v $\in$ V
    If (v.value!= "<script>") then
        Ignore v;
    Else (v.value == "<script>")
        While (vi.value!= "</script>")
            Script_log $\leftarrow$ vi.value ;
            vi $\leftarrow$ DFS(v);
        End while
    End If
End For
Return Script_log
End
```

2.2.4.1. Determine Vulnerable Flow

This component is responsible for the determination of the flow of untrusted user data from source to the sensitive function present in the response web page. Web application is marked as XSS free if it is possible to decide the output of web page statically (i.e. static JavaScript). Nevertheless, if dynamic JavaScript is generated by the web application that generates the improperly sanitized content, then, attacker can utilized this vulnerability to inject some malicious code. Table 4 illustrates the vulnerable source and sinks locations used in our framework. For example, consider the following code snippet shown in the Figure 9. In this code, uname is directly resulting from the parameter value name and is output to user deprived of any validation mechanism.

Adversary may exploit this vulnerability by injecting malicious code at name parameter like *"<script>alert(document.cookie);</script>*. Consequently, uname hold this JavaScript code and browser renders attacker's provided code. Hence, this component extracts tainted source and sink information as: taint source (*uname= request.getParametervalue("name")* and taint sink (*document.write(tag)*). This information is forwarded to the next step phase to determine the context and analyze it for the presence of XSS loopholes.

Figure 8. Flow chart showing the working process of the remark variance detector



2.2.4.2. Exploit Generation

This module examines the vulnerable flow to identify vulnerable source and sink. Then, it generates context-based testing attack payload that can be simply used to verify vulnerable webpages. It is achieved in 3 main steps: Malicious Flow Decomposer, Context Recognizer and Testing Payload injector.

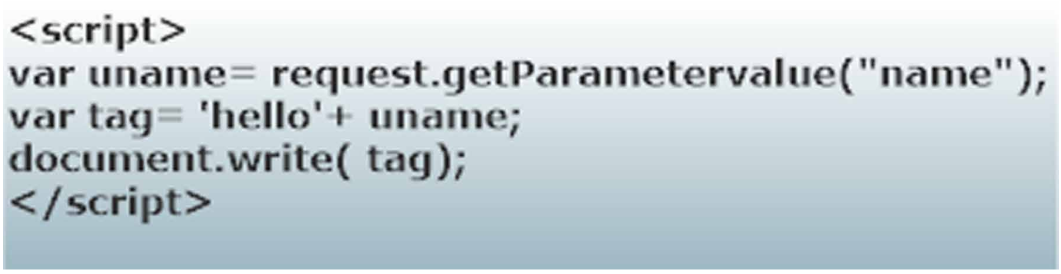
2.2.4.2.1. Malicious Flow Decomposer

It is the component which receives the logs that contains information about the vulnerable flow present in the web page. It extracts the following from the log: 1) Taint Source which contains the

Table 4. Vulnerable source and sink location

	Taint source	Taint sink
Location	document.URI, window.location, location. search, location.href, URL, document. location, baseURI, location.hash	location.href, location.pathname, location. port, location.protocol, location.search, location.assign, location.replace, location.hash, location.host
Link	href, media, rel, rev, type	link.innerhtml, link.namespaceURI, link.toString, style.
HTML	Iframe, image, body, div, audio, video, em, form, input, style, var	InnerHTML, outerhtml, src, action, value, toString, defaultChecked, checked, selectedIndex, rel, write, writeln, script.innerhtml, textcontent. Crossorigin.
JavaScript	Script, document, events.	Src, eval, setTimeout, setInterval, baseURI, document. URL, document.cookie, document.scripts, onClick, onLoad, onChange, onMouseOver.
Storage	Cookie, localStorage, sessionStorage	Cookie, localStorage, sessionStorage
History	-	Current, next, previous, length, toSring
Window	-	defaultStatus, status, location, frames, innerHeight, innerWidth, localStorage.

Figure 9. Code snippet


```
<script>
var uname= request.getParametervalue("name");
var tag= 'hello'+ uname;
document.write( tag);
</script>
```

source type and the string value in the source used in sink. 2) Taint Sink which defines the location where string value supplied by the attacker is used in the program. 3) TaintID which is the unique identifier to identify each vulnerable flow. 4) Taint URL which contains the URL of the web page in which vulnerable flow was revealed.

2.2.4.2.2. Context Recognizer

This component is responsible for the determination of the context of the vulnerable source. It accepts the information provided by the malicious flow decomposer and then uses it to determine the portion of the web page where vulnerable string value is injected. Figure 10 illustrates the algorithm implemented for this step. This algorithm works as follows: Input to the above algorithm is the set of IDs of untrusted source T_ID. Con_ log is a log maintained to store context of each untrusted source. For each tainted source $T_I \in$ T_ID, it attached a context recognizer CR in the form as $C_I \leftarrow (CR)T_I$. The generated output is the internal representation of the extracted JavaScript code embedded with the context recognizer CR corresponds to each untrusted variable present in it. After this, it is merged with the Con_log as $Con\_log \leftarrow C_I \cup Con\_log$. For each $C_I \in$ Con_log, it generates and solves the type constraints. Here, Λ represents the type environment that performs the mapping of the JavaScript variable to the Context recognizer CR. In the path sensitive system, variable's context changes from one point to other point. Thus, to handle this issue, untrusted variables are represented through the typing judgments as $\Lambda \mapsto$ e: CR. It indicates that at any program location, e has context recognizer CR in the type environment Λ. Finally, all C_I variables have been assigned the context dynamically and produce the modified log Con_log as output. This step provides tainted source with their identified context, in which browser interprets it.

2.2.4.2.3. Testing Payload Injector

To exploit the vulnerability present in the webpage, it substitutes attack vector at the place of tainted string. Testing attack vector must be injected according to the context of the tainted string. It is achieved with the help of available repository of XSS attack vector. Then, this module validates the successful execution of the injected attack vector. If attack is successful then, Tainted ID and Taint Source information is forwarded to the filtering module, otherwise, web page is not vulnerable and is returned to the user.

2.2.4.2.4. Filtering

This component accepts the taint Source and tainted ID information as its input. It applies filtering on the tainted string present at the taint Source in the web page, with the help of Filtering APIs. This is done to halt the execution of injected malicious string and triggers malicious effects. Figure 11 shows the algorithm processed for the completion of the filtering process. The working procedure of this algorithm is explained below: Algorithm takes Con_log as its input which stores identified context of all untrusted JavaScript variable. FAPI_lib is the externally available library which stores filtering APIs corresponding to each malicious context. T_ID is the list comprising the IDs of each tainted source. For each Untrusted Source (i.e tainted value) $T_I \in$ T_ID, it extracts the context of T_I as X_I from the Con_log. Then, it identifies for the corresponding Filtering API, from the FAPI_lib, with matching context and stores it in F_I. It applies the identified filtering API on the T_I and store result into the Y_I. It then merges Y_I with FAPI_lib as FAPI_*lib* $\leftarrow Y_I \cup$ FAPI_lib; finally, it embeds all sanitized variable into the HTTP response and produce HTTP response for the user.

3. IMPLEMENATION AND PERFORMANCE EVALUATION

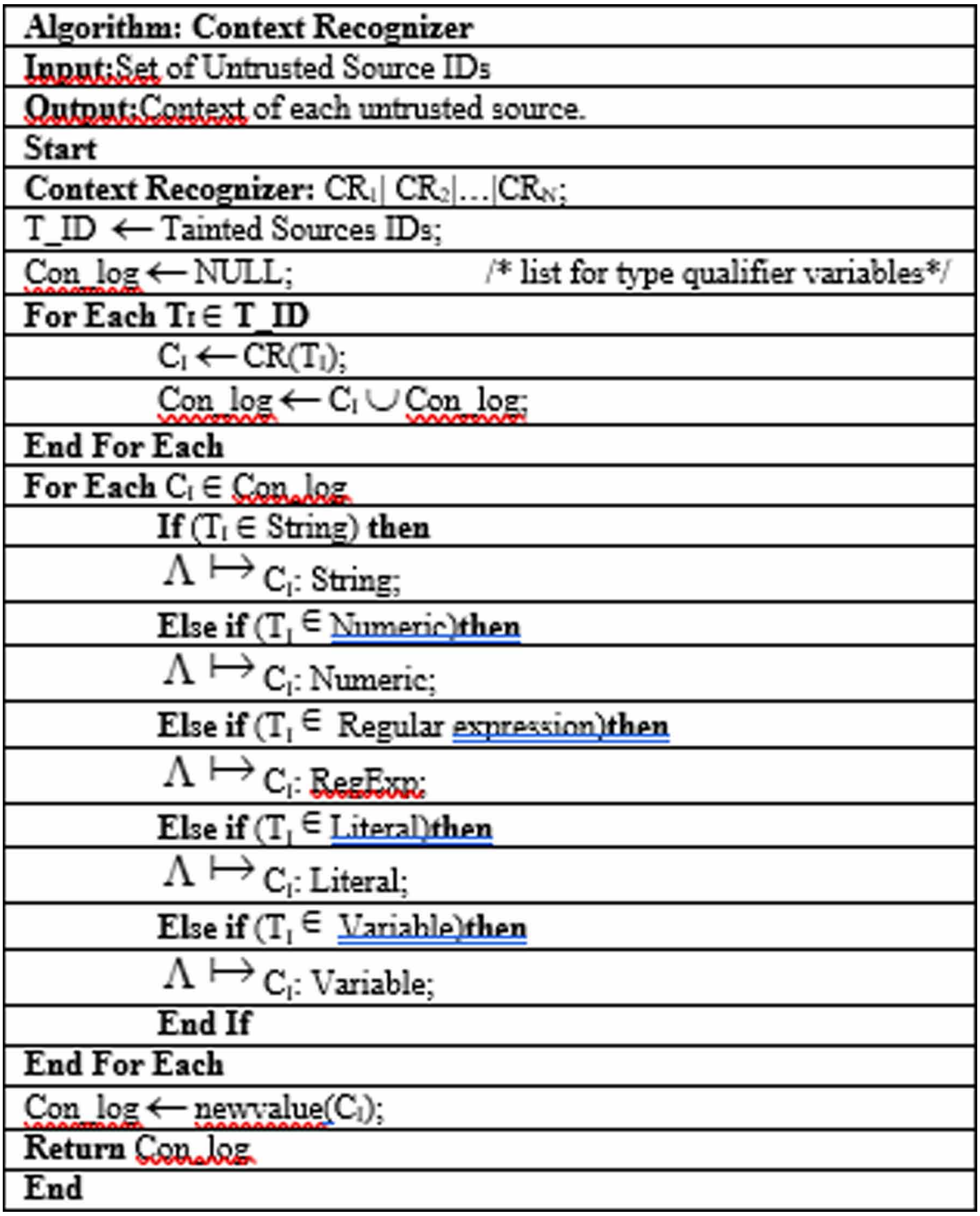
Authors have implemented their work in java, for mitigating the effect of XSS vulnerabilities from the tested suite of real-world web applications. Initially, Authors have manually verified the performance of the framework against available XSS attack repositories (HTML5 Security Cheat Sheet, RSnake, 2008, Technical Attack Sheet for Cross Site Penetration Tests), which includes the list of old and new XSS attack vectors. Very few XSS attack vectors were able to bypass the framework. Authors have utilized HtmlUnit (HtmlUnit parser) to inject the initial trusted remark statements. To inject remark, it modifies locations where JavaScript code is present (i.e. inline, event handling, etc.) and store the modified source program information in MySQL database. To extract the features of valid JavaScript code, Authors have used Rhino (Rhino JavaScript parser) JavaScript parser.

3.1. Experimental Evaluation

Authors have categorized the XSS attack vectors into four main categories i.e. Character Encoding Scripts (CES), Embedded Character Tags (ECT), Event Handlers (EH) and HTML Quote Encapsulation (HQE). Table 5 illustrates the attack patterns of these categories of XSS attack vectors. The observed results of our framework on five real world HTML5 web applications corresponding to chosen categories of XSS worms has been shown in the Figure 12-16.

Authors have also calculated the XSS detection rate of our framework by dividing the number of attacks detected (i.e. # of True Positives) to the number of XSS attack vectors injected on each individual HTML5 web application. Table 6 highlights the detection rate of all five web applications w.r.t. individual category of XSS worms. It is clearly reflected from the Table 6 that OsCommerce and BlogIt observed overall higher percentage detection rate in all the four categories of XSS worms.

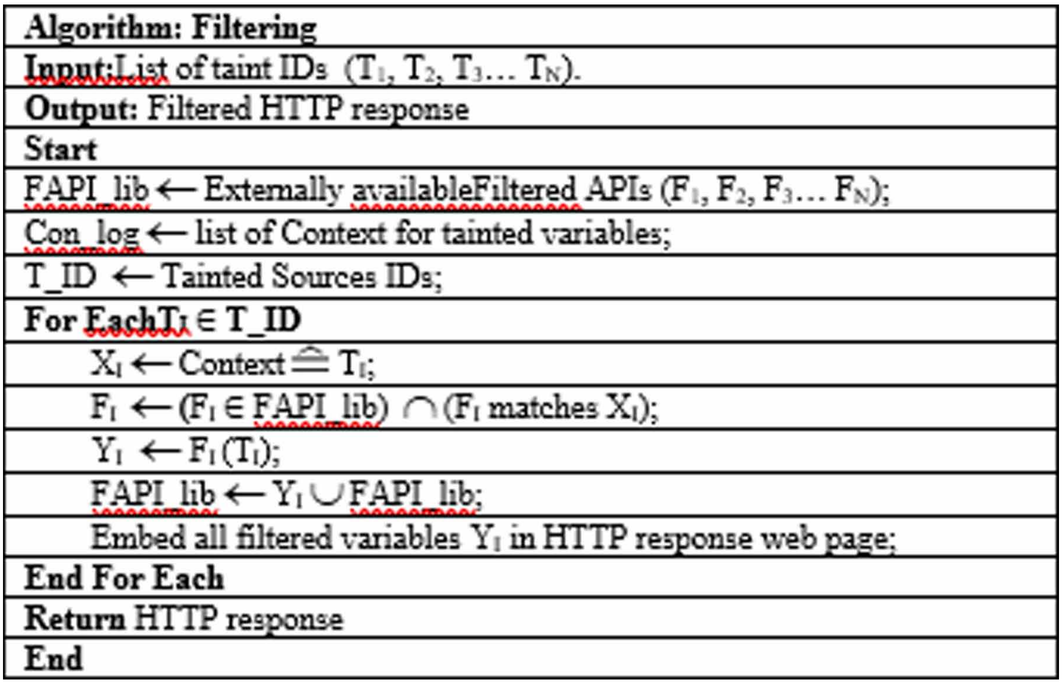
Figure 10. Algorithm implemented for the context recognizer



```
Algorithm: Context Recognizer
Input:Set of Untrusted Source IDs
Output:Context of each untrusted source.
Start
Context Recognizer: $CR_1 | CR_2 | \ldots | CR_N$;
T_ID ← Tainted Sources IDs;
Con_log ← NULL;                    /* list for type qualifier variables*/
For Each $T_I \in$ T_ID
            $C_I \leftarrow CR(T_I)$;
            Con_log ← $C_I \cup$ Con_log;
End For Each
For Each $C_I \in$ Con_log
            If ($T_I \in$ String) then
             $\Lambda \mapsto C_I$: String;
            Else if ($T_I \in$ Numeric)then
             $\Lambda \mapsto C_I$: Numeric;
            Else if ($T_I \in$ Regular expression)then
             $\Lambda \mapsto C_I$: RegExp;
            Else if ($T_I \in$ Literal)then
             $\Lambda \mapsto C_I$: Literal;
            Else if ($T_I \in$ Variable)then
             $\Lambda \mapsto C_I$: Variable;
            End If
End For Each
Con_log ← newvalue($C_I$);
Return Con_log
End
```

In the next two sub-sections, authors have evaluated the performance analysis of the framework by applying two statistical analysis methods: F-Score and F-test.

Figure 11. Algorithm for filtering process



```
Algorithm: Filtering
Input:List of taint IDs ($T_1$, $T_2$, $T_3$… $T_N$).
Output: Filtered HTTP response
Start
FAPI_lib ← Externally availableFiltered APIs ($F_1$, $F_2$, $F_3$… $F_N$);
Con_log ← list of Context for tainted variables;
T_ID ← Tainted Sources IDs;
For Each$T_I \in$ T_ID
        $X_I \leftarrow$ Context $\triangleq T_I$;
        $F_I \leftarrow (F_I \in$ FAPI_lib$) \cap (F_I$ matches $X_I)$;
        $Y_I \leftarrow F_I (T_I)$;
        FAPI_lib ← $Y_I \cup$ FAPI_lib;
        Embed all filtered variables $Y_I$ in HTTP response web page;
End For Each
Return HTTP response
End
```

Figure 12. Observed results on Simplecms

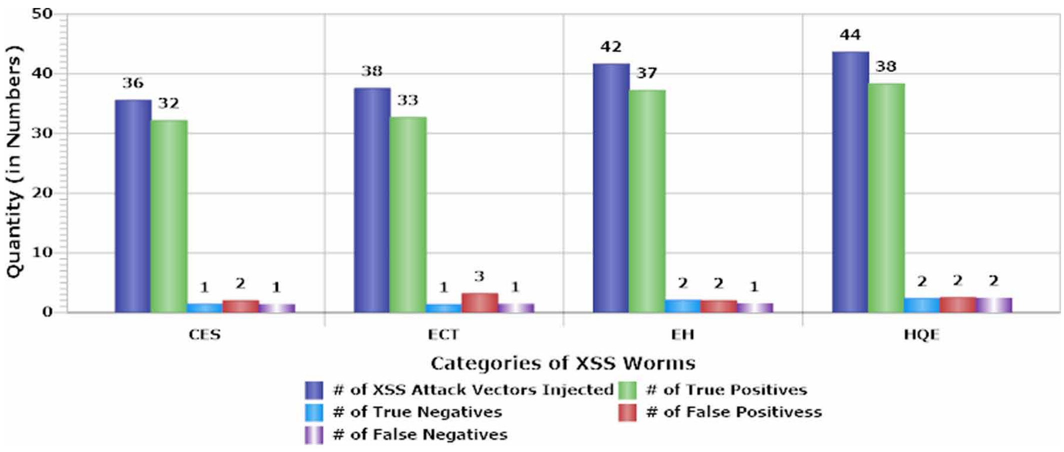


Table 5. Categories of XSS worms with their pattern examples

XSS Worm Category	Explanation	Script Example
CES	The techniques of character encoding are utilized for exemplifying a database of typescripts through certain encoding system. Such techniques are utilized for calculation, data storing, and broadcast of documented info and could be utilized for exploiting numerous attacks.	%253cscript%253ealert(document.cookie)%253c/script%253e
ECT	This category of attack is generally embedded inside the normal syntax of JavaScript code.	<IMG SRC=”jav&#x09;ascript:alert(‘XSS’);”>
EH	These are non-compulsory system commands in the form of scripts, which only execute when an alteration is observed in the service state.	<input onBlur =”alert(‘xss’)” type=”text” >
HQE	This category is generally used for exploiting the XSS attack on web applications that permit “<script>” tag but not “<SCRIPT SRC...”	<script src=”data:text/javascript,alert(1)”></script>

Figure 13. Observed Results on OsCommerce

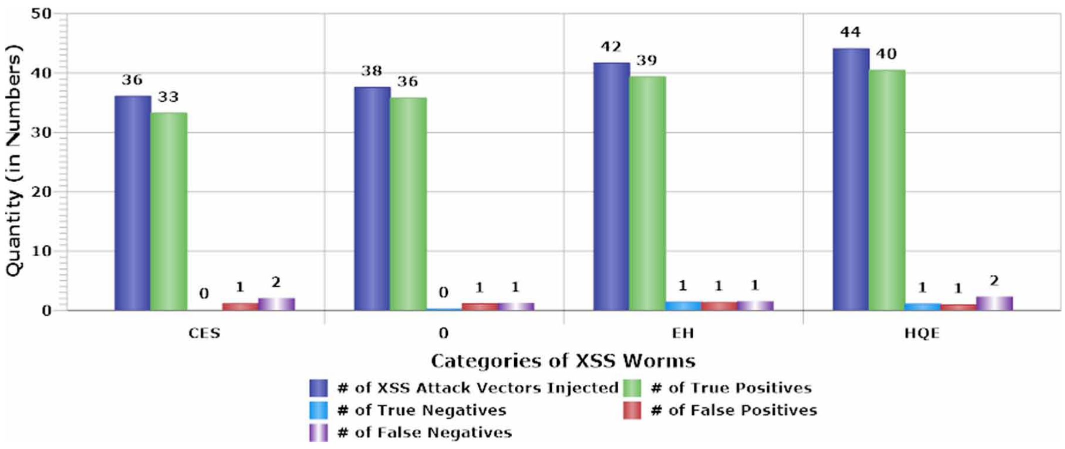


Figure 14. Observed Results on WebCalender

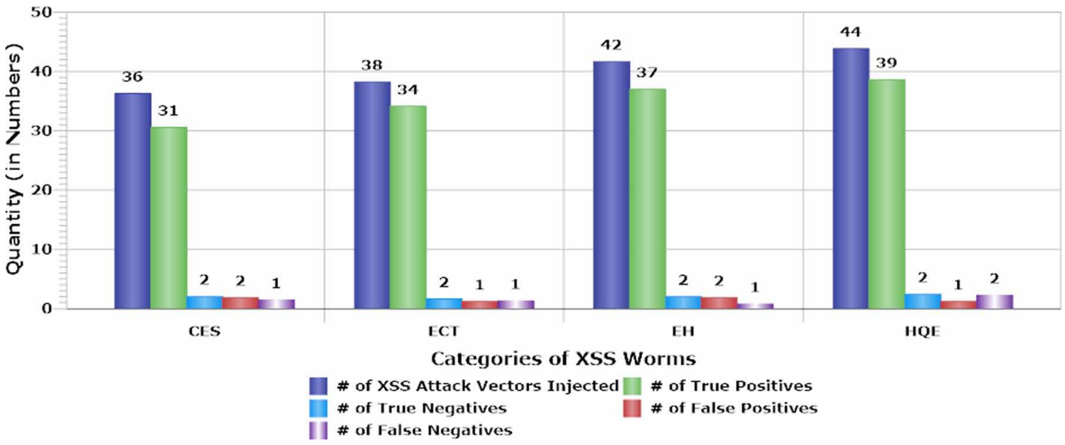


Figure 15. Observed Results on PunBB

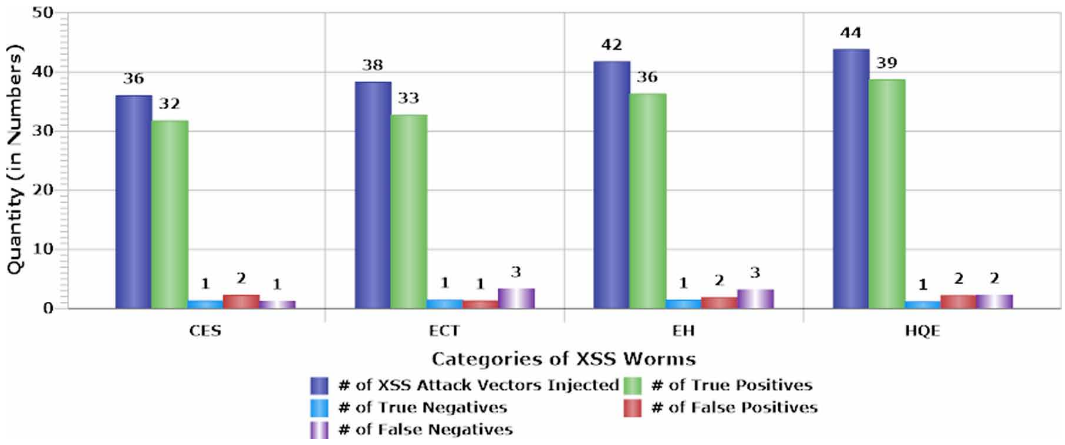


Figure 16. Observed Results on BlogIt

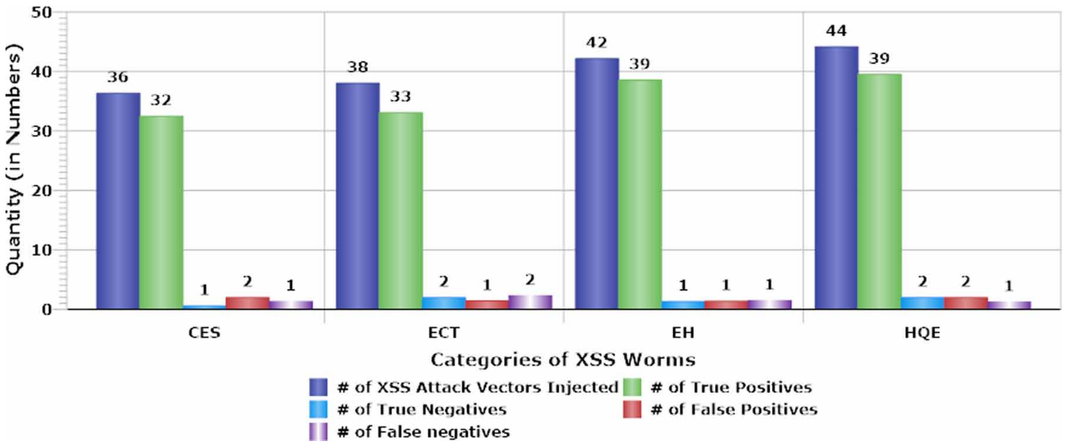


Table 6. Detection rate (in %age) of HTML5 web applications

Web Applications	CES	ECT	EH	HQE
Simplecms	88.8	86.8	88.0	86.3
OsCommerce	91.6	94.7	92.8	91.0
WebCalender	86.1	89.4	88.0	88.6
PunBB	88.8	86.8	85.7	88.6
BlogIt	88.8	86.8	92.8	88.6

3.2. Performance Analysis using F-Score

Authors have presented detailed performance analysis of our framework by conducting a statistical analysis method (i.e. F-Score). The analysis conducted reveals that our framework exhibits high performance as the observed value of F-Scores in all the platforms of HTML5 web applications is 0.9. Therefore, the proposed framework exhibits 90% success rate in all the five HTML5 web applications. Table 7 highlights the detailed performance analysis of our work.

$$\text{Precision} = \frac{True\,Positives\,(TP)}{True\,Positives\,(TP) + False\,Positives\,(FP)}$$

$$\text{Recall} = \frac{True\,Positives\,(TP)}{True\,Positives\,(TP) + False\,Negatives\,(FN)}$$

$$\text{False Positive Rate (FPR)} = \frac{False\,Positves\,(FP)}{False\,Positives\,(FP) + True\,Negatives\,(TN)}$$

$$\text{False Negative Rate (FNR)} = \frac{False\,Negatives\,(FN)}{False\,Negatives\,(FN) + True\,Positives\,(TP)}$$

$$\text{F-Score} = \frac{2\,(True\,Positives)}{2\,(True\,Positives) + False\,Negatives + False\,Positives}$$

Table 7. Performance analysis by calculating F-Score

Web Application	Total	# of TP	# of TN	# of FP	# of FN	Precision	FPR	FNR	Recall	F-Measure
SimpleCms	160	140	6	9	5	0.939	0.6	0.034	0.965	0.952
OsCommerce	160	148	2	4	6	0.973	0.67	0.038	0.961	0.927
Web Calender	160	141	8	6	5	0.959	0.428	0.034	0.965	0.962
PunBB	160	140	4	7	9	0.952	0.478	0.087	0.939	0.945
BlogIt	160	143	6	6	5	0.959	0.5	0.034	0.966	0.913

3.3. Performance Analysis Using F-Test Hypothesis

In order to prove that the number of XSS worms detected is less than to the number of XSS attack vectors injected, we use the F-test hypothesis, which is defined as:

Null Hypothesis (H0) = Number of XSS worms detected is equal to the number of XSS attack vectors injected. ($S1^2 = S2^2$)
Alternate Hypothesis (H1) = Number of XSS worms injected is greater than number of XSS attack vectors detected ($S1^2 < S2^2$)

Table 8. Statistics of XSS Attack Vectors Applied

# of Malicious Scripts Injected (X_i)	**(X_i - μ)**	**(X_i - μ)^2**	**Standard Deviation S_1 =** $\sqrt{\sum_{i=1}^{N1}(Xi-\mu)^2 / (N1-1)}$
158	2	4	3.162
160	4	16	
156	0	0	
154	-2	4	
152	-4	16	
Mean (μ) = $\sum Xi / N1$ = 156		$\sum_{i=1}^{N1}(Xi-\mu)^2 = 40$	

Table 9. Statistics of XSS attack vectors detected

# of JS Attack Payload Detected (X_j)	**(X_i - μ)**	**(X_i - μ)^2**	**Standard Deviation S_2 =** $\sqrt{\sum_{i=1}^{N2}(Xi-\mu)^2 / (N2-1)}$
150	1	1	3.316
154	5	25	
148	-1	1	
148	-1	1	
145	-4	16	
Mean (μ) = $\sum Xi / N2$ = 149		$\sum_{i=1}^{N1}(Xi-\mu)^2 = 44$	

The level of Significance is ($\alpha = 0.05$). The detailed analyses of statistics of XSS attack worms applied and detected are illustrated in the Tables 8 and 9. Analysis using F-test is showing below:

Number of XSS Worms Injected (# of Observation (N_1)) = 5
Degree of Freedom (df_1) = N_1 -1 = 4.
Number of XSS Worms Detected (# of Observation (N_2)) = 5
Degree of Freedom (df_2) = N_2 -1 = 4.

We can calculate F-Test as: $F_{CALC} = S_1^2 / S_2^2 = [(3.162)^2/ (3.316)^2] = 0.9092$
The tabulated value of F-Test at $df_1 = 4$, $df_2 = 4$ and $\alpha = 0.05$ is

$$F_{(df1,df2,1-\alpha)} = F_{(4,\,4,\,0.95)} = 6.3882$$

It is already known that the Null hypothesis is correct if the two variances are equal and condition $F_{CALC} < F_{(df1,df2,1-\alpha)}$ is false, however, in this case, since $F_{CALC} < F_{(4,\,4,\,0.95)}$ is true, therefore, the alternate hypothesis (H1) is true that means the number of XSS worms detected is less than number of XSS attack vectors injected and any difference in the sample standard deviation is due to random error.

3.4 Limitations of our Work

This section discussed about some of the limitations of our work.

- Vulnerable to Click Jacking and Phishing Attacks: Initially, the attack vector could utilize the capabilities of Click Jacking or Phishing attacks for stealing sensitive credentials of user. (e.g. user-id, password, etc.). This is beyond the scope of this current work. Secondly, it can also be argued that as the attack vectors are executing under the license of web site, this is considered an easy way for the attack vector to exploit such attacks.
- Vulnerable to Drive-by Download Worms: Finally, this framework is also exposed to worms like Koobface, which transmit the binaries to the web browser of victim by utilizing drive-by exploits. This category of attack is also beyond the scope of this work.

Table 10. Comparison of existing work with our XSS defensive work

Techniques Parameters	XSSFilt	XSSAudior	BIXSAN	ScriptGard	Livshits	Jsand	XSS-Guard	(Present work)
AM	Active	Active	Passive	Passive	Passive	Passive	Passive	Active
MP	Dynamic	Static	Static	Static	Dynamic	Static	Static	Dynamic
TOXH	Reflected	Reflected	Stored	Reflected	Reflected	Stored	Reflected	Reflected, Stored
Ttrac	✗	✗	✗	✗	✓	✓	✓	✓
CRW	✗	✗	✓	✗	✓	✗	✓	✓
APPR	✓	✓	✓	✓	✓	✗	✗	✗
PSID	✓	✗	✗	✓	✗	✗	✗	✓
SCM	✓	✓	✓	✓	✓	✓	✗	✓
SCMod	✓	✓	✓	✓	✓	✓	✗	✗

3.5 Comparative and Strengths of the Proposed Work

This section discusses the comparison of our framework with the other recent existing XSS defensive methodologies. Table 10 compares the existing XSS defensive methodologies with our work based on nine useful performance parameters: AM: Analyzing Mechanism, MP: Monitoring Procedure, TOXH: Type of XSS Worm Handled, Ttrac: Taint Tracking, CRW: Code Rewriting, APPR: Automated Pre-Processing Required, PSID: Partial Script Injection Detection, SCM: Source Code Monitoring and SCMod: Source Code Modifications.

In addition, recognition of malicious script methods is simply evaded by most of the techniques. Moreover, lot of pre-processing is required in the existing framework of web applications for their successful execution on different platforms of web browsers as well as web applications. Most of the existing work relies on the concept of exact JavaScript injection; however, they could not able to detect the partial injection of XSS worms.

This framework simply isolates the untrusted JavaScript code from the actual data by executing the process of code rewriting, that is not handled by most of the existing XSS defensive techniques. In addition, the proposed framework executes the runtime monitoring on the JavaScript code for determining the dependency between the tainted source and sink functions in the program code. Moreover, context of untrusted variables embedded in such code is determined. Now, here, instead of performing context-sensitive sanitization on such variables, our framework performs the deep string analysis on such variables for tracking their tainted flow. The examination of tainted variables will be carried out in order to determine whether it may function as vulnerable point or not. The existing work performs the context-sensitive sanitization on such variables.

4. CONCLUSION

Digitalization in every aspect of life demands more technological advancements for providing information anytime anywhere. This materializes the vision of making cities “smarter”. Undoubtedly, this development induces multiple benefits to the people; nevertheless, it brings to light multiple threats like XSS. Therefore, this paper described a technique to protect users from XSS attack. This is achieved through classifying response web page into static and dynamic web pages. Static web pages are processed by injecting and verifying trusted remark statements to detect persistent XSS attack. Moreover, it accomplishes vulnerable flow analysis followed by filtering of tainted string, for determining XSS attack in dynamic web pages. Authors have implemented the framework using java development framework and has evaluated its detection capability on five real-world web applications. The results revealed low false negative rate with tolerable performance overhead due to injection and removal of remarks.

ACKNOWLEDGMENT

This publication is an outcome of the R&D work undertaken under the projects YFRF, Visvesvaraya PhD Scheme of Ministry of Electronics & Information Technology, Government of India and being implemented by Digital India Corporation.

REFERENCES

Agten, P., Van Acker, S., Brondsema, Y., Phung, P. H., Desmet, L., & Piessens, F. (2012, December). JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference* (pp. 1-10). ACM. doi:10.1145/2420950.2420952

Almomani, A. et al.. (2013). Phishing dynamic evolving neural fuzzy framework for online detection zero-day phishing email. *Indian Journal of Science and Technology*, *6*(1), 3960–3964.

Bates, D., Barth, A., & Jackson, C. (2010, April). Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web* (pp. 91-100). ACM. doi:10.1145/1772690.1772701

Bisht, P., & Venkatakrishnan, V. N. (2008, July). XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 23-43). Springer. doi:10.1007/978-3-540-70542-0_2

Chandra, V. S., & Selvakumar, S. (2011). BIXSAN: Browser Independent XSS Sanitizer for prevention of XSS attacks. *Software Engineering Notes*, *36*(5), 1–7. doi:10.1145/1968587.1968603

Chaudhary, P., Gupta, B. B., & Gupta, S. (2018). Defending the OSN-based web applications from XSS attacks using dynamic javascript code and content isolation. In *Quality, IT and Business Operations* (pp. 107–119). Singapore: Springer. doi:10.1007/978-981-10-5577-5_9

Chaudhary, P., Gupta, B. B., & Gupta, S. (2019). A Framework for Preserving the Privacy of Online Users Against XSS Worms on Online Social Network. *International Journal of Information Technology and Web Engineering*, *14*(1), 85–111. doi:10.4018/IJITWE.2019010105

Chaudhary, P., Gupta, S., & Gupta, B. B. (2016). Auditing Defense against XSS Worms in Online Social Network-Based Web Applications. In Handbook of Research on Modern Cryptographic Solutions for Computer and Cyber Security (pp. 216-245). Hershey, PA: IGI Global. doi:10.4018/978-1-5225-0105-3.ch010

Drennan, J., Sullivan, G., & Previte, J. (2006). Privacy, risk perception, and expert online behavior: An exploratory study of household end users. *Journal of Organizational and End User Computing*, *18*(1), 1–22. doi:10.4018/joeuc.2006010101

Ferraz, F. S., & Ferraz, C. A. G. (2014, December). Smart city security issues: depicting information security issues in the role of an urban environment. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing* (pp. 842-847). IEEE. doi:10.1109/UCC.2014.137

Gupta, B. B., Gupta, S., & Chaudhary, P. (2017a). Enhancing the browser-side context-aware sanitization of suspicious HTML5 code for halting the DOM-based XSS vulnerabilities in cloud. [IJCAC]. *International Journal of Cloud Applications and Computing*, *7*(1), 1–31. doi:10.4018/IJCAC.2017010101

Gupta, S., & Gupta, B. B. (2016a). An infrastructure-based framework for the alleviation of JavaScript worms from OSN in mobile cloud platforms. In *Proceedings of the International conference on network and system security* (pp. 98-109). Cham: Springer. doi:10.1007/978-3-319-46298-1_7

Gupta, S., & Gupta, B. B. (2016b). Enhanced XSS defensive framework for web applications deployed in the virtual machines of cloud computing environment. *Procedia Technology*, *24*, 1595–1602. doi:10.1016/j.protcy.2016.05.152

Gupta, S., & Gupta, B. B. (2017b). Smart XSS attack surveillance system for OSN in virtualized intelligence network of nodes of fog computing. *International Journal of Web Services Research*, *14*(4), 1–32. doi:10.4018/IJWSR.2017100101

Gupta, S., & Gupta, B. B. (2018b). Robust injection point-based framework for modern applications against XSS vulnerabilities in online social networks. *International Journal of Information and Computer Security*, *10*(2-3), 170–200. doi:10.1504/IJICS.2018.091455

Gupta, S., & Gupta, B. B. (2018c). RAJIVE: Restricting the abuse of JavaScript injection vulnerabilities on cloud data centre by sensing the violation in expected workflow of web applications. *International Journal of Innovative Computing and Applications*, *9*(1), 13–36. doi:10.1504/IJICA.2018.090822

Gupta, S., & Gupta, B. B. (2018d). Evaluation and monitoring of XSS defensive solutions: A survey, open research issues and future directions. *Journal of Ambient Intelligence and Humanized Computing*, 1–29.

Gupta, S., & Gupta, B. B. (2018e). POND: Polishing the execution of nested context-familiar runtime dynamic parsing and sanitisation of XSS worms on online edge servers of fog computing. *International Journal of Innovative Computing and Applications*, *9*(2), 116–129. doi:10.1504/IJICA.2018.092588

Gupta, S., & Gupta, B. B. (2018f). A robust server-side javascript feature injection-based design for JSP web applications against XSS vulnerabilities. In *Cyber Security: Proceedings of CSI 2015* (pp. 459-465). Springer Singapore. doi:10.1007/978-981-10-8536-9_43

Gupta, S., Gupta, B. B., & Chaudhary, P. (2018a). Hunting for DOM-Based XSS vulnerabilities in mobile cloud-based online social network. *Future Generation Computer Systems*, *79*, 319–336. doi:10.1016/j.future.2017.05.038

Hossain, M. S., Muhammad, G., Abdul, W., Song, B., & Gupta, B. B. (2018). Cloud-assisted secure video transmission and sharing framework for smart cities. *Future Generation Computer Systems*, *83*, 596–606. doi:10.1016/j.future.2017.03.029

HTML5 Security Cheat Sheet. (n.d.). Retrieved from https://html5sec.org/

HtmlUnit parser. (n.d.). Retrieved from https://sourceforge.net/projects/htmlunit/files/htmlunit/

Li, D., Deng, L., Bhooshan Gupta, B., Wang, H., & Choi, C. (2019). A novel CNN based security guaranteed image watermarking generation scenario for smart city applications. *Information Sciences*, *479*, 432–447. doi:10.1016/j.ins.2018.02.060

Li, J., Yu, C., Gupta, B. B., & Ren, X. (2018). Color image watermarking scheme based on quaternion Hadamard transform and Schur decomposition. *Multimedia Tools and Applications*, *77*(4), 4545–4561. doi:10.1007/s11042-017-4452-0

Livshits, B., & Chong, S. (2013, January). Towards fully automatic placement of security sanitizers and declassifiers. *ACM SIGPLAN Notices*, *48*(1), 385–398. doi:10.1145/2480359.2429115

Parada, R., Melià-Seguí, J., & Pous, R. (2018). Anomaly Detection Using RFID-Based Information Management in an IoT Context. *Journal of Organizational and End User Computing*, *30*(3), 1–23. doi:10.4018/JOEUC.2018070101

Pelizzi, R., & Sekar, R. (2012). Protection, usability and improvements in reflected XSS filters. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (p. 5). ACM. doi:10.1145/2414456.2414458

Rhino JavaScript parser. (n.d.). Retrieved from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/Download_Rhino

XSS RSnake. (2008). Cheat Sheet. Retrieved from https://n0p.net/penguicon/php_app_sec/mirror/xss.html

Saxena, P., Molnar, D., & Livshits, B. (2011, October). SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (pp. 601-614). ACM. doi:10.1145/2046707.2046776

Sen, M., Dutt, A., Agarwal, S., & Nath, A. (2013, April). Issues of privacy and security in the role of software in smart cities. In *Proceedings of the 2013 International Conference on Communication Systems and Network Technologies* (pp. 518-523). IEEE. doi:10.1109/CSNT.2013.113

Technical Attack Sheet for Cross Site Penetration Tests. (n.d.). Retrieved from http://www.vulnerability-lab.com/resources/documents/531.txt

B. B. Gupta received PhD degree from Indian Institute of Technology Roorkee, India in the area of information security. He has published more than 250 research papers in international journals and conferences of high repute. He has visited several countries to present his research work. His biography has published in the Marquis Who's Who in the World, 2012. At present, he is working as an Assistant Professor in the Department of Computer Engineering, National Institute of Technology Kurukshetra, India. His research interest includes information security, cyber security, cloud computing, web security, intrusion detection, computer networks and phishing.

Pooja Chaudhary is currently pursuing her PhD degree in Information and Cyber security from National Institute of Technology, Kurukshetra, Haryana, India. She has completed her M.Tech in Computer Engineering from National Institute of Technology, Kurukshetra. She has received her B.Tech degree in Computer Science and Engineering from Bharat Institute of Technology, Meerut, Affiliated to Uttar Pradesh Technical University, India. Her areas of interest include online social network security, Big Data analysis and security, database security, and cyber security.

Shashank Gupta is currently working as an Assistant Professor in Computer Science and Information Systems Division at Birla Institute of Technology and Science, Pilani, Rajasthan, India. He has done his PhD under the supervision of Dr. B. B. Gupta in Department of Computer Engineering specialization in Web Security at National Institute of Technology Kurukshetra, Haryana, India. Recently, he was working as an Assistant Professor in the Department of Computer Science and Engineering at Jaypee Institute of Information Technology (JIIT), Noida, Sec-128. Prior to this, he has also served his duties as an Assistant Professor in the Department of IT at Model Institute of Engineering and Technology (MIET), Jammu. He has completed M.Tech. in the Department of Computer Science and Engineering Specialization in Information Security from Central University of Rajasthan, Ajmer, India. He has also done his graduation in Bachelor of Engineering (B.E.) in Department of Information Technology from Padmashree Dr. D.Y. Patil Institute of Engineering and Technology Affiliated to Pune University, India. He has also spent two months in the Department of Computer Science and IT, University of Jammu for completing a portion of Post-graduation thesis work. He bagged the 1st Cash Prize in Poster Presentation at National Level in the category of ICT Applications in Techspardha'2015 and 2016 event organized by National Institute of Kurukshetra, Haryana. He has numerous online publications in International Journals and Conferences including IEEE, Elsevier, ACM, Springer, Wiley, Elsevier, IGI-Global, etc., along with several book chapters. He is also serving as reviewer for numerous peer-reviewed Journals and conferences of high repute. He is also a professional member of IEEE and ACM. His research area of interest includes web security, cross-site scripting (XSS) attacks, online social network security, cloud security, fog computing and theory of computation.