Research Article

Automatic Test Pattern Generator for Fuzzing Based on Finite State Machine

Ming-Hung Wang, Han-Chi Wang, You-Ru Chen, and Chin-Laung Lei

Department of Electrical Engineering, National Taiwan University, Taipei City, Taiwan

Correspondence should be addressed to Chin-Laung Lei; cllei@ntu.edu.tw

Received 5 April 2017; Revised 14 August 2017; Accepted 10 October 2017; Published 13 November 2017

Academic Editor: Emanuele Maiorana

Copyright © 2017 Ming-Hung Wang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

With the rapid development of the Internet, several emerging technologies are adopted to construct fancy, interactive, and user-friendly websites. Among these technologies, HTML5 is a popular one and is widely used in establishing modern sites. However, the security issues in the new web technologies are also raised and are worthy of investigation. For vulnerability investigation, many previous studies used fuzzing and focused on generation-based approaches to produce test cases for fuzzing; however, these methods require a significant amount of knowledge and mental efforts to develop test patterns for generating test cases. To decrease the entry barrier of conducting fuzzing, in this study, we propose a test pattern generation algorithm based on the concept of finite state machines. We apply graph analysis techniques to extract paths from finite state machines and use these paths to construct test patterns automatically. According to the proposal, fuzzing can be completed through inputting a regular expression corresponding to the test target. To evaluate the performance of our proposal, we conduct an experiment in identifying vulnerabilities of the input attributes in HTML5. According to the results, our approach is not only efficient but also effective for identifying weak validators in HTML5.

1. Introduction

Nowadays, with the rapid development of modern web technologies, a large number of social, interactive, and commercial services and a great amount of information are provided and exchanged between websites and end users. Meanwhile, more and more sensitive data such as payment or personal information has been exchanged and stored online. For example, HTML5 is an emerging interactive syntax which has been adopted in website development for its up-to-date specifications and support of multimedia content. Compared with HTML4, HTML5 defines more specific input attributes such as telephone number, color, and email address. However, on the other hand, some studies have also presented the security issues affiliated with HTML5 [1–5], especially the injection attacks on the websites [6–9]. Even though in HTML5 regular users can only enter a valid value through the user interfaces supported by browsers, malicious users can skip the user interface and directly send malformed HTTP requests to the web server. These requests could contain

strings for injection attacks. As the requests are directly sent to the web server and are not checked by the user interfaces, websites should handle the malicious string inputs to avoid injection attacks.

Current website developers rely on developing several regular expressions to validate the correctness of the input strings and filter malicious ones. However, while the validators are designed and constructed by mere brainstorming of engineers without a formal and systematic methodology, there may be some underneath threats based on the loopholes in these expressions due to the human limitations [10]. To address the issue, some studies adopt fuzzing [11, 12], a software testing technique that has been used in quality assurance and vulnerability detection [13, 14]. For its programmability and flexibility in conducting regularized and large-scale tests, fuzzing helps to reduce human efforts and provides quick test case generation [15–17].

In this paper, we design an automatic fuzzing framework that can be used to investigate potential injection vulnerabilities for modern HTML5 websites. We develop a systematic

and automatic testing framework that is capable and efficient for engineers to identify vulnerabilities early. To generate test inputs extensively, we leverage the concept of finite state machines (FSM) and graphical analysis algorithms. Our fuzzing procedure starts by analyzing the regular expressions of the corresponding input attributes (e.g., a date format or a telephone number format in HTML5). We next transform the regular expression to a corresponding FSM and take the negation of the FSM, so that we can derive a FSM for generating incorrect input strings to imitate abnormal behaviors or malfunctions of websites. From the negated FSM, we treat it as graphs and perform path extraction according to different algorithms to create invalid input cases.

By incorporating the above algorithms with automation components in our fuzzing framework, we present three major advantages of our system. First, we enhance the accessibility of our system by accepting regular expressions to produce invalid test patterns for fuzzing. Second, we provide the automation in test case generation and injection toward the target server. Third, our framework aggregates and filters potential abnormal responses while fuzzing using a similarity comparison mechanism of HTTP responses. According to the above features, our system offloads the efforts for reviewing the result of testing manually, which is beneficial while conducting large-scale tests.

Our contributions in this study are threefold:

- (i) We design a framework adopting fuzzing to investigate potential vulnerabilities in websites containing HTML5 and automatically aggregate and summarize results into reports for manual review.
- (ii) We present a FSM-based pattern generation algorithm to regularly produce invalid input test cases for fuzzing, according to the grammar of input attributes. From the method, labor efforts and time spent on brainstorming invalid input strings can be almost waived when conducting large-scale generation-based fuzzing.
- (iii) We implement our design and evaluate the system by in-lab experiments and on a popular website. From the results, our system achieves high detection rates on misdesigned websites and completes website testing within several minutes.

The remainder of this paper is organized as follows. We discuss the related studies of fuzzing and the strength of our study in Section 2. In Section 3, we present our framework for fuzzing HTML5 websites. In Section 4, we propose test pattern generation method based on FSM and graphical analysis. We demonstrate evaluation results and discuss the finding of our study in Section 5. In Section 6, we present the results on fuzzing a popular website which is in operation. The conclusion and future issues of this study are depicted in Section 7.

2. Related Works

2.1. Fuzzing. Fuzzing was first proposed by Miller et al. [11, 18] as a software testing technique. The primary procedure of

fuzzing is to develop programs to generate a series of test strings automatically and inject them into the target. By observing behaviors and responses of the target, engineers could investigate the abnormal behaviors or unexpected responses to identify potential vulnerabilities of the test target. As fuzzing usually relies on generating a large number of test cases, several fuzzing studies presented methods for creating test cases, which can be categorized into grammar-based fuzzing [12, 15, 16, 19, 20] and mutation-based fuzzing [19, 21].

In this study, we focus on grammar-based fuzzing, which is more suitable for generating strings according to the pre-defined rules in HTML5. From the past studies on grammar-based fuzzing, test input generation can be divided into two mechanisms: randomized generating [22–24] and exhaustive generating [25]. However, both algorithms require specific knowledge and investigation on the test target. Also, as the injection attacks are usually conducted for certain purposes, the injection strings are mostly in certain forms rather than in randomized formats. Thus, testing engineers often try to provide grammar rules based on brainstorming and self-innovation to concentrate on particular vulnerabilities. However, the procedure usually consumes an increasing amount of time and effort to achieve enough quality and quantity of cases. To address such labor-intensive problem with generating test patterns, past scholars worked on various creating methods [26, 27]. However, the proposals mentioned above are usually designed for a certain product or are not fully applicable to web injection issues so far.

2.2. FSM in Testing. FSM-based methodologies for software testing have been popularly utilized in hardware testing and protocol testing [28, 29]. The main idea of previous methods is to combine the state diagram and directed edges to build a graph, where the vertices denote machine states and the edges correspond to the transitions between machine states [30–32]. To traverse the graph, a number of algorithms have been presented by scholars [33, 34], including testing methods based on the shortest path, the states coverage, and the transition coverage. On the other hand, a number of scholars have also incorporated regular expressions for systemized test generation [35, 36].

Motivated by the vast number of studies using FSM, regular expressions, and graph traversal for program testing, we advantage these concepts in generating test sequences not for software verification but for fuzzing strings, where methods based on FSM and structural analysis are not well investigated. We observe that the input validators of websites are usually developed by regular expressions, and creating fuzzing strings systematically and regularly is a challenging problem. In this study, we propose combining regular expressions, FSM, and graph traversal techniques to construct a systematical test pattern generator. The main advantage of our proposal is to use both FSM and the negated FSM to generate both valid and invalid test patterns. In addition, the automated generating mechanism can mimic the misdesigned scenarios of engineers. Thus, our fuzzer can accept formal expressions of input strings and automatically generate fuzzing strings and a series of weak validators. Our

FIGURE 1: The infrastructure of our fuzzing framework.

framework also consists of a series of fuzzing procedures, including producing injection sequences, fuzzing targets, and filtering and identifying potential vulnerabilities. According to our methods, we assert that our proposal could help the state-of-the-art fuzzing to reduce both mental efforts of developers and time consumption of conducting large-scale webpage injection testing.

3. Fuzzing Framework for HTML5

In this section, we present our design of a fuzzing framework for HTML5 websites. We concentrate on the input attributes because they are the most common channels for users to communicate with web servers by using forms and input elements.

To construct a smart fuzzing framework for HTML5, we start by investigating the newly defined input attributes in HTML5. Even though more types of input attributes are supported in HTML5 and well implemented in popular browsers, potential injection could be conducted if the crackers do not follow the corresponding user interface to enter values. A practical attack is via sending HTTP requests.

To detect such weakness of these websites, our framework consists of five major modules: (1) a webpage traversal to visit all available pages of a website, (2) a webpage analyzer to investigate entries that would be channels for injecting websites, (3) a test pattern generator to automatically produce test inputs corresponding to the attribute of each input element, (4) an intelligent injector for conducting fuzzing using HTTP requests and collecting responses from servers, and (5) a result filtering module to aggregate and filter the large-scale results from fuzzing. The infrastructure of our system is shown in Figure 1. We describe each component in detail as follows.

3.1. Website Traversal. Website traversal is essential for scanning potential vulnerabilities systematically among large-scale websites. We design a web traversal tool that accepts the URL of a site and traverse as many available pages of it. Starting from the designated address, the tool first visits the initial page and treats the address as the root. Next, the tool searches entire sublinks in the webpage and checks

the availability of these links. The tool stores the available sublinks in a queue; afterwards, for each link in the queue, the tool visits the link, checks all its sublinks, and stores the available ones. The web traversal continues until all links in the queue have been visited.

However, if we simply use the link-based search algorithm for traversal, the searching process may not converge because not every link is affiliated with the website. To address this issue, we also apply some constraints on the sublinks. Only the sublinks that are in the same and deeper depth as the root are retained in our traversal queue; otherwise, we discard them. After the traversal procedure, our tool returns a dataset containing all available webpage addresses and their content to the webpage analyzing module for the following fuzzing actions.

3.2. Webpage Analyzing and Test Pattern Generation. To thoroughly investigate input elements of the retrieved pages, our webpage analyzing module parses the content, searches the form elements in HTML5 format, and extracts the input columns belonging to the form. Next, we generate corresponding test patterns for conducting fuzzing. We design a test pattern generator based on FSM and graphical analysis to produce test cases corresponding to different input attributes. The detailed algorithm is described in Section 4.

Constructing a series of valid and invalid inputs helps us to conduct fuzzing from two perspectives: functional test (using valid inputs) and vulnerability investigation (using invalid inputs). After the two steps, the generated test cases will be sent to the intelligent injector for conducting injection.

3.3. Intelligent Injection. Most of the modern browsers are now supporting HTML5. These browsers are implemented with user interfaces for predefined input attributes in HTML5. For example, if we want to enter the value in the *date* input column, the screen will show a calendar window for the user to choose. Moreover, the agent may validate if the value entered in the input column is valid to prevent users from entering malformed and malicious values. For instance, the value entered in *number* type should be an integer if its *step* attribute is not set to “any”; otherwise, the form containing the number element could not be submitted. Due to these

Input:

The HTTP response body message from the target server (invalid test cases) $\mathcal{J}$;
 The HTTP response body message from the target server (valid test case) $\mathcal{V}$;
 The designated threshold ρ ;

Output:

Logical value
 0: no/low injection risk; 1: high injection risk

```

(1) if row.count( $\mathcal{J}$ ) == row.count( $\mathcal{V}$ ) then                                ▷ While responses  $\mathcal{J}$  and  $\mathcal{V}$  have same number of rows
(2)   similarity = NULL                                                  ▷ Initiate a similarity vector for storing row-by-row similarity results.
(3)   for  $r$  in 1 to row.count( $\mathcal{J}$ ) do
(4)     similarity = append(similarity, 1 - (diff( $\mathcal{J}_r$ ,  $\mathcal{V}_r$ )))          ▷ Compare the similarity of the two responses line by line.
(5)   similarity.score = mean(similarity)                                ▷ Take the average of the similarity vector
(6) else                                                                ▷ If the length of response is different, concatenate responses as strings for comparison.
(7)    $\mathcal{J}^*$  = NULL
(8)    $\mathcal{V}^*$  = NULL
(9)   for  $r$  in 1 to row.count( $\mathcal{J}$ ) do
(10)     $\mathcal{J}^*$  = string.concat( $\mathcal{J}^*$ ,  $\mathcal{J}_r$ )
(11)   for  $r$  in 1 to row.count( $\mathcal{V}$ ) do
(12)     $\mathcal{V}^*$  = string.concat( $\mathcal{V}^*$ ,  $\mathcal{V}_r$ )
(13)   similarity.score = 1 - (diff( $\mathcal{J}^*$ ,  $\mathcal{V}^*$ ))
(14) return if (similarity.score  $\geq \rho$ );

```

ALGORITHM 1: Response similarity comparison.

spec-specific implementations in the browser, the submitted string should be valid and meet the specifications of HTML5.

However, the injection attacks are usually not via browser user interfaces; instead, many injection attacks are conducted through HTTP requests, such as POST requests. To imitate such malicious behavior and investigate the possible vulnerabilities, we do not use the default user interface provided by browsers to inject test cases. In our system, we rather directly create customized HTTP requests and send them to the target website. The HTTP POST requests from our injection module contain a series of test strings according to the form structures of the page. After the injection, our system monitors the responses from the target server and records them for the result analysis.

3.4. Result Filtering. Because fuzzing may use a lot of test cases to examine large-scale websites, reviewing the reports is a labor-intensive task. To offload the result analysis task, we develop a filtering method to identify potential abnormal responses in the test.

Our idea is based on the similarity of the HTTP responses of the server. To validate whether the web servers can resist injection attacks, we assume that the server should examine the injection input and reject an invalid string. Thus, differences should exist in the HTTP response body messages from the server, for example, error messages or a webpage redirection. Once there are no or only small differences between the replies from a server while injecting valid and invalid test inputs, we assert that the validator of input strings in the website is not dependable enough and may encounter risks from injection attacks.

The algorithm of our filtering module is described as follows. For the response body of the HTTP request

containing *valid* strings, we denote them as a valid response. Next, we collect the HTTP responses from fuzzing requests containing *invalid* strings. Our result filtering module performs a similarity measurement among the responses using Levenshtein distance [37]. Once the similarity between the HTTP response body messages of valid and invalid inputs is higher than a predefined threshold, we filter them out for further manual analysis, because the websites may not handle such invalid strings. The details of our approach are described in Algorithm 1 and Table 1. For example, here we take two responses from a server and calculate the response similarity through Algorithm 1. The Levenshtein distance of the two responses, `Your telephone number is 123-456-7890.` and `The number is invalid. Please revise it!`, is 37. While the maximum length of the two responses is 40 characters, the derived similarity score can be calculated as $1 - (37/40) = 0.075$.

The above example responses are pretty different from the other ones. In the practical scenario, choosing a suitable threshold will help to decrease false-positive cases and increase the detection rates. To discuss the issue, we present four false-positive samples to demonstrate the influence of different thresholds. From Table 2, we assume that four responses are all from a well-designed server. We find that the top two responses will be considered as vulnerable if we set a threshold as 0.8, while the bottom two will not. However, from our observation, it is unusual to see a rejection response (as the top two responses) which is so similar to the acceptance response. Normally, the similarity between a rejection and an acceptance response is quite small. Based on the above observation, we set 0.8 as our default threshold for the following experiment.

TABLE 1: Table of functions and notations in Algorithm 1.

Name	Type	Description
row.count	Function	The number of rows in an HTTP response.
diff	Function	The difference ratio of two strings. “distance (string 1, string 2)/max (# of characters of string 1, # of characters of string 2).”
string.concat	Function	Concatenate the strings while the row numbers of the valid responses and injection responses are not the same.
$\mathcal{J}$	Notation	The response of an injection HTTP request containing an <i>invalid</i> test case generated by our system.
$\mathcal{V}$	Notation	The response of an injection HTTP request containing a <i>valid</i> test case generated by our system.
$\mathcal{J}^*$	Notation	The concatenated string from the response $\mathcal{J}$.
$\mathcal{V}^*$	Notation	The concatenated string from the response $\mathcal{V}$.
ρ	Notation	The threshold value for judging if the target website has handled the request containing invalid test cases.

TABLE 2: Different thresholds and their false-positive cases (accepting response: the number is valid).

Threshold ρ	Sample false-positive response	Similarity score
0.9	The number is <i>invalid</i>	0.904
0.8	This number is <i>invalid</i>	0.818
0.7	Your number is <i>invalid</i>	0.727
0.6	The number is <i>incorrect</i>	0.608

4. Test Pattern Generator Based on Finite State Machine

In this section, we describe our proposed FSM-based pattern generator for producing test strings. Our idea is to apply graphical analysis on FSM to generate test patterns for fuzzing automatically. The main advantage of our proposal is to enable pattern generation by simply inputting regular expressions. From the algorithm, we not only mutate the valid regular expressions into misdesigned ones regularly but also use different graph traversal techniques to generate extensive and useful fuzzing patterns.

4.1. Overview. Figure 2 presents steps for generating test patterns using our algorithm. First, we accept a regular expression r as a specified type of input; for example, an expression r can be $/[1-9]\{1\}[0-9]\{3\}/$, representing a number from 1000 to 9999. Next, we transform the regular expression r into a FSM M . At this step, the FSM constructor transforms regular expressions to deterministic finite automata (DFA) according to Brzozowski’s algorithm [38]. To generate invalid test patterns, we take the negation of the original FSM M and derive a negated machine M' . This can be done by two steps: first, we complete the transitions of M to ensure that M would reach finite states on every kind of valid inputs; second, we change all finite states in M to nonfinal states and vice versa, and we derive M' . Figures 3 and 4 present the above-mentioned procedure. We take a simple regular expression $/a@b+ /$ as an example; first, we construct the FSM M of the regular expression $/a@b+ /$ as in Figure 3 according to Brzozowski’s algorithm [38]; next, we take the negation of the FSM M in Figure 3 and derive a negated FSM M' as shown in Figure 4.

Since the FSM M' is a negation of M , all valid strings generating from regular expression r will not arrive at the finite states and would be rejected, while all incorrect ones would be accepted in machine M' . Next, the graph constructor builds graphs based on the machines M and M' ; nodes in both graphs are corresponding to states of the two FSMs, and edges represent the transition between states. The two graphs are then passed to the graph processor for extracting valid and invalid patterns from the graphical analysis.

According to the previous steps, we construct two graphs (for valid and invalid test inputs) based on FSM of the input regular expression r . However, we find that the derived graphs would be much more complicated if the regular expressions become complex. Traversing every path is not practical due to the increasing amount of the computing overhead. To effectively and efficiently generate test patterns, we next apply different algorithms to simplify graphs and to extract paths that correspond to both valid and invalid test patterns.

4.2. Path Selection. Since loops may exist in the graph and will result in infinite possible paths, we simplify the graph and then extract all the paths from the simplified subgraph. We first identify the strongly connected components (SCCs) in the graph using Tarjan’s algorithm [39]. Then we regard each SCC as a pseudonode and transform the FSM graph into a directed acyclic graph (DAG) consisting of some SCCs. We next apply different algorithms for selecting paths inside the SCC and the DAG, respectively.

4.2.1. Path Selection on SCC. We start by extracting paths inside SCC. In the simplified graph, except the SCC which contains the initial node, every SCC should have one or more inward nodes, which own inward edges from other SCCs. Every SCC may or may not contain outward nodes, which have outward edges to other SCC. Our algorithm extracts paths between each inward node and outward node pair. To ensure that we could apply algorithms on every SCC, we treat the initial node of the FSM as an inward node and final nodes as outward nodes. Our algorithm contains two traversal approaches and we presented them as follows:

- (i) Shortest path: in this method, we discover the shortest path from an inward node to an outward

FIGURE 2: The work flow of our test pattern generator based on FSM.

FIGURE 3: The FSM transformed from regular expression /a@b+/.

FIGURE 4: The *negation* of the FSM transformed from regular expression /a@b+/.

node (as every edge has an equal weight in the SCC, we traverse the path from an inward node to an outward node containing the least edges). The main advantage of using the shortest path method is that we could derive simpler test patterns. We use the breadth-first searching (BFS) algorithm (https://en.wikipedia.org/wiki/Breadth-first_search. Online; accessed on Aug. 1, 2017) to extract the shortest path.

- (ii) **All-nodes-covered path:** to generate test patterns which are in more complicated forms, we also discover ways that should traverse all states inside the SCC, as the all-nodes-covered paths. Our algorithm for constructing such paths is described as follows: (1) We start from the inward node and randomly choose an outgoing edge of it. (2) Start from the current node and repeat step (1) until all nodes have been

Input:

The inward node in an SCC $\mathcal{I}$;
 The outward node in an SCC $\mathcal{O}$;
 The node set of an SCC $\mathcal{N}$;
 The edges in an SCC $\mathcal{E}$;

Output:

The path containing a list of nodes $\mathcal{L}$;

- (1) current_node $\mathcal{C} = \mathcal{I}$;
- (2) current_path $\mathcal{L} = [\mathcal{I}]$;
- (3) traversed_node $\mathcal{T} = [\mathcal{I}]$;
- (4) **while** $\mathcal{T} \neq \mathcal{N} \wedge \mathcal{C} \notin \mathcal{O}$ **do**
- (5) Find edge set E containing edges starting at node $\mathcal{C}$;
- (6) Randomly choose an edge $(\mathcal{C}, \mathcal{X})$ from E ;
- (7) Append $\mathcal{X}$ to $\mathcal{L}$;
- (8) Add $\mathcal{X}$ into $\mathcal{T}$;
- (9) current_node $\mathcal{C} = \mathcal{X}$;
- (10) **return** $\mathcal{L}$;

ALGORITHM 2: All-nodes-covered path identification.

covered. (3) Traverse to an outward node. We present the algorithm as in Algorithm 2. However, to avoid possible exhausted traversals, we also try a heuristic described as follows. Instead of selecting an edge randomly, we give priorities to the outgoing edges which lead to an uncovered node. For example, as in Figure 5, currently, we have traversed the following path $[0, 1, 2, 3, 1]$, and we have two edges for selection, $[1, 2]$ and $[1, 4]$. Because node 4 has not been covered in the current traversal, we give priority to $[1, 4]$ to accelerate the process to include all nodes in the graph.

Figure 5 presents a sample SCC consisting of 5 states. From the above path selection algorithms, we list a series of sample paths extracted:

- (1) *Shortest path*
 $[0, 4, 5]$ and $[0, 1, 2]$
- (2) *All-nodes-covered path*
 $[0, 1, 2, 3, 1, 4, 5]$ and $[0, 1, 2, 3, 1, 4, 5, 0, 1, 2]$

FIGURE 5: Sample paths inside an SCC.

4.2.2. Path Selection on DAG. Next, we discuss paths between SCCs. In the simplified FSM graph, representing as a DAG, each node in the DAG is an SCC. An SCC containing initial nodes is called an initial SCC. If an SCC contains final nodes, it is regarded as a final SCC. Our goal is to extract paths from the initial SCC to each final SCC. Since there is no loop in the DAG, we can traverse all possible paths in the DAG. However, even though the DAG has fewer edges than the original graph, there are still too many edges for us to traverse. To address the issue, we propose two methods to extract representative paths within the DAG. Our algorithm is described below.

- (i) **Tree branch:** DAG and tree have similar structures, and the primary difference between them is that nodes in a DAG could have multiple parents. In this method, we transform a DAG into a tree by taking the initial SCC as the root and construct a tree using BFS and depth-first searching (DFS). We next extract paths from the root to each leaf in the DAG. Once a leaf is not one of the final SCCs, we extend the path by adding an edge between the leaf to another branch that has a final SCC on the leaf node.
- (ii) **All-edges-covered tree:** since some edges would be discarded during tree construction, while mutating, some patterns might not be covered. To address the issue, we implement a reinforced algorithm. After the tree construction according to BFS and DFS, we identify all discarded edges and then create additional paths which are composed of the shortest path between an initial node, end nodes of the discarded edges, and a final node. Therefore, we seek to cover all edges at least once in the DAG.

4.3. Test Pattern Generator. After we obtain paths from the graph, we traverse these routes to generate test patterns. Each path could be transformed into a corresponding regular expression with each edge as an alphabet set. For example, $/[a-z][^@][a-zA-Z]/$ represents a path that generates invalid strings for the *email* attribute. $/[a-z]/$, $/[^@]/$, and $/[a-zA-Z]/$ are alphabet sets which are represented as edges

on the path. Because the length of the extracted path is finite, we can generate all possible strings from a path. However, because the same category of vulnerabilities may be identified by inputs of the same path, for our evaluation experiment, we randomly choose one alphabet in each alphabet set and generate one string for one path to measure the effectiveness of our method.

5. Evaluation

In this section, we evaluate the performance of our FSM-based pattern generation algorithm by implementing it in the HTML5 fuzzer. We generate a series of validators for HTML5 input strings systematically to imitate the correct and misdesigned scenarios by engineers. In the experiment, every validator is a regular expression to be mutated or not. Thus, some validators have vulnerabilities that could lead to injection threats. Next, our fuzzer generates test cases and injects them to the validators. If the injected invalid strings are not well handled by the validators, we consider that some vulnerabilities exist on the website. We discuss the performance of our fuzzer according to the detection rate and time consumption.

5.1. Abnormal Validators. Our evaluation initially starts by constructing a series of abnormal validators. As we have analyzed vulnerabilities of injection attacks on webpages, we observe that many injection vulnerabilities can be attributed to the fact that the string validators on the web servers are not fully constrained. These validators work properly for valid inputs and can identify some typical malicious inputs; however, under some circumstances or in extreme cases, they accept a portion of incorrect strings. These vulnerabilities are common when web developers do not apply extensive and dependable restrictions in designing string validators. To imitate the vulnerabilities and generate abnormal validators, we mutate the strongly restricted validators (presented as regular expressions), which are shown in Table 3, into more tolerant (misdesigned) ones. To achieve this, we separate the elements in a regular expression into two categories, the alphabet set (e.g., $/[0-9a-z]/$) and the frequency (e.g., $/\{3,5\}/$). For each element, we have two choices, mutate it or not. Therefore, for a regular expression consisting of n elements, we have a total of 2^n mutation choices. To generate misdesigned validators, we first select some elements to mutate, from 6 to 10. Next, we mutate the alphabet set element by expanding the range of the characters set. For the frequency element, we mutate it by adding the frequency. For example, if the correct regular expression is $/[a-z]\{3,5\}/$, our algorithm will mutate it into $/[a-z]/$, $/\{3,5\}/$ or $/[a-z]*/$. By this approach, we can generate a series of misdesigned validators containing vulnerabilities for evaluation.

5.2. Experiment Setup. We set up our experiment on two computers; both of them have Intel Core i7-6700 CPU and 64 GB memory. We establish the abnormal websites on a computer and install our framework on the other one. Next, we use four different types of input attributes in HTML5, *date*, *email*, *number*, and *time*, for testing. We consider that these

TABLE 3: The well-designed regular expressions for validating four HTML5 input types.

Attribute	Sample regular expression for validation
<i>Date</i>	<code>/((([1-9]*[0-9]{4,})-([013578] 1[02])-([01-9] 12 [0-9] 3[01])) ([1-9]*[0-9]{4,})-([013456789] 1[012])-([01-9] 12 [0-9] 30)) ([1-9]*[0-9]{4,})-02-([01-9] 1[0-9] 2[0-8])) ([1-9]*([13579][26] 2468)[048])00-02-29) ([1-9]*[0-9]{2}([13579][26] 2468)[048][04 08]-02-29))/</code>
<i>Email</i>	<code>/[a-zA-Z0-9.!#\$%&'*/=?\^_{}~\-\+@][a-zA-Z0-9]([a-zA-Z0-9\-\+@]{0,61}[a-zA-Z0-9])?(\.[a-zA-Z0-9]([a-zA-Z0-9\-\+@]{0,61}[a-zA-Z0-9])?)*/</code>
<i>Number</i>	<code>/[-+]?[0-9]*([\.[0-9]*)?([eE]([-+]?[0-9]*)?)?/</code>
<i>Time</i>	<code>/([01]?[0-9] 2[0-3]):[0-5][0-9](:[0-5][0-9](\.[0-9]{1,3})?)?)?/</code>

four types that contain numerical-value-only expressions (*date*, *time*) and mixtures of numerical value and alphabet set expressions (*email*, *number*) are suitable for us to evaluate our system's performance. We list the regular expression used in our experiment in Table 3. These expressions are referenced from the HTML5 W3C document [40] and other suggestions from a popular online forum for program developers (<https://stackoverflow.com/questions/14772142/24-hour-time-regex-for-html-5>. Online; accessed on Aug. 1, 2017, <https://stackoverflow.com/questions/10925710/regex-for-date-dd-mm-yyyy>. Online; accessed on Aug. 1, 2017).

From the four expressions of HTML5 attributes, we generate weak validators and test cases for the experiment. We set the number of elements to mutate in a regular expression as 6, 8, and 10. Thus, we will retrieve $2^6 - 1$, $2^8 - 1$, and $2^{10} - 1$ validators as misdesigned validators (and one correct validator). We next use the four regular expressions to generate 40 test patterns (paths in a graph) for each combination of the path extraction algorithms (e.g., shortest path + BFS). From these patterns, we generate one test string from each test pattern as test cases for fuzzing.

5.3. Fuzzing Performance. We conduct the experiment 50 times and examine the performance of our system according to two metrics: (1) the average detection rates of identifying vulnerabilities using test strings generated by our framework and (2) the total time spent for testing a website.

5.3.1. Detection Rates. We present the detection rates of test patterns from different combinations of path selection methods on the SCC and the DAG, as shown in Figure 6. From the results, we observe that the system detects more than 80% of misdesigned validators. We assert that the results prove that our fuzzer could achieve certain accuracy by using only 40 strings generated by 40 test patterns. In addition, when the input types become simpler (e.g., *number* and *time*), our system reaches even higher detection rates (more than 95%). Furthermore, for those validators containing more elements to be mutated (number of elements = 10), our fuzzer also achieves higher detection rates.

To understand the weakness of our generation algorithm, we investigate those vulnerable validators which are not detected by our test strings. From the investigation, we observe that those undetected validators are due to the lack of coverage of our patterns. As these patterns are based on the paths extracted from the graph by our proposed methodologies (e.g., shortest path + BFS), some

vulnerabilities are not covered in the paths obtained by our algorithms; consequently, our system is unable to identify such weaknesses during the experiment.

5.3.2. Time Spent in Generating Cases. Next, we investigate the time consumption of our test case generation. Table 4 demonstrates the time spent on the major steps, including transforming correct regular expressions into corresponding FSM, taking negation of the FSM, extracting paths, and generating patterns. From the table, we observe that the *email* type requires much more time than other kinds. The results are reasonable as the regular expression of the *email* is more complicated than the other three attributes. As shown in Table 3, we can observe that the *email* and *date* attributes are more complex than the *time* and *number* attributes. The complexity of expressions corresponds to the time consumption in our experiment.

5.3.3. Time Spent in Fuzzing Website. In addition, we also measure the total time spent in conducting a fuzzing test to an HTML5 website containing four designated input types (*date*, *email*, *number*, and *time*). We mutate each regular expression listed in Table 3 into 63, 255, and 1,023 mutated validators for testing. Then we construct three websites of different scales containing 63, 255, and 1,023 webpages for evaluation. Each webpage contains four validators corresponding to the four types. We use our fuzzing framework to test the three websites. From the results, the average times spent for fuzzing a website containing 63, 255, and 1,023 webpages are 4.49 minutes, 13.63 minutes, and 46.23 minutes, respectively. If we add the previous time for generating test cases, the total time required for conducting a complete fuzzing test to 1,024 webpages is less than one hour.

Compared with previous studies using generation-based approaches, our methodology almost waives the effort for designing a series of generation patterns. For example, if we recruit an engineer and the engineer could develop ten fully considered generation patterns per hour, we still need several hours to several days to produce the generation patterns for different types of input attributes. This work is extraordinarily exhaustive for those complicated attributes such as *email*. In contrast, our system completes the task in minutes and achieves high quality.

6. A Real Case

In addition to the simulation test, we use our fuzzer to investigate whether any popular websites have potential

FIGURE 6: Detection rates of different input attributes in HTML5.

TABLE 4: Average time consumption of generating test cases of different input types in HTML5.

Type	Regex to FSM (s)	# states	Negation & generation (s)
Date	0.793	61	3.501
Time	0.031	14	3.028
Email	230.731	127	27.286
Number	0.010	6	3.001

vulnerabilities. We take the HTML section of W3Schools Online Web Tutorials (<https://www.w3schools.com/html/>. Online; accessed on Aug. 1, 2017) as a target because they provide a series of try-out pages for the HTML5 syntax.

By traversing the website, our system successfully visits 431 pages. There are 55 pages which have input forms.

To demonstrate the capability of our system, we inject valid and invalid strings toward these pages with input forms and observe the responses from these pages. We find that the similarity of responses of valid and invalid inputs from these pages is all over 0.85. As we set the threshold as 0.8, our system considers these pages vulnerable.

We next check the pages manually to investigate their weakness. From our investigation, most of the pages contain a submission form of HTML input attributes (https://www.w3schools.com/html/tryit.asp?filename=tryhtml_input_month. Online; accessed on Sept. 25, 2017, https://www.w3schools.com/html/tryit.asp?filename=tryhtml_input_number. Online;

accessed on Sept. 25, 2017). Once the invalid strings are injected through HTTP request and passed to the server, the server would encounter potential threats. From the real case, we demonstrate the capability of our system in fuzzing current websites.

7. Conclusion

To identify the vulnerabilities in websites efficiently and effectively, in this study, we present a fuzzing framework and apply it to websites containing HTML5. We design a FSM-based algorithm and incorporate the graphical analysis to systematically and automatically generate test cases for fuzzing. By our design, the algorithm could be adopted not only for HTML5 websites but also for targets that rely on regular expressions to validate input strings. The scheme is beneficial for engineers to conduct a pilot examination without requiring extensive knowledge of the test target. Also, we automate the procedure on the result filtering to offload the human efforts on reviewing large-scale testing results. From our evaluation, our system achieves high detection rates in different kinds of HTML5 attributes; also, the time spent in conducting large-scale website fuzzing is significantly reduced by our framework.

To conclude this study, our contributions are threefold:

- (i) We develop an HTML5 webpage crawler and analyzer to investigate potential injection vulnerabilities of websites.
- (ii) We present a FSM-based test pattern generation algorithm for fuzzing.
- (iii) We automate the result filtering and reduce the effort on manual review.

Even though we have proposed and implemented a fuzzing framework, there are some issues remaining to be addressed in the future. The issues include (1) improving the performance and the test case coverage of fuzzing on different targets, (2) providing a more user-friendly interface for conducting a fuzzing test, and (3) enhancing the traversal technique in our system to crawl websites with dynamic pages. From this study, we hope our approach could provide a user-friendly fuzzing procedure for identifying vulnerabilities among modern websites and ease both mental and physical loading of conducting a large-scale test.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by Ministry of Science and Technology, Taiwan, under Grant MOST 106-3114-E-002-005.

References

- [1] S. Son and V. Shmatikov, "The postman always rings twice: attacking and defending postmessage," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS '13)*.
- [2] W. West and S. M. Pulimood, "Analysis of privacy and security in html5 web storage," *Journal of Computing Sciences in Colleges*, vol. 27, no. 3, pp. 80–87, 2012.
- [3] M. Heiderich, T. Frosch, M. Jensen, and T. Holz, "Crouching tiger—hidden payload: security risks of scalable vectors graphics," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pp. 239–250, ACM, October 2011.
- [4] S. Mansfield-Devine, "Divide and conquer: the threats posed by hybrid apps and HTML 5," *Network Security*, vol. 2010, no. 3, pp. 4–6, 2010.
- [5] Y. Demchenko, L. Gommans, C. De Laat, and B. Oudenaarde, "Web services and grid security vulnerabilities and threats analysis and model," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pp. 262–267, IEEE Computer Society, November 2005.
- [6] D. Hoffman, H.-Y. Wang, M. Chang, and D. Ly-Gagnon, "Grammar based testing of HTML injection vulnerabilities in RSS feeds," in *Proceedings of the Testing: Academic and Industrial Conference—Practice and Research Techniques (TAIC PART '09)*, pp. 105–110, IEEE, September 2009.
- [7] P. Kumar, "The multi-tier architecture for developing secure website with detection and prevention of sql-injection attacks," *International Journal of Computer Applications*, vol. 62, no. 9, pp. 30–36, 2013.
- [8] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: characterization, detection and mitigation," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 66–77, ACM, November 2014.
- [9] S. Gupta and B. B. Gupta, "JS-SAN: defense mechanism for HTML5-based web applications against javascript code injection vulnerabilities," *Security and Communication Networks*, vol. 9, no. 11, pp. 1477–1495, 2016.
- [10] Fuzzing—owasp, <https://www.owasp.org/index.php/Fuzzing>.
- [11] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
- [12] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Pearson Education, 2007.
- [13] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, Artech House, 2008.
- [14] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the Workshop on Programming Languages and Analysis for Security*, pp. 27–36, ACM, June 2006.
- [15] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 206–215, 2008.
- [16] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Security Symposium*, pp. 445–458, 2012.
- [17] J. Botella, B. Legeard, F. Peureux, and A. Vernet, "Risk-based vulnerability testing using security test patterns," in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 337–352, Springer, 2014.
- [18] B. P. Miller, D. Koski, C. P. Lee et al., "Fuzz revisited: a re-examination of the reliability of unix utilities and services," Tech. Rep., 1995.

- [19] C. Miller and Z. N. Peterson, "Analysis of mutation and generation-based fuzzing," 2007.
- [20] M. Widl, "Test case generation by grammar-based fuzzing for model-driven engineering," in *Proceedings of the Haifa Verification Conference*, pp. 278–279, Springer, 2013.
- [21] S. Rawat and L. Mounier, "Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: few preliminary results," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '11)*, pp. 531–533, IEEE, March 2011.
- [22] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, 1990.
- [23] D. Coppit and J. Lian, "Yagg: An easy-to-use generator for structured test inputs," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 356–359, ACM, November 2005.
- [24] B. A. Malloy and J. F. Power, "An interpretation of purdom's algorithm for automatic generation of test cases," 2001.
- [25] R. Lämmel and W. Schulte, "Controllable Combinatorial Coverage in Grammar-Based Testing," in *Proceedings of the IFIP International Conference on Testing of Communicating Systems*, pp. 19–38, Springer, 2006.
- [26] S. Y. Kim, S. Cha, and D.-H. Bae, "Automatic and lightweight grammar generation for fuzz testing," *Computers & Security*, vol. 36, pp. 1–11, 2013.
- [27] M. Islam and C. Csallner, "Generating test cases for programs that are coded against interfaces and annotations," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 3, pp. 21:1–21:38, 2014.
- [28] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp. 178–187, 1978.
- [29] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2016.
- [30] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines—a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.
- [31] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test development for communication protocols: Towards automation," *Computer Networks*, vol. 31, no. 17, pp. 1835–1872, 1999.
- [32] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 109–124, ACM, August 1994.
- [33] K. Sabnani and A. Dahbura, "A protocol test generation procedure," *Computer Networks*, vol. 15, no. 4, pp. 285–297, 1988.
- [34] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha, "Test cases generation from UML state diagrams," *IEE Proceedings Software*, vol. 146, no. 4, pp. 187–192, 1999.
- [35] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE '01)*, pp. 34–43, IEEE, 2001.
- [36] N. Li, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Reggae: automated test generation for programs using complex regular expressions," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, pp. 515–519, IEEE, November 2009.
- [37] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics—Doklady*, vol. 10, no. 8, pp. 707–710, 1966.
- [38] G. Berry and R. Sethi, "From regular expressions to deterministic automata," *Theoretical Computer Science*, vol. 48, no. 1, pp. 117–126, 1986.
- [39] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.
- [40] I. Hickson, S. Pfeiffer, T. O'Connor et al., "HTML5," W3C, W3C Recommendation, October 2014, <https://www.w3.org/TR/2014/REC-html5-20141028/>.